Type-Driven Gradual Security with References

Matías Toro, Ronald Garcia, Éric Tanter

Type-Driven Gradual Security with References

MATÍAS TORO, PLEIAD Laboratory, Computer Science Department (DCC), University of Chile
RONALD GARCIA, Software Practices Laboratory, University of British Columbia
ÉRIC TANTER, PLEIAD Laboratory, Computer Science Department (DCC), University of Chile

In security-typed programming languages, types statically enforce noninterference between potentially conspiring values, such as the arguments and results of functions. But to adopt static security types, like other advanced type disciplines, programmers face a steep wholesale transition, often forcing them to refactor working code just to satisfy their type checker. To provide a gentler path to security typing that supports safe and stylish but hard-to-verify programming idioms, researchers have designed languages that blend static and dynamic checking of security types. Unfortunately most of the resulting languages only support static, type-based reasoning about noninterference if a program is entirely statically secured. This limitation substantially weakens the benefits that dynamic enforcement brings to static security typing. Additionally, current proposals are focused on languages with explicit casts, and therefore do not fulfill the vision of gradual typing, according to which the boundaries between static and dynamic checking only arise from the (im)precision of type annotations, and are transparently mediated by implicit checks.

In this paper we present GSL_{Ref}, a gradual security-typed higher-order language with references. As a gradual language, GSL_{Ref} supports the range of static-to-dynamic security checking exclusively driven by type annotations, without resorting to explicit casts. Additionally, GSL_{Ref} lets programmers use types to reason statically about termination-insensitive noninterference in *all* programs, even those that enforce security dynamically. We prove that GSL_{Ref} satisfies all but one of Siek *et al.*'s criteria for gradually-typed languages, which ensure that programs can seamlessly transition between simple typing and security typing. A notable exception regards the dynamic gradual guarantee, which some specific programs must violate if they are to satisfy noninterference; it remains an open question whether such a language could fully satisfy the dynamic gradual guarantee. To realize this design, we were led to draw a sharp distinction between syntactic type *safety* and semantic type *soundness*, each of which constrains the design of the gradual language.

CCS Concepts: • **Security and privacy → Information flow control**; • **Theory of computation → Type structures**; **Program semantics**;

Additional Key Words and Phrases: Noninterference, language-based security, gradual typing

1 INTRODUCTION

Gradual typing is typically viewed as a means to combine the agility of dynamic languages, like Python and Ruby, with the reliability of static languages, like OCaml and Scala [Siek and Taha 2006]. But static and dynamic are merely relative notions, and several researchers have explored a more relativistic view. For example, Disney and Flanagan [2011] and Fennell and Thiemann [2013] develop languages where only information-flow security properties are enforced using both dynamic and static checking; Bañados Schwerter et al. [2014, 2016] develop a language where only computational effect capabilities are gradualized; Lehmann and Tanter [2017] gradualize only the logical assertions of refinement types; and Jafery and Dunfield [2017] gradualize only refinements of sum types. In each of these cases, the "fully-dynamic" corner of the gradual language is not dynamic by typical standards, but rather simply typed. Nonetheless, each language supports migration toward a richer typing discipline that subsumes simple typing.

This paper revisits gradual information-flow security typing, with a particular focus on the strong information-flow guarantees that security types have historically implied. We describe a new language, GSL_{Ref}, that introduces a *type-driven* conception of gradual security. Unlike most

To appear in ACM Transactions on Programming Languages.
This work is partially funded by CONICYT FONDECYT Regular Project 1150017.

prior work, GSL_{Ref} supports the same static, type-based reasoning about information-flow for gradually-typed programs as SSL_{Ref}, its purely static counterpart. To explain this innovation, we review the power of static security types and then show what it means to preserve type-based reasoning power in a gradual language.

Static security typing. Consider a program that processes employee data:[1]

```
1 let age = 31
2 let salary = 58000
3 let intToString : Int → String = ...
4 let print : String → Unit = ...
5 print(intToString(salary))
```

The program is well-typed, but it has a significant error that simple types do not catch: if salaries are confidential and printing is publicly observable, then this program leaks confidential data.

Information-flow security typing lets a programmer statically classify program entities according to a lattice of *security labels* [Denning 1976] and rely on type-checking to prevent information leaks. One exemplar security lattice, which we use as a running example, is the U.S. Dept of Defense classification scheme: Unclassified $\preccurlyeq$ Confidential $\preccurlyeq$ Secret $\preccurlyeq$ Top Secret, which we simplify to $\bot \preccurlyeq \mathsf{L} \preccurlyeq \mathsf{H} \preccurlyeq \top$, denoting minimum, low, high, and maximum security respectively [Zdancewic 2002]. To inform static type checking, each type constructor is statically annotated with a security label (*e.g.* $\mathsf{Int}_{\mathsf{L}}$); source program values are also annotated to unambiguously determine their static security (*e.g.* 58000_{H} has type $\mathsf{Int}_{\mathsf{H}}$). Security label ordering induces a natural subtyping relation (*e.g.* $\mathsf{Int}_{\mathsf{L}} <: \mathsf{Int}_{\mathsf{H}}$ and $\mathsf{Int}_{\mathsf{H}} \to_{\mathsf{L}} \mathsf{String}_{\mathsf{L}} <: \mathsf{Int}_{\mathsf{L}} \to_{\mathsf{H}} \mathsf{String}_{\mathsf{H}}$), which denotes security-respecting substitutability. An attacker or observer at level ℓ_o can discriminate values that have security level at most ℓ_o. Armed with security types and subtyping, an information-flow security type system statically ensures that high-confidence data may not flow directly or indirectly to low-confidence channels [Volpano et al. 1996].

In the example above, if we annotate the `salary` as high-security data (of type $\mathsf{Int}_{\mathsf{H}}$), and specify that `print` takes a low-security argument (of type $\mathsf{String}_{\mathsf{L}}$), then our operational intuition tells us that the program cannot satisfy these directives: it should be rejected. Before the type system can confirm our intuitions, though, we must determine the security levels of *every* type in the program. In SSL_{Ref}, our static language, this means that every type and value must be annotated. While security label inference and polymorphism [Myers and Liskov 2000] can reduce this burden, one cannot experiment with *some* security levels without first determining *all* security levels. Once all security types are assigned, the static type system forbids passing a high-security value to a function that expects a low-security argument, so the type checker rejects the program. GSL_{Ref} conservatively extends this model to support incremental and localized adoption of security types.

Security types induce free noninterference theorems. The employee data example demonstrates a simple security leak, where high-security data flows directly to a low-security channel. But security types must also contend with sophisticated leaks, where low-security variables may change control-flow through high-security code and mutable state can enable implicit security leaks [Denning 1976]. To combat this, information-flow security languages enforce a general property called *noninterference*, which guarantees that high-security inputs do not affect low-security results [Goguen and Meseguer 1982]. Noninterference clearly subsumes our simple security leak, but it also prevents implicit and control-based leaks, where an attacker attempts to use low-security inputs and outputs to learn about high-security data.

[1] Adapted from [Disney and Flanagan 2011].

In security-typed languages, higher-order security types denote *modular* guarantees about noninterference [Heintze and Riecke 1998]. In particular, they use Reynold's theory of parametricity [Reynolds 1983] to ensure that a typing judgment dictates how replacing inputs can affect the resulting output [Abadi et al. 1999]. For example, consider a hypothetical function:

```
let mix : Int_L →_L Int_H →_L Int_L = fun pub priv => ...
```

At first sight, it appears to "mix" its arguments `pub` and `priv` to produce some result. However, the security annotations on its type guarantee that the integer result *cannot* leak information about `priv`, no matter what value is given to `pub`. The key to this result is how the relevant typing judgment is interpreted. The body of the `mix` function, t, must satisfy the typing judgment $\texttt{pub} : \mathsf{Int}_\mathsf{L}, \texttt{priv} : \mathsf{Int}_\mathsf{H} \vdash t : \mathsf{Int}_\mathsf{L}$. To endow this judgment with meaning, a logical relation-based semantic model is defined directly in terms of the language's dynamic semantics. According to this *semantic typing judgment*, changing the value of `priv` has no effect on the final value of t. This guarantee holds even if `mix` uses mutable state [Zdancewic 2002]. The end result is that an attacker with no direct access to a high-security channel cannot manipulate the value of `pub` to uncover the value of `priv`, even by modifying `mix`'s implementation.

In a static security language, these noninterference guarantees follow from the type structure of the language. No runtime checks are required, and the security labels applied to values and types are simply static annotations.[2] In essence, static security types induce *free theorems* about the noninterference behaviors of computations, just as parametric polymorphic types induce free theorems about data abstraction [Wadler 1989]. Free noninterference theorems provide enormous benefits to programmers. First, they support *modular* reasoning about noninterference: a programmer who implements a higher-order function with type $(\mathsf{Int}_\mathsf{L} \to_\mathsf{L} \mathsf{Int}_\mathsf{H} \to_\mathsf{L} \mathsf{Int}_\mathsf{L}) \to_\mathsf{L} \mathsf{Bool}_\mathsf{H}$ knows that the function's body can safely call its argument with high-security data as the second argument: the provided function cannot leak that data. Second, type-based reasoning is *compositional*: the syntactic typing rules precisely specify how the security properties of subprograms (*e.g.* a function-typed expression and a potential argument) compose to determine security properties of a larger program (*e.g.* via function application). Finally, this reasoning is *static*: one need not reason directly about operational behavior or data flow to understand security. That reasoning was done once-and-for-all in the type-driven noninterference proof. Instead, type structure guides reasoning. These properties are especially useful for partial programs like software libraries. Below we show that $\mathsf{GSL}_\mathsf{Ref}$ preserves these advantages while introducing new flexibility by dynamically enforcing some type guarantees.

Relaxing security typing. Like any static type discipline, security typing has its downsides. As discussed above, security typing cannot be checked until all types are given a security level, through ascription, polymorphism, or inference. One cannot incrementally add security levels and observe the consequences. In addition, verifying noninterference is in general undecidable, so static security checking is necessarily conservative, and as a result programmers must sometimes refactor perfectly safe and clear code simply to appease the type checker.

To address these shortcomings, researchers have explored ways to combine static and dynamic security checking. These approaches can be classified roughly as *hybrid* or *gradual*. Hybrid approaches, e.g. [Buiras et al. 2015; Chandra and Franz 2007; Shroff et al. 2007; Zheng and Myers 2007], blend various static analysis and runtime monitoring techniques to make analyses more precise, to incorporate dynamically-defined policies, and to target safe *executions* rather than just safe *programs*. Gradual approaches [Disney and Flanagan 2011; Fennell and Thiemann 2013, 2016],

[2] Like type annotations, security labels appear in dynamic semantics solely to prove type safety: they are erased in a practical runtime.

inspired by gradual typing, focus on type systems for static analysis and add the extra goal of enabling seamless incremental evolution from programs with no information-flow control whatsoever to programs with security-type based static enforcement, while fulfilling the goals of hybrid approaches.

To clearly understand the contribution of the present work, it is important to clarify that the prior work in this space, hybrid and gradual alike, take a *check-driven* approach to analysis: the core of the security model is based on associating a security level to each *value* in a program and managing security levels using two distinct operations: security *upgrades* and *checks*. A security upgrade elevates a value's security label, *e.g.* $(\mathsf{Int}_\mathsf{H}!)5_\mathsf{L} \longrightarrow 5_\mathsf{H}$. A security check signals an error if the checked label is not at least as high as the value's tag, *e.g.* $(\mathsf{Int}_\mathsf{H}?)5_\mathsf{L} \longrightarrow 5_\mathsf{L}$, but $(\mathsf{Int}_\mathsf{L}?)5_\mathsf{H} \longrightarrow$ **error**. Upgrades and checks have different dynamic behavior, but with help from static typing, gradual security languages combine them into type-based *upcasts* and *downcasts*, *e.g.* $(\mathsf{Int}_\mathsf{L})t$, which checks t if L is lower than t's static security and upgrades t otherwise. This approach easily detects direct flows of high-security values to low-security channels, but preventing implicit flows through control transfer requires extra care, including prophylactic upgrades to program values [Chandra and Franz 2007] and policies to restrict upgrades [Fennell and Thiemann 2013]. As we will see, our development similarly requires careful treatment of assignments.

Check-driven approaches break free theorems. Dynamic security casts give flexibility to programmers, but fundamentally cripple the ability to reason statically using security types. In particular, if security downcasts are added to the language, although noninterference is still preserved, static type judgments no longer imply free theorems about security of programs, as was discussed above. As a result, programmers must reason about the *dynamic semantics*—dynamic labels, dynamic upgrades, and dynamic checks—to uncover which values do not interfere with one another. In particular, a function's type no longer denotes noninterference properties about its arguments and results. For example, consider the function:

```
let mix : Int_L →_L Int_H →_L Int_L =
  fun pub priv => if pub < (Int_L)priv then 1_L else 2_L
```

This program is statically accepted by languages that only check for compatibility of base types [Disney and Flanagan 2011; Fennell and Thiemann 2013]. The type of `mix`, while fully static, does not guarantee that `mix` never reveals information about its second argument. Rather, the type merely guarantees that the second argument's security level is *at most* H and the result is *at most* L. But upper-bounds on security labels do not suffice to make definitive assertions about the noninterference behavior of this function.[3] Indeed, the program `mix` 1_L 5_L successfully reduces to 1_L. In order to avoid such behavior, the programmer must *explicitly* upgrade the dynamic security level of the value passed as second argument at each call site. Alternatively, one can upgrade `mix` to its *own* type, thereby forcing the second argument to be upgraded before executing the function body (and hence preventing any information leak about that argument). This highlights the fact that *types* alone do not denote noninterference properties: the two versions of the `mix` function behave differently although they have the same type.

This phenomenon, that adding dynamic checking to a static system may weaken type-based reasoning principles, is not unique to security typing. Prior work on cast calculi with parametric polymorphism observes that adding runtime type tests to System F preserves *type safety*—*i.e.* that programs do not crash—but sacrifices *type soundness*—*i.e.* that polymorphic types denote strong data abstraction guarantees via parametricity [Ahmed et al. 2011, § 5.1].

[3] Recent work by Fennell and Thiemann [2016] on LGJS addresses this particular problem, as described in Sec. 7.

Contribution: type-driven gradual security typing. Modular, compositional, and type-based reasoning are hallmark benefits of type systems. Thus, to facilitate the seamless transition toward static security typing, the typing judgment of a gradual type system should imply the same semantic invariants that its fully-static counterpart does. To that end, this paper presents GSL_{Ref}, a *type-driven* gradual security language that extends a static security type discipline with gradual security labels and corresponding notions of *gradual type precision* and *consistent subtyping*. To secure GSL_{Ref} programs, one just adds static security labels: dynamic checks arise automatically and implicitly, as needed to enforce the noninterference guarantees denoted by static types.

Unlike most prior work, GSL_{Ref}'s static security types denote the same noninterference guarantees as its fully static counterpart language SSL_{Ref}. As such, GSL_{Ref}'s security types enable modular and compositional type-based reasoning about noninterference, just like the fully static SSL_{Ref}, whereas security types in most prior gradual languages do not. GSL_{Ref}'s type system supports reasoning about termination-insensitive noninterference because it is sound with respect to a security logical relation defined directly in terms of type structure. This result is standard for a purely-static security language [Heintze and Riecke 1998], but novel for a gradual security language with imprecise types supported by dynamic checks. In fact the dynamics are guided by the needs of the noninterference proof.

To summarize, this work makes the following contributions:

- We present GSL_{Ref}, a gradual security language that supports seamless transition between simply-typed and security-typed programming. Security typing annotations alone drive the balance between static and dynamic information flow checking. (Sec. 4)
- We prove that GSL_{Ref}'s type discipline enforces termination-insensitive noninterference: GSL_{Ref}'s types reflect strong information-flow invariants that hold even in code that contains gradually-typed subexpressions. (Sec. 5)
- We prove the static gradual criteria of Siek et al. [2015]. Interestingly, in order to ensure noninterference in presence of references (and hence implicit flows through the heap), GSL_{Ref} sacrifices the dynamic gradual guarantee.
- We contribute more generally to the foundations of gradual typing for advanced type disciplines. We find that GSL_{Ref}'s security invariants require separate consideration of syntactic type *safety* and semantic type *soundness*, each of which constrains the design of the gradual language.
- This work also represents a particularly challenging application of the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016]. AGT is a framework that uses abstract interpretation [Cousot and Cousot 1977] at the type level to systematically construct gradually-typed languages from pre-existing statically typed ones. We report on our experience with a number of important considerations that complement the original presentation of AGT. In addition, we highlight the limitation of AGT when applied to semantically-rich type disciplines. (Sec. 6)

Before diving into the development of GSL_{Ref}, Sec. 2 informally introduces the type-driven approach to gradual security typing through examples. Then, Sec. 3 presents SSL_{Ref}, the fully-static security type language from which GSL_{Ref} is derived. Supplementary definitions can be found in the Appendix. Complete definitions, as well as the proofs of all the results stated in the paper, can be found in the companion technical report [Toro et al. 2018]. An interactive executable model of GSL_{Ref} is available online at https://pleiad.cl/gradual-security/.

2 TYPE-DRIVEN GRADUAL SECURITY TYPING IN ACTION

Static security type systems impose a burdensome all-or-nothing adoption model: all security types must be determined before the type system can check security. Even then, some secure programs have no statically-checkable type assignment, or may require substantial refactoring to satisfy the type checker. *Gradual security typing* addresses these shortcomings by enabling a programmer to incrementally add security information to the program, progressively introducing dynamic and static checks and guarantees.

Let us consider how gradual security typing can progressively introduce security guarantees and help detect and fix bugs in our first example from Sec. 1. Recall the problem with the program: `salary` is a high-security value, but `print` is a low-security channel. We can statically reflect these intentions:

```
1  let age = 31?
2  let salary = 58000H
3  let intToString : Int? →? String? = ...
4  let print : StringL →? Unit? = ...
5  print(intToString(salary))
```

In practice the programmer just marks the value of `salary` and the input type of `print`: all omitted security annotations desugar to the *unknown* security label ?. Under our gradual security semantics, this program type checks, but triggers a runtime check failure at line 5. If the highlighted annotations were omitted or ?, then the program would check and run exactly as a simply-typed one, because it would not impose, and thus not enforce, any security invariants.

How do we repair this program? Simply adding more annotations cannot fix it. Case in point, adding a reasonable security annotation to line 3 escalates the runtime failure to a static type error.

```
3  let intToString: IntL → String = ...
```

If the security annotations are as intended, however, then the runtime error must be due to some behavioral bug in the program (*e.g.* the programmer might have intended to print the employee's `age` instead).

Reasoning with imprecision. The gradual type checker statically enforces the invariants it can, deferring checks to runtime when the static type information is insufficient. Rather than introducing dynamic casts, as in the check-driven approach, our *type-driven* approach to gradual security typing builds on foundations laid by prior research on gradual typing. Siek and Taha [2006] observe similar difficulties as in the check-driven approach when trying to use subtyping to combine dynamic and simple type checking. This inspired gradual typing, which extends static types with an *unknown type* to form *gradual types*, relating them to one another using *consistency* and *precision* relations [Siek et al. 2015]. Since these notions are conceptually orthogonal to subtyping, they blend well with pre-existing subtyping disciplines [Siek and Taha 2007]. Our type-driven approach adapts these concepts to gradual security and its natural notion of subtyping.

In this model, the *unknown label* ? represents imprecise security information. Precision $\sqsubseteq$ is a partial order from more-precise labels to less-precise labels: static security labels are perfectly precise, *e.g.* $\mathsf{H} \sqsubseteq \mathsf{H}$, while ? denotes utter imprecision, *e.g.* $\mathsf{H} \sqsubseteq ?$. Precision extends *covariantly* to security types, *e.g.* $\mathsf{Int}_{\mathsf{H}} \to \mathsf{Int}_{\mathsf{L}} \sqsubseteq \mathsf{Int}_{?} \to \mathsf{Int}_{?}$, in contrast to subtyping.

The ordering on security labels $\preccurlyeq$ consequently extends to *consistent ordering* $\widetilde{\preccurlyeq}$ on gradual labels. Consistent ordering preserves every order relation among precise labels (*e.g.* $\bot \mathrel{\widetilde{\preccurlyeq}} \top$ and $\top \mathrel{\widetilde{\not\preccurlyeq}} \bot$), but mathematically, it is not an ordering relation (*e.g.* both $? \mathrel{\widetilde{\preccurlyeq}} \top$ *and* $\top \mathrel{\widetilde{\preccurlyeq}} ?$). Rather, it reflects consistent reasoning in the face of imprecise information: since we do not know what label ? represents, either static order is *plausible*. Consistent ordering induces an analogous notion of

consistent subtyping, *e.g.* $\mathsf{Int}_\top \lesssim \mathsf{Int}_?$ *and* $\mathsf{Int}_? \lesssim \mathsf{Int}_\bot$, which is not transitive, *e.g.* $\mathsf{Int}_\top \not\lesssim \mathsf{Int}_\bot$, so it is not a subtyping relation, but embodies imprecise reasoning about static subtyping [Siek and Taha 2007]. An attacker or observer at level ℓ_o can now also observe values that have unknown security levels, as long as the dynamic security information about the value is observable at ℓ_o. This is formally explained in Section 5.

Flexibility. As we have seen, GSL_{Ref} lets programmers write statically secure programs by first writing the simply-typed version and progressively adding labels. But gradual typing also provides flexibility, so that safe programs that veer from the static type discipline can strategically revert to dynamic checking. GSL_{Ref}'s type-driven approach provides this flexibility. Consider an example adapted from Fennell and Thiemann [2013].[4]

```
1  let infoH : Ref_L Report_H = ...
2  let sendToFacebook : Ref_L Report_L -L->_L Unit_L = ...
3  let sendToManager : Ref_L Report_H -H->_L Unit_L = ...
4  let addPrivileged : Bool_? -H->_? (Ref_L Report_? -?->_L Unit_L) -H->_? Ref_L Report_? -?->_? Unit_L =
5    fun isPrivileged worker report =>
6      if isPrivileged then report := !report + !infoH else ();
7      worker report
8  let sendHi :  Ref_L Report_H -L->_L Unit_L = addPrivileged true sendToManager
9  let sendLow : Ref_L Report_L -L->_L Unit_L = addPrivileged false sendToFacebook
```

The program starts with the creation of a public reference to a private report, `infoH`. It then defines two routines for submitting reports: `sendToFacebook` publishes data publicly, and `sendToManager` publishes data privately. The `addPrivileged` function decides dynamically whether to add high-security information to the sent report, and is used to implement the `sendHi` and `sendLow` functions. This code is secure, but SSL_{Ref}, our static security system, cannot type check `addPrivileged` because of its dynamic choice.

Interestingly, GSL_{Ref} *can* type check this program, thanks to a few well-placed ? labels (line 4), and it dynamically ensures that the program does not leak data. Case in point, the following gradually-typeable function is poised to leak private data:

```
let sendFail : Ref_L Report_L -L->_L Unit_L = addPrivileged true sendToFacebook
```

but if called, GSL_{Ref}'s dynamic security monitor signals an error when `sendToFacebook` dereferences the report, thereby preventing the leak.

Type-based reasoning in GSL_{Ref}. Like prior work, GSL_{Ref} supports smooth migration to static security and flexible programming idioms. Its most significant innovation is that GSL_{Ref} retains the type-based reasoning power of static security typing.

Consider again the example `mix` function of Sec. 1. In GSL_{Ref}, the function body cannot violate the noninterference property implied by its type, *just as in its fully static counterpart language* SSL_{Ref}. In particular, the following definition is rejected statically as expected:

```
let mix : Int_L ->_L Int_H ->_L Int_L = fun pub priv => if pub < priv then 1_L else 2_L
```

In fact, no function body can satisfy this type signature and use its second argument to determine the result. To do so, we must change the type signature, and with it the implied security invariants:

```
let mix : Int_L ->_L Int_? ->_L Int_L = fun pub priv => if pub < priv then 1_L else 2_L
```

[4] Security labels above function arrows track mutation effects (Sec. 3).

The second argument now has statically unknown security. This definition is accepted statically because the function *might* respect the static security invariants of its clients. Consider two such clients, which only differ in the security level of the second argument:

`mix` 1_L 5_H	`mix` 1_L 5_L
Client 1	Client 2

Both type check because the security level of the second argument is *consistent* with the expected, unknown level. Client 2 returns 1_L without incident, because its second argument is public, so applying `mix` does not leak private information. Client 1, however, signals a runtime security error: the function's intended result would implicitly leak information from a private input, but the impending leak is trapped and reported. Treating static security levels as precise requirements rather than upper-bounds, and supporting imprecision, provides the same flexibility as the check-driven approach, as demonstrated in the reporting example above. The key difference is that dynamicity manifests as imprecision in a function's static type, so precise types can preserve their static security interpretation. The interaction between types of different precision is transparently guarded by implicit runtime checks.

If we changed the type signature of `mix` to $\mathsf{Int}_L \to_L \mathsf{Int}_H \to_L \mathsf{Int}_?$, making the return type imprecise, then the definition would type check as well. Nonetheless, GSL_{Ref}'s dynamic enforcement ensures that the returned value could never leak to a public channel, be it a variable or a heap location, because the result is dynamically secured.

The type-driven model lets programmers use type ascriptions to impose static security guarantees on code that is built from imprecisely typed components. Gradual typing automatically introduces dynamic checks to soundly enforce these invariants. Consider a function called `smix` that has a fully static signature but is implemented using the imprecisely-typed `mix` function:

```
let mix : Int_L →_L Int_? →_L Int_L = fun pub priv => if pub < priv then 1_L else 2_L
let smix : Int_L →_L Int_H →_L Int_L = fun pub priv => mix pub priv
```

Type-based reasoning about noninterference dictates that `smix` *cannot* reveal any information about its second argument (regardless of the actual security label of the second argument). For instance, consider the clients:

`smix` 1_L 5_H	`smix` 1_L 5_L
Client 1	Client 2

In GSL_{Ref}, both clients type check, but both fail at runtime! Client 2 fails because `smix`'s type dictates a strong noninterference property, independent of the client's dynamic security levels. To see why, observe that `smix` accepts as second argument any integer value that has a security level no higher than H. When 5_L is substituted in the body of `smix`, its runtime security information is upgraded to H. This new security level in turn strengthens the confidentiality of the value returned by `mix`, which contradicts the static return type of `mix` (L), hence resulting in a runtime error. This behavior preserves local type-based reasoning about the behavior of components, regardless of how they are composed.

To summarize, in GSL_{Ref} different gradual security types denote different security guarantees. Most importantly, the flexibility introduced by *imprecise* security types cannot be abused to violate the type-based noninterference guarantees imposed by *static* security types.

References and implicit flows. In the presence of mutable references, information-flow security faces the classic problem of *implicit flows* through the heap [Denning 1976]. Consider the following program, adapted from Austin and Flanagan [2009]:

```
1  fun x: Bool_H =>
2    let y: Ref_L Bool_L = ref true_L
3    let z: Ref_L Bool_L = ref true_L
4    if x then y := false_L else unit
5    if !y then z := false_L else unit
6    !z
```

This program attempts to downgrade the security of it's input. A static security type system easily rejects it because the first branch of the first conditional (line 4) assigns a low-security reference under a high-security boolean condition. Indeed, in GSL_{Ref} this program is statically rejected as well.

This program is tricky for *dynamic* information flow monitors, however, and has inspired many approaches, *e.g.* [Austin and Flanagan 2009, 2010, 2012; Hedin and Sabelfeld 2012a]. Since gradual security typing includes both static and dynamic security checking, GSL_{Ref} must also address the challenge of dynamically detecting implicit flows. Consider the same program as above but with some imprecise annotations:

```
1  fun x: Bool_H =>
2    let y: Ref? Bool? = ref true?
3    let z: Ref_L Bool_L = ref true_L
4    if x then y := false? else unit
5    if !y then z := false_L else unit
6    !z
```

This gradually-typed variant type checks because the reference bound to `y` now has an unknown security level. But if `x` is bound to true_{H} at runtime, then the program fails with an error at the assignment on line 4, because it cannot replace the contents of a reference in a manner that violates the security context H imposed by the conditional expression `x`. This restriction, and its motivation, is analogous to the "no-sensitive-upgrade" approach of Austin and Flanagan [2009].

Now suppose we make `y`'s type have unknown static security but force its initial contents to have high security, *i.e.*:

```
2  let y: Ref? Bool? = ref true_H
```

Then at runtime the assignment on line 4 succeeds because the assignment on line 2 already refined `y`'s dynamic security to H, which satisfies the security context. Now if `x` is false_{H} then this program fails at the assignment on line 5, because `z`'s security level violates the dynamic security context introduced by branching on the contents of `y`.

To sum up, GSL_{Ref} ensures termination-insensitive noninterference, gradually, even in the presence of references.

3 STATIC SECURITY TYPING WITH REFERENCES

This section introduces SSL_{Ref}, a higher-order static security-typed language with references, which serves as the static extreme of our gradual language. The language is a straightforward adaptation of prior information-flow security typing disciplines [Fennell and Thiemann 2013; Heintze and Riecke 1998; Zdancewic 2002]. The most significant novelties include a syntax-directed type system and a dynamic semantics that tracks security levels but performs no security checks: the type system *alone* guarantees noninterference.

Syntax. Fig. 1 presents the syntax of SSL_{Ref}, at heart a simply-typed higher-order language with references: it includes booleans, functions, unit, mutable references, and type ascription. Each value and type constructor is annotated with a security label $\ell \in \textsc{Label}$ with partial order $\preccurlyeq$, where $\top$ and $\bot$ denote the greatest and least labels respectively. Function abstractions, and their corresponding

$$
\begin{array}{rcll}
S & ::= & \mathsf{Bool}_\ell \mid S \xrightarrow{\ell}_\ell S \mid \mathsf{Ref}_\ell\ S \mid \mathsf{Unit}_\ell & \text{(types)} \\
b & ::= & \mathsf{true} \mid \mathsf{false} & \text{(Booleans)} \\
r & ::= & b \mid (\lambda^\ell x : S.t) \mid \mathsf{unit} \mid \boxed{o} & \text{(raw values)} \\
v & ::= & r_\ell \mid x & \text{(values)} \\
t & ::= & v \mid t\ t \mid t \oplus t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{ref}^S\ t \mid !t \mid t{:=}t \mid t :: S \mid \boxed{\mathsf{prot}_\ell(t)} & \text{(terms)} \\
\oplus & ::= & \wedge \mid \vee & \text{(operations)}
\end{array}
$$

$$(S\mathrm{x})\ \frac{x : S \in \Gamma}{\Gamma; \Sigma; \ell_c \vdash x : S} \qquad (S\mathrm{b})\ \frac{}{\Gamma; \Sigma; \ell_c \vdash b_\ell : \mathsf{Bool}_\ell} \qquad (S\mathrm{u})\ \frac{}{\Gamma; \Sigma; \ell_c \vdash \mathsf{unit}_\ell : \mathsf{Unit}_\ell}$$

$$(S\mathrm{o})\ \frac{o : S \in \Sigma}{\Gamma; \Sigma; \ell_c \vdash o_\ell : \mathsf{Ref}_\ell\ S} \qquad (S\lambda)\ \frac{\Gamma, x : S_1; \Sigma; \ell' \vdash t : S_2}{\Gamma; \Sigma; \ell_c \vdash (\lambda^{\ell'} x : S_1.t)_\ell : S_1 \xrightarrow{\ell'}_\ell S_2}$$

$$(S\mathrm{prot})\ \frac{\Gamma; \Sigma; \ell_c \vee \ell \vdash t : S}{\Gamma; \Sigma; \ell_c \vdash \mathsf{prot}_\ell(t) : S \vee \ell} \qquad (S\oplus)\ \frac{\Gamma; \Sigma; \ell_c \vdash t_1 : \mathsf{Bool}_{\ell_1} \quad \Gamma; \Sigma; \ell_c \vdash t_2 : \mathsf{Bool}_{\ell_2}}{\Gamma; \Sigma; \ell_c \vdash t_1 \oplus t_2 : \mathsf{Bool}_{(\ell_1 \vee \ell_2)}}$$

$$(S\mathrm{app})\ \frac{\Gamma; \Sigma; \ell_c \vdash t_1 : S_{11} \xrightarrow{\ell'}_\ell S_{12} \quad \Gamma; \Sigma; \ell_c \vdash t_2 : S_2 \quad S_2 <: S_{11} \quad \ell_c \vee \ell \preccurlyeq \ell'}{\Gamma; \Sigma; \ell_c \vdash t_1\ t_2 : S_{12} \vee \ell} \qquad (S\mathrm{if})\ \frac{\Gamma; \Sigma; \ell_c \vdash t : \mathsf{Bool}_\ell \quad \Gamma; \Sigma; \ell_c \vee \ell \vdash t_i : S_i}{\Gamma; \Sigma; \ell_c \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : (S_1 \ddot{\vee} S_2) \vee \ell}$$

$$(S\mathrm{asgn})\ \frac{\Gamma; \Sigma; \ell_c \vdash t_1 : \mathsf{Ref}_\ell\ S_1 \quad \Gamma; \Sigma; \ell_c \vdash t_2 : S_2 \quad S_2 <: S_1 \quad \ell_c \vee \ell \preccurlyeq \mathsf{label}(S_1)}{\Gamma; \Sigma; \ell_c \vdash t_1 := t_2 : \mathsf{Unit}_\bot} \qquad (S\mathrm{ref})\ \frac{\Gamma; \Sigma; \ell_c \vdash t : S' \quad S' <: S \quad \ell_c \preccurlyeq \mathsf{label}(S)}{\Gamma; \Sigma; \ell_c \vdash \mathsf{ref}^S\ t : \mathsf{Ref}_\bot\ S}$$

$$(S\mathrm{deref})\ \frac{\Gamma; \Sigma; \ell_c \vdash t : \mathsf{Ref}_\ell\ S}{\Gamma; \Sigma; \ell_c \vdash !t : S \vee \ell} \qquad (S{::})\ \frac{\Gamma; \Sigma; \ell_c \vdash t : S_1 \quad S_1 <: S_2}{\Gamma; \Sigma; \ell_c \vdash t :: S_2 : S_2}$$

Fig. 1. $\mathrm{SSL}_{\mathrm{Ref}}$: Syntax and Static Semantics

types, are annotated with an additional security label called the *latent security effect*: we explain its static semantics below. Two forms arise only at runtime (highlighted in gray): mutable locations o and a *protection term* $\mathsf{prot}_\ell(t)$, which restricts the security effects of its subterm t.

Statics. Fig. 1 also presents the type system of $\mathrm{SSL}_{\mathrm{Ref}}$, which is technically a type-and-effect system [Gifford and Lucassen 1986]. The judgment $\Gamma; \Sigma; \ell_c \vdash t : S$ says that the term t has type S under type environment Γ, store type Σ, and security effect $\ell_c \in \textsc{Label}$. A type environment Γ is a finite map from variables to types. A store type Σ is a finite map from locations to types. The security effect, sometimes called the program counter label [Denning 1976], is a security label that denotes the least security level of those references that a given term may allocate or mutate [Heintze and Riecke 1998]. The security effect prevents high-security computations—*e.g.* the branch of an if expression that is chosen based on a high-security Boolean—from leaking information by assigning to low-security references. An $\mathrm{SSL}_{\mathrm{Ref}}$ source program t is well-typed if $\cdot; \cdot; \bot \vdash t : S$.

- Rule (Sx) and rule (So) type variable and location references as usual. Simple values are also typed as usual, but their types inherit their labels from the values themselves (Sb/Su).
- Rule ($S\lambda$) annotates the type of a function with the latent security effect of its body, as is standard for type-and-effect systems. The greatest (*i.e.* best) security effect can be inferred from the function body, but for simplicity this type system consults an explicit annotation ℓ'.

- Rule (*S*prot) imposes a lower bound ℓ on the security effect of the subterm t. This restriction is captured by *stamping* the label ℓ onto the type [Heintze and Riecke 1998]–*e.g.* $\mathsf{Bool}_\ell \vee \ell' = \mathsf{Bool}_{(\ell \vee \ell')}$, where $\ell \vee \ell'$ represents the least upper-bound, or *join*, of security levels ℓ and ℓ'.
- Rule (*S*⊕) types Boolean operations, yielding a result with the join of the operand security levels.
- Rule (*S*app) is mostly standard, but also enforces security restrictions. First, to prevent mutation-based security leaks, the operator's latent effect ℓ' must *upper-bound* its security level as well as the latent security effect of the entire expression. Both restrictions are captured with a single label comparison in the premise. Second, to prevent value-based security leaks, the security level of the entire expression must upper-bound the level ℓ of the operator—this is done by stamping label ℓ onto the type. Rule (*S*app) also appeals to the *subtyping* relation induced by ordering the security labels. Subtyping is driven by security labels: it is invariant on reference types, covariant on security labels, and contravariant on latent effects [Pottier and Simonet 2003]:

$$\frac{\ell \preccurlyeq \ell'}{\mathsf{Bool}_\ell <: \mathsf{Bool}_{\ell'}} \qquad \frac{\ell \preccurlyeq \ell'}{\mathsf{Unit}_\ell <: \mathsf{Unit}_{\ell'}} \qquad \frac{\ell \preccurlyeq \ell'}{\mathsf{Ref}_\ell\ S <: \mathsf{Ref}_{\ell'}\ S}$$

$$\frac{S_1' <: S_1 \quad S_2 <: S_2' \quad \ell_1 \preccurlyeq \ell_1' \quad \ell_2' \preccurlyeq \ell_2}{S_1 \xrightarrow{\ell_2}_{\ell_1} S_2 <: S_1' \xrightarrow{\ell_2'}_{\ell_1'} S_2'}$$

- Rule (*S*if) incorporates the standard structure for a subtype discipline: the type of the expression involves the *subtyping join* $\dot{\vee}$ of its branches. To protect against *explicit information flows*, the expression type is stamped to incorporate the security level ℓ of the predicate. Additionally, to prevent *effect-based leaks*, each branch is type checked with a security effect that incorporates the security level of the predicate.[5]
- Rules (*S*ref) and (*S*asgn), which perform write effects, are constrained by the security effect of the typing judgment to prevent leaks through the store.
  Rule (*S*ref) honors the effect discipline by requiring the current security effect to lower-bound the security level of the stored value. The resulting reference has least security $\bot$ because it is newly minted and cannot leak information: the type of the stored content is known and its security level prevents further prying.
  Rule (*S*asgn) ensures that the security level of the location and current security effect lower-bound the assigned value. The result of assignment has $\bot$ security because unit cannot leak information. Rule (*S*deref) stamps the security level of the reference onto the resulting type.
- Finally, Rule (*S*::) is typical for ascription, requiring the ascribed type to be a supertype of the subterm's type.

Dynamics. With fully static security typing, programs execute on a standard runtime with no additional security-enforcing machinery. Type *safety*—well-typed terms do not get stuck—is guaranteed by the underlying run-of-the-mill simple type discipline. However, to establish the *soundness* of security typing—high-security computations have no effect on low-security observations—one must characterize computations and their resulting values with respect to their security levels. To this end, the $\mathrm{SSL}_{\mathsf{Ref}}$ dynamic semantics explicitly *tracks* security labels as programs evaluate, but never *checks* them. The noninterference proof demonstrates that no such

[5]Note that $\mathrm{SSL}_{\mathsf{Ref}}$ does not have an explicit effect ascription form $t :: \ell_c$ [Bañados Schwerter et al. 2014], but this can be encoded using the expression $(\lambda^{\ell_c} x : \mathsf{Unit}_\bot . t)_\bot\ \mathsf{unit}_\bot$.

$$\boxed{t \mid \mu \xrightarrow{\ell_c} t \mid \mu} \textbf{ Notion of Reduction}$$

$$b_1{}_{\ell_1} \oplus b_2{}_{\ell_2} \mid \mu \xrightarrow{\ell_c} (b_1 \,[\![\oplus]\!]\, b_2)_{(\ell_1 \curlyvee \ell_2)} \mid \mu \qquad (\lambda^{\ell'} x : S.t)_\ell \; v \mid \mu \xrightarrow{\ell_c} \mathsf{prot}_\ell([v/x]t) \mid \mu$$

$$\mathsf{if\ true}_\ell \ \mathsf{then}\ t_1 \ \mathsf{else}\ t_2 \mid \mu \xrightarrow{\ell_c} \mathsf{prot}_\ell(t_1) \mid \mu \qquad \mathsf{if\ false}_\ell \ \mathsf{then}\ t_1 \ \mathsf{else}\ t_2 \mid \mu \xrightarrow{\ell_c} \mathsf{prot}_\ell(t_2) \mid \mu$$

$$\mathsf{prot}_\ell(v) \mid \mu \xrightarrow{\ell_c} v \curlyvee \ell \mid \mu \qquad \mathsf{ref}^S \; v \mid \mu \xrightarrow{\ell_c} o_\bot \mid \mu[o \mapsto v \curlyvee \ell_c] \text{ where } o \notin dom(\mu)$$

$$!o_\ell \mid \mu \xrightarrow{\ell_c} v \curlyvee \ell \mid \mu \text{ where } \mu(o) = v \qquad o_\ell := v \mid \mu \xrightarrow{\ell_c} \mathsf{unit}_\bot \mid \mu[o \mapsto v \curlyvee \ell_c \curlyvee \ell]$$

$$v :: S \mid \mu \xrightarrow{\ell_c} v \curlyvee \mathit{label}(S) \mid \mu$$

$$\boxed{t \mid \mu \overset{\ell_c}{\longmapsto} t \mid \mu} \textbf{ Reduction}$$

$$(\mathrm{R}{\rightarrow}) \frac{t_1 \mid \mu_1 \xrightarrow{\ell_c} t_2 \mid \mu_2}{t_1 \mid \mu_1 \overset{\ell_c}{\longmapsto} t_2 \mid \mu_2} \qquad (\mathrm{R}f) \frac{t_1 \mid \mu_1 \overset{\ell_c}{\longmapsto} t_2 \mid \mu_2}{f[t_1] \mid \mu_1 \overset{\ell_c}{\longmapsto} f[t_2] \mid \mu_2}$$

$$(\mathrm{Rprot}) \frac{t_1 \mid \mu_1 \overset{\ell_c \curlyvee \ell}{\longmapsto} t_2 \mid \mu_2}{\mathsf{prot}_\ell(t_1) \mid \mu_1 \overset{\ell_c}{\longmapsto} \mathsf{prot}_\ell(t_2) \mid \mu_2}$$

Fig. 2. $\mathsf{SSL}_{\mathsf{Ref}}$: Label Tracking Dynamic Semantics

checks are required: static typing suffices. Tracking labels provides weak security guarantees that are exploited in the proof of the stronger noninterference result.

Fig. 2 presents the rules of the label-tracking dynamic semantics. The judgment $t_1 \mid \mu_1 \overset{\ell_c}{\longmapsto} t_2 \mid \mu_2$ says that a term t_1 and store μ_1 step to t_2 and μ_2 respectively, in security effect ℓ_c. Reduction of terms is specified using *term frames* f:

$$f \quad ::= \quad \square \oplus t \mid v \oplus \square \mid \square \; t \mid v \; \square \mid \square :: S \mid \mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t \mid !\square \mid \square := t \mid v := \square \mid \mathsf{ref}^S \; \square$$

The core semantics is typical, so we focus on tracking security. The runtime security effect ℓ_c, which reflects its static counterpart, affects the security level of reads from and writes to the store, as well as the security level of values returned from high-security contexts to low-security ones.

Protection terms $\mathsf{prot}_\ell(t)$ control the current program counter label. Apart from prot, all expressions propagate the current program counter to subterms. Rule (Rprot) upgrades ℓ_c for the dynamic extent of t. The resulting value is stamped with the protected label ℓ, in case the contents leak information to a context that lacks the confidentiality of ℓ. Values are stamped much like types: $r_\ell \curlyvee \ell' = r_{(\ell \curlyvee \ell')}$. Protection terms do not exist in source programs: they are introduced by control operations, *i.e.* function calls and conditionals. The intuition is that calling a function or destructing a Boolean of security level ℓ may leak information about the identity of the function or Boolean respectively. As such, the context of the resulting computation should communicate (via mutation) only with reference cells that have high-enough security, and the result of the computation is classified as well.[6] Function calls ignore the operator's latent effect ℓ', which promises the *type system*

[6] Zdancewic [2002] observes that *e.g.* if x then e_{L} else e_{L} leaks no information about Boolean $x : \mathsf{Bool}_{\mathsf{H}}$ so could be deemed low-security, but security type systems must be conservative for the sake of tractability.

that the ensuing computation will not violate the stated confidentiality. However the operator's security label determines the confidentiality of the ensuing computation.

When stored, a value inherits confidentiality from both the current security effect and the location itself. This behavior tracks both the confidentiality of the location and the induced security effect.

Properties. SSL_{Ref} is type safe: we establish this result via a standard progress and preservation argument [Toro et al. 2018]. Since the runtime semantics includes no security checks, progress mirrors the corresponding argument for the underlying simple type discipline. To prove preservation, we must show that after each reduction step the resulting term still has the same security according to the typing rules of Fig. 1, modulo subtyping.

Proposition 3.1 (Type Safety). *If* $\cdot; \Sigma; \ell_c \vdash t : S$ *then either*

- t *is a value* v
- *for any store* μ *such that* $\Sigma \vdash \mu$ *and any* $\ell'_c \preccurlyeq \ell_c$*, we have* $t \mid \mu \overset{\ell'_c}{\longmapsto} t' \mid \mu'$ *and* $\cdot; \Sigma'; \ell_c \vdash t' : S'$ *for some* $S' <: S$*, and some* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash \mu'$.

The store typing judgment $\Sigma \vdash \mu$ holds if and only if $dom(\mu) = dom(\Sigma)$ and $\cdot; \Sigma; \ell_c \vdash \mu(o) : \Sigma(o)$ for all $o \in dom(\mu)$, $\ell_c \in$ Label.

The most important property of a security-typed language like SSL_{Ref} is the *soundness* of security typing, *i.e.* that well-typed programs have no forbidden information flows. We formally state and prove noninterference using step-indexed logical relations (see the companion technical report [Toro et al. 2018]).We do not include the definitions of the logical relations and noninterference statement here because proving that SSL_{Ref} is secure is not the main focus of this work, and the full treatment of noninterference for the gradual language (Sec. 5) subsumes them.

4 GSL_{Ref}: TYPE-DRIVEN GRADUAL SECURITY TYPING

This section presents the static and dynamic semantics of GSL_{Ref}, and addresses its type safety and gradual guarantees. We show that GSL_{Ref} enforces noninterference in Sec. 5.

The reader might (understandably!) wonder how some of the definitions presented in this section were conceived. This section largely appeals to intuition to justify these definitions, but in practice they were obtained by following the Abstracting Gradual Typing methodology [Garcia et al. 2016], which exploits principles of abstract interpretation [Cousot and Cousot 1977] to systematically derive a gradual language from a static one. In fact, this work can be seen as a particularly challenging case study for AGT—which has led us to identify the limits of the AGT approach when applied to disciplines where type *safety* (*i.e.* "well-typed terms do not get stuck") does not imply type *soundness* (*i.e.* "well-typed terms do not leak"). The gradual language obtained by a straightforward application of AGT is type safe, but does not ensure noninterference because of subtle interactions between security typing imprecision and heap-based flows. We discuss the key elements, pitfalls, and discoveries of this systematic derivation process in Sec. 6.

To aid the reader, Fig. 3 indicates where important terms, operations and relations are presented, along with their notation.

4.1 Static semantics

Fig. 4 presents the syntax and static semantics of GSL_{Ref}.[7] A gradual security label $g \in$ GLabel is either a static label ℓ or the unknown label ?, which represents any label whatsoever. Each value and gradual type constructor is now annotated with a gradual security label.

[7] In GSL_{Ref}, the o and $\text{prot}_g(t)$ forms and typing rules merely serve to induce corresponding $\text{GSL}^{\varepsilon}_{\text{Ref}}$ forms (Sec 4.2).

Term	Notation	Ref
Gradual Type	U	F 4
Gradual label	g	F 4
Term	t	F 4
Interval	$\imath$	P 17
Evidence for labels	ε	P 17
Evidence for types	ε	P 17
Evidence term	t	P 18
Frames	f, h	P 23

Operation/Relation	Notation	Ref
Consistent label ordering (g)	$\widetilde{\preccurlyeq}$	P 14
Consistent label join (g)	$\widetilde{\curlyvee}$	P 15
Consistent label meet (g)	$\widetilde{\curlywedge}$	P 15
Gradual meet (g)	$\sqcap$	P 16
Evidence join (ε on labels)	$\widetilde{\curlyvee}$	F 15
Evidence meet (ε on labels)	$\widetilde{\curlywedge}$	F 15
Gradual meet ($\imath$)	$\sqcap$	P 21
Gradual meet (ε on labels)	$\sqcap$	F 15
Lower-bound-comparison (ε)	$\lfloor \leq \rfloor$	P 23
Initial evidence (g)	$\mathcal{g}$	F 18
Reflexive initial evidence (g)	$\mathcal{g}^{\circlearrowleft}$	F 5
Transitivity (ε on labels)	$\circ^{\preccurlyeq}$	P 21

Operation/Relation	Notation	Ref
Consistent subtyping (U)	$\lesssim$	P 14
Consistent join (U)	$\widetilde{\dot{\curlyvee}}$	P 16
Consistent meet (U)	$\widetilde{\dot{\curlywedge}}$	F 13
Gradual meet (U)	$\sqcap$	P 16
Evidence join (ε on types)	$\widetilde{\curlyvee}$	F 16
Evidence meet (ε on types)	$\widetilde{\curlywedge}$	F 16
Gradual meet (ε on types)	$\sqcap$	F 16
Initial evidence (U)	$\mathcal{g}$	F 19
Reflexive initial evidence (U)	$\mathcal{g}^{\circlearrowleft}$	F 5
Transitivity (ε on types)	$\circ^{<:}$	F 16
Evidence inversion label (ε)	*ilbl*	F 17
Evidence inversion ref (ε)	*iref*	F 17
Evidence inversion dom (ε)	*idom*	F 17
Evidence inversion cod (ε)	*icod*	F 17
Evidence inversion latent (ε)	*ilat*	F 17
Label Stamping ($S \curlyvee \ell$)	$\curlyvee$	P 48
Subtyping join (S)	$\dot{\curlyvee}$	F 11
Subtyping meet (S)	$\dot{\curlywedge}$	F 11

Fig. 3. Index of terms, operations and relations used in this article, along with their notation, and reference to corresponding Figure (F) or Page (P).

The typing judgment $\Gamma; \Sigma; g_c \vdash t : U$ says that the term t has gradual type U under type environment Γ, store environment Σ, and *gradual* security effect g_c. The typing rules are analogous to the static typing rules presented in Fig. 1 except that security labels, types, type functions and predicates are all replaced by their gradual counterparts. For instance, static label ordering $\preccurlyeq$ is replaced with *consistent label ordering* $\widetilde{\preccurlyeq}$:

$$\frac{}{? \mathrel{\widetilde{\preccurlyeq}} g} \qquad \frac{}{g \mathrel{\widetilde{\preccurlyeq}} ?} \qquad \frac{\ell_1 \preccurlyeq \ell_2}{\ell_1 \mathrel{\widetilde{\preccurlyeq}} \ell_2}$$

Intuitively, if consistent label ordering between two gradual labels holds, then it means that the static relation holds for some static labels represented by the gradual labels. It is always plausible in the presence of ?, since the unknown label represents any label. Similarly, subtyping is lifted to *consistent subtyping* $\lesssim$, whose definition is analogous to static subtyping, but using consistent label ordering:

$$\frac{g \mathrel{\widetilde{\preccurlyeq}} g'}{\mathsf{Bool}_g \lesssim \mathsf{Bool}_{g'}} \qquad \frac{g \mathrel{\widetilde{\preccurlyeq}} g'}{\mathsf{Unit}_g \lesssim \mathsf{Unit}_{g'}} \qquad \frac{g \mathrel{\widetilde{\preccurlyeq}} g' \quad U_1 \lesssim U_2 \quad U_2 \lesssim U_1}{\mathsf{Ref}_g\ U_1 \lesssim \mathsf{Ref}_{g'}\ U_2}$$

$$\frac{U_1' \lesssim U_1 \quad U_2 \lesssim U_2' \quad g_1 \mathrel{\widetilde{\preccurlyeq}} g_1' \quad g_2' \mathrel{\widetilde{\preccurlyeq}} g_2}{U_1 \xrightarrow{g_2}_{g_1} U_2 \lesssim U_1' \xrightarrow{g_2'}_{g_1'} U_2'}$$

$$
\begin{array}{rcll}
U & ::= & \mathsf{Bool}_g \mid U \xrightarrow{g}_g U \mid \mathsf{Ref}_g\ U \mid \mathsf{Unit}_g & \text{(gradual types)} \\
g & ::= & \ell \mid ? & \text{(gradual labels)} \\
b & ::= & \mathsf{true} \mid \mathsf{false} & \text{(Booleans)} \\
r & ::= & b \mid (\lambda^g x : U.t) \mid \mathsf{unit} \mid o & \text{(base values)} \\
v & ::= & r_g \mid x & \text{(values)} \\
t & ::= & v \mid t\ t \mid t \oplus t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{ref}^U\ t \mid !t \mid t{:=}t \mid \mathsf{prot}_g(t) \mid t :: U & \text{(terms)} \\
\oplus & ::= & \wedge \mid \vee & \text{(operations)}
\end{array}
$$

$$(U\mathrm{x})\ \frac{x : U \in \Gamma}{\Gamma; \Sigma; g_c \vdash x : U} \qquad (U\mathrm{b})\ \frac{}{\Gamma; \Sigma; g_c \vdash b_g : \mathsf{Bool}_g} \qquad (U\mathrm{u})\ \frac{}{\Gamma; \Sigma; g_c \vdash \mathsf{unit}_g : \mathsf{Unit}_g}$$

$$(U o)\ \frac{o : U \in \Sigma}{\Gamma; \Sigma; g_c \vdash o_g : \mathsf{Ref}_g\ U} \qquad (U\lambda)\ \frac{\Gamma, x : U_1; \Sigma; g' \vdash t : U_2}{\Gamma; \Sigma; g_c \vdash (\lambda^{g'} x : U_1.t)_g : U_1 \xrightarrow{g'}_g U_2}$$

$$(U\mathrm{prot})\ \frac{\Gamma; \Sigma; g_c \mathbin{\widetilde{\vee}} g \vdash t : U}{\Gamma; \Sigma; g_c \vdash \mathsf{prot}_g(t) : U \mathbin{\widetilde{\vee}} g} \qquad (U\oplus)\ \frac{\Gamma; \Sigma; g_c \vdash t_1 : \mathsf{Bool}_{g_1} \quad \Gamma; \Sigma; g_c \vdash t_2 : \mathsf{Bool}_{g_2}}{\Gamma; \Sigma; g_c \vdash t_1 \oplus t_2 : \mathsf{Bool}_{(g_1 \mathbin{\widetilde{\vee}} g_2)}}$$

$$(U\mathrm{app})\ \frac{\Gamma; \Sigma; g_c \vdash t_1 : U_{11} \xrightarrow{g'}_g U_{12} \quad \Gamma; \Sigma; g_c \vdash t_2 : U_2 \quad U_2 \lesssim U_{11} \quad \widetilde{g \vee g_c \preccurlyeq g'}}{\Gamma; \Sigma; g_c \vdash t_1\ t_2 : U_{12} \mathbin{\widetilde{\vee}} g}$$

$$(U\mathrm{if})\ \frac{\Gamma; \Sigma; g_c \vdash t : \mathsf{Bool}_g \quad \Gamma; \Sigma; g_c \mathbin{\widetilde{\vee}} g \vdash t_1 : U_1 \quad \Gamma; \Sigma; g_c \mathbin{\widetilde{\vee}} g \vdash t_2 : U_2}{\Gamma; \Sigma; g_c \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : (U_1 \mathbin{\widetilde{\vee}} U_2) \mathbin{\widetilde{\vee}} g}$$

$$(U\mathrm{asgn})\ \frac{\Gamma; \Sigma; g_c \vdash t_1 : \mathsf{Ref}_g\ U_1 \quad \Gamma; \Sigma; g_c \vdash t_2 : U_2 \quad U_2 \lesssim U_1 \quad \widetilde{g \vee g_c \preccurlyeq \mathit{label}(U_1)}}{\Gamma; \Sigma; g_c \vdash t_1{:=}t_2 : \mathsf{Unit}_\bot}$$

$$(U\mathrm{ref})\ \frac{\Gamma; \Sigma; g_c \vdash t : U' \quad U' \lesssim U \quad g_c \mathbin{\widetilde{\preccurlyeq}} \mathsf{label}(U)}{\Gamma; \Sigma; g_c \vdash \mathsf{ref}^U\ t : \mathsf{Ref}_\bot\ U}$$

$$(U\mathrm{deref})\ \frac{\Gamma; \Sigma; g_c \vdash t : \mathsf{Ref}_g\ U}{\Gamma; \Sigma; g_c \vdash !t : U \mathbin{\widetilde{\vee}} g} \qquad (U::)\ \frac{\Gamma; \Sigma; g_c \vdash t : U_1 \quad U_1 \lesssim U_2}{\Gamma; \Sigma; g_c \vdash t :: U_2 : U_2}$$

Fig. 4. $\mathrm{GSL}_{\mathsf{Ref}}$: Static Semantics

The label join and meet operators are replaced with *consistent join* and *consistent meet* respectively:

$$\top \mathbin{\widetilde{\vee}} ? = ? \mathbin{\widetilde{\vee}} \top = \top \qquad g \mathbin{\widetilde{\vee}} ? = ? \mathbin{\widetilde{\vee}} g = ?\ \text{ if } g \neq \top \qquad \ell_1 \mathbin{\widetilde{\vee}} \ell_2 = \ell_1 \vee \ell_2$$

$$\bot \mathbin{\widetilde{\wedge}} ? = ? \mathbin{\widetilde{\wedge}} \bot = \bot \qquad g \mathbin{\widetilde{\wedge}} ? = ? \mathbin{\widetilde{\wedge}} g = ?\ \text{ if } g \neq \bot \qquad \ell_1 \mathbin{\widetilde{\wedge}} \ell_2 = \ell_1 \wedge \ell_2$$

These operators recover precise label information when the unknown label interacts with the relevant boundary element ($\top$ for $\widetilde{\vee}$, and $\bot$ for $\widetilde{\wedge}$), otherwise the result is always unknown. Intuitively, this is because *any* label ℓ joined (resp. met) with $\top$ (resp. $\bot$), yields $\top$ (resp. $\bot$), so imprecise arguments do not perturb the results. But when the relevant boundary is not involved, then varying ℓ can vary the results, a possibility that is captured by using the unknown label as result.

The join operators for subtyping and label ordering are replaced with consistent join $\widetilde{\dot{\vee}}$ and consistent label join $\widetilde{\vee}$ respectively:

$$\mathsf{Bool}_g \mathbin{\widetilde{\dot{\vee}}} \mathsf{Bool}_{g'} = \mathsf{Bool}_{(g \mathbin{\widetilde{\vee}} g')} \qquad \mathsf{Unit}_g \mathbin{\widetilde{\dot{\vee}}} \mathsf{Unit}_{g'} = \mathsf{Unit}_{(g \mathbin{\widetilde{\vee}} g')} \qquad \mathsf{Ref}_g\ U \mathbin{\dot{\vee}} \mathsf{Ref}_{g'}\ U' = \mathsf{Ref}_{(g \mathbin{\widetilde{\vee}} g')}\ U \sqcap U'$$

$$(U_{11} \xrightarrow{g_1'}_{g_1} U_{12}) \mathbin{\widetilde{\dot{\vee}}} (U_{21} \xrightarrow{g_2'}_{g_2} U_{22}) = (U_{11} \mathbin{\widetilde{\wedge}} U_{21}) \xrightarrow[(g_1 \mathbin{\widetilde{\vee}} g_2)]{g_1' \mathbin{\widetilde{\wedge}} g_2'} (U_{12} \mathbin{\widetilde{\dot{\vee}}} U_{22})$$

$$U \mathbin{\widetilde{\dot{\vee}}} U \text{ undefined otherwise}$$

The consistent subtyping meet operator is defined dually (definition in Appendix A.3).

Consistent subtyping join appeals to a gradual meet operator $\sqcap$ on the referent types. This gradual meet arises because static subtyping is invariant for the contents of references, so static subtype join is only defined for references with equal referent types. The gradual meet operator can be understood as the gradual counterpart of a static type equality partial function *equate* (*i.e. equate*$(S, S) = S$, undefined otherwise) [Garcia et al. 2016]. Intuitively, if the $\sqcap$ of two gradual entities is defined, then it means that they are possibly equal. For instance, $\mathsf{H} \sqcap \mathsf{L}$ is undefined, but $\mathsf{H} \sqcap ? = \mathsf{H}$. Formally:

$$\begin{aligned}
g \sqcap g &= g \\
g \sqcap ? = ? \sqcap g &= g \\
\mathsf{Bool}_g \sqcap \mathsf{Bool}_{g'} &= \mathsf{Bool}_{g \sqcap g'} \\
\mathsf{Unit}_g \sqcap \mathsf{Unit}_{g'} &= \mathsf{Unit}_{g \sqcap g'} \\
\mathsf{Ref}_g\ U \sqcap \mathsf{Ref}_{g'}\ U' &= \mathsf{Ref}_{g \sqcap g'}\ U \sqcap U' \\
U_1 \xrightarrow{g_2}_{g_1} U_2 \sqcap U_1' \xrightarrow{g_2'}_{g_1'} U_2' &= (U_1 \sqcap U_1') \xrightarrow[g_1 \sqcap g_1']{g_2 \sqcap g_2'} (U_2 \sqcap U_2')
\end{aligned}$$

Finally, The $\mathrm{SSL}_{\mathsf{Ref}}$ rules (*S*app) and (*S*asgn) from Fig. 1 have compound premises that combine both label join and label ordering, *e.g.* $\ell_c \vee \ell \preccurlyeq \ell'$. One subtlety we discovered while applying the AGT methodology is that these premises lose precision when lifted compositionally: simply replacing join with consistent join and label ordering with consistent label ordering yields different results than when lifted in aggregate; we discuss this further in Sec. 6. Therefore rules (*U*app) and (*U*asgn) use the *consistent bounding* predicate,which is defined algorithmically as: $\widetilde{g_1 \vee g_2 \preccurlyeq g_3} \iff g_1 \mathbin{\widetilde{\preccurlyeq}} g_3 \wedge g_2 \mathbin{\widetilde{\preccurlyeq}} g_3$. Technically, we could have used this definition to split each premise, but treating the predicate atomically matters when we consider the dynamic semantics.

4.2 Dynamic semantics

To present the dynamic semantics of $\mathrm{GSL}_{\mathsf{Ref}}$, we first define a reduction relation for an internal language $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ that directly mirrors $\mathrm{GSL}_{\mathsf{Ref}}$, except that all terms are augmented with some *evidence information* that justifies why the term is well-typed according to the gradual type system. During reduction steps, units of evidence are combined to form new evidence that supports type preservation between a term and its contractum. If the combination succeeds, reduction goes on; if the combination fails, a runtime error is raised. We first explain what evidence is, then how $\mathrm{GSL}_{\mathsf{Ref}}$ programs are elaborated with evidence information into $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$, and finally how evidence is combined, yielding the $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ reduction rules.

Evidence for consistent judgments. Evidence captures *why* a consistent judgment holds. To explain this concept, we begin with consistent judgments about security labels, then consider the more complex consistent judgments about types.

We use the metavariable ε to range over evidence, and write $\varepsilon \vdash g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$ to say that evidence ε supports the plausibility that $g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$ holds.

For instance, consider the consistent ordering judgment $? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$. Even though the unknown label generally denotes any security label, consistent ordering insists that this ? can only denote labels that are bounded from above by L. Furthermore, this consistent ordering judgment yields no additional information about the right-hand side, which is already precise. We capture this learned information by representing evidence as a *pair of static label intervals*, noted $\langle \iota_1, \iota_2 \rangle$, where $\iota = [\ell, \ell']$. If $\langle \iota_1, \iota_2 \rangle \vdash g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$ then ι_1 and ι_2 represent inferred range restrictions for g_1 and g_2 respectively. Therefore,

$$\langle [\bot, \mathsf{L}], [\mathsf{L}, \mathsf{L}] \rangle \vdash ? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$$

By analogous reasoning, the consistent judgment $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ is initially justified by the evidence $\langle [\mathsf{H}, \mathsf{H}], [\mathsf{H}, \top] \rangle$, gaining precision about the right-hand side. Interval precision is defined as containment over intervals, i.e. $[\ell_1, \ell_2] \sqsubseteq [\ell_1', \ell_2']$ if and only if $\ell_1' \preccurlyeq \ell_1$ and $\ell_2 \preccurlyeq \ell_2'$. Precision between interval pairs $\langle \iota_1, \iota_2 \rangle \sqsubseteq \langle \iota_1', \iota_2' \rangle$ is defined pointwise.

We represent evidence as pairs of intervals, rather than pairs of labels, essentially because pairs of labels are not precise enough to support gradual security. The formal rationale is involved, so we defer it to Sec. 6. For some intuition, though, consider the program $\mathsf{true}_? :: \mathsf{Bool}_\mathsf{H} :: \mathsf{Bool}_? :: \mathsf{Bool}_\mathsf{L}$. Evaluating it ultimately involves combining evidence for three consecutive judgments:[8] $\varepsilon_1 \vdash ? \mathrel{\widetilde{\preccurlyeq}} \mathsf{H}$, $\varepsilon_2 \vdash \mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$, and $\varepsilon_3 \vdash ? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$. The program should fail at runtime because an H security value should not be coerceable to L, so these three evidences should not compose. Unfortunately, pairs of labels are not precise enough to ensure this: they forget the intermediate step through H. In contrast, pairs of label intervals retain enough precision to warrant the expected runtime failure.

To justify consistent judgments about types like consistent subtyping, we lift label evidence to *type evidence* ε by naturally lifting intervals to types: type constructors are now marked with label intervals instead of labels. For instance:

$$\langle \mathsf{Bool}_{[\bot,\mathsf{L}]}, \mathsf{Bool}_{[\mathsf{L},\mathsf{L}]} \rangle \vdash \mathsf{Bool}_? \lesssim \mathsf{Bool}_\mathsf{L}$$

The syntax of evidence is as follows:

$$\begin{array}{lcll} E \in \textsc{GeType}, & \iota \in \textsc{Interval}, & \varepsilon \in \textsc{Evidence} & \\ \iota & ::= & \langle \ell, \ell \rangle & \text{(intervals)} \\ E & ::= & \mathsf{Bool}_\iota \mid E \xrightarrow{\iota}_\iota E \mid \mathsf{Ref}_\iota\ E \mid \mathsf{Unit}_\iota & \text{(type evidences)} \\ \varepsilon & ::= & \langle E, E \rangle \mid \langle \iota, \iota \rangle & \text{(evidences)} \end{array}$$

Note that we use the same metavariable ε to represent both label evidence and type evidence, since which kind of evidence is meant is always clear from the context.

Terms with evidence. Each well-typed term of $\mathrm{GSL}_{\mathsf{Ref}}$ is recursively elaborated into a $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ term by decorating it with evidence for the consistent judgments used to establish its well-typedness.

[8] in a way that we make precise below.

The syntax of $\text{GSL}^{\varepsilon}_{\text{Ref}}$ terms follows:

$$
\begin{array}{rcll}
t & ::= & v \mid \varepsilon t\ @_{\varepsilon}\ \varepsilon t \mid \varepsilon t \oplus \varepsilon t \mid \mathsf{if}\ \varepsilon t\ \mathsf{then}\ \varepsilon t\ \mathsf{else}\ \varepsilon t \mid & \\
 & & \mathsf{ref}^{U}_{\varepsilon}\ \varepsilon t \mid\ !\varepsilon t \mid \varepsilon t :=_{\varepsilon} \varepsilon t \mid \mathsf{prot}_{\varepsilon g} \varepsilon g(\varepsilon t) \mid \varepsilon t & \text{(terms)} \\
r & ::= & b \mid (\lambda^{g} x : U.t) \mid \mathsf{unit} \mid o & \text{(base values)} \\
u & ::= & r_g \mid x & \text{(raw values)} \\
v & ::= & u \mid \varepsilon u & \text{(values)}
\end{array}
$$

During reduction, the actual type of a subterm may evolve to a consistent subtype of the statically-determined type. For this reason, each term is augmented with evidence for their immediate sub-redexes (*i.e.* all subterms that have to be reduced to a value for computation to proceed), justifying why the subterms are consistent subtypes of the types demanded statically by the outer term constructor. For instance, in the term $\varepsilon_1 t_1 \oplus \varepsilon_2 t_2$, ε_1 justifies t_1 being a consistent subtype of Bool_{g_1}, the type deduced during type checking. In particular, t_1 could be such a consistent subtype because it is a value that was ascribed type Bool_{g_1} using an explicit ascription. In fact, GSL_{Ref} ascriptions are represented simply as evidence-augmented terms εt in $\text{GSL}^{\varepsilon}_{\text{Ref}}$: the evidence ε holds all the computationally-relevant information about consistent subtyping. For instance, the GSL_{Ref} term $(10_{\mathsf{L}} :: \mathsf{Int}_?) :: \mathsf{Int}_{\mathsf{H}}$ is translated to $\varepsilon_2(\varepsilon_1 10_{\mathsf{L}})$, where $\varepsilon_1 \vdash \mathsf{Int}_{\mathsf{L}} \lesssim \mathsf{Int}_?$ and $\varepsilon_2 \vdash \mathsf{Int}_? \lesssim \mathsf{Int}_{\mathsf{H}}$.

Note that in addition, some terms carry extra evidences that are needed during reduction to justify type preservation. A conditional $\mathsf{if}\ \varepsilon_1 t_1\ \mathsf{then}\ \varepsilon_2 t_2\ \mathsf{else}\ \varepsilon_3 t_3$ carries evidences ε_2 and ε_3 that justify that the type of each branch t_2 and t_3 is a consistent subtype of the type of the conditional expression. For instance, if U_2 and U_3 are the types of t_2 and t_3 respectively, then $\varepsilon_2 \vdash \widetilde{U_2 <: U_2 \dot{\vee} U_3}$, where $\widetilde{U_1 <: U_2 \dot{\vee} U_3}$ is the consistent lifting of the ternary static judgment $T_1 <: T_2 \dot{\vee} T_3$. Similarly, a protection term $\mathsf{prot}_{\varepsilon_1 g_1} \varepsilon_2 g_2 (\varepsilon_3 t)$ carries a security effect g_2 (and its evidence ε_2), which represents the security effect of the subterm t; specifically, g_2 is the join of g_1 and the current security effect.

Values are either raw values u or evidence-augmented raw values εu. The latter correspond to ascribed values $v :: U$ in GSL_{Ref}: the evidence ε confirms that the u's type is a consistent subtype of the ascribed type U.

Several terms—applications, references, assignment, and protection—have evidence in addition to that of their subterms. This extra evidence supports the consistent label ordering judgments of their corresponding typing rule, which relate to the current latent effect label. For instance, in the term $\mathsf{ref}^{U}_{\varepsilon'}\ \varepsilon t$, the evidence ε' supports the consistent label ordering judgment $g_c \mathrel{\widetilde{\preccurlyeq}} \mathsf{label}(U)$. For uniformity, we overload the metavariable ε to denote both label and type evidence, since the difference is always clear from the context. Evidence attached to subterms is type evidence, and evidence attached to the security effect or to an expression symbol (@, ref, := , or prot) is label evidence.

Introducing evidence. Fig. 5 presents rules for elaborating GSL_{Ref} source terms to evidence-augmented $\text{GSL}^{\varepsilon}_{\text{Ref}}$ terms. This elaboration is akin to a cast insertion translation [Siek and Taha 2006], but simpler because it inserts evidence uniformly [Garcia et al. 2016]. Basically, each consistent label and type judgment in Fig. 4 is replaced by an evidence-computing partial function called an *initial evidence operator* ($\mathcal{J}$). An initial evidence operator computes the most precise evidence that can be deduced from a given judgment. For instance, given a consistent label ordering judgment $g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$, the initial evidence for it is computed as follows:

$$\mathcal{J}[\![g_1 \mathrel{\widetilde{\preccurlyeq}} g_2]\!] = \mathit{intr}(\mathit{bounds}(g_1), \mathit{bounds}(g_2))$$

$$\boxed{\Gamma; \Sigma; g_c \vdash t \rightsquigarrow t' : U}$$

$$(T\mathrm{x})\ \frac{\Gamma(x) = U}{\Gamma; \Sigma; g_c \vdash x \rightsquigarrow x : U} \qquad (T\mathrm{b})\ \frac{}{\Gamma; \Sigma; g_c \vdash b_g \rightsquigarrow b_g : \mathsf{Bool}_g}$$

$$(T\mathrm{u})\ \frac{}{\Gamma; \Sigma; g_c \vdash \mathsf{unit}_g \rightsquigarrow \mathsf{unit}_g : \mathsf{Unit}_g} \qquad (T\lambda)\ \frac{\Gamma; \Sigma; g' \vdash t \rightsquigarrow t' : U_2}{\Gamma; \Sigma; g_c \vdash (\lambda^{g'} x : U_1.t)_g \rightsquigarrow (\lambda^{g'} x : U_1.t')_g : U_1 \xrightarrow{g'}_g U_2}$$

$$(T\oplus)\ \frac{\Gamma; \Sigma; g_c \vdash t_1 \rightsquigarrow t_1' : \mathsf{Bool}_{g_1} \quad \Gamma; \Sigma; g_c \vdash t_2 \rightsquigarrow t_2' : \mathsf{Bool}_{g_2} \quad \varepsilon_1 = \mathcal{G}^{\cup}[\![\mathsf{Bool}_{g_1}]\!] \quad \varepsilon_2 = \mathcal{G}^{\cup}[\![\mathsf{Bool}_{g_2}]\!]}{\Gamma; \Sigma; g_c \vdash t_1 \oplus t_2 \rightsquigarrow \varepsilon_1 t_1' \oplus \varepsilon_2 t_2' : \mathsf{Bool}_{g_1 \mathbin{\widetilde{\vee}} g_2}}$$

$$(T\mathrm{app})\ \frac{\Gamma; \Sigma; g_c \vdash t_1 \rightsquigarrow t_1' : U_{11} \xrightarrow{g'}_g U_{12} \quad \Gamma; \Sigma; g_c \vdash t_2 \rightsquigarrow t_2' : U_2 \quad \varepsilon_1 = \mathcal{G}^{\cup}[\![U_{11} \xrightarrow{g'}_g U_{12}]\!] \quad \varepsilon_2 = \mathcal{G}[\![U_2 \lesssim U_{11}]\!] \quad \varepsilon_3 = \mathcal{G}[\![\widetilde{g_c \vee g \preccurlyeq g'}]\!]}{\Gamma; \Sigma; g_c \vdash t_1\ t_2 \rightsquigarrow \varepsilon_1 t_1'\ @_{\varepsilon_3}\ \varepsilon_2 t_2' : U_{12} \mathbin{\widetilde{\vee}} g}$$

$$(T\mathrm{if})\ \frac{\Gamma; \Sigma; g_c \vdash t_1 \rightsquigarrow t_1' : \mathsf{Bool}_g \quad g_c' = g_c \mathbin{\widetilde{\vee}} g \quad \Gamma; \Sigma; g_c' \vdash t_2 \rightsquigarrow t_2' : U_2 \quad \Gamma; \Sigma; g_c' \vdash t_3 \rightsquigarrow t_3' : U_3 \quad \varepsilon_1 = \mathcal{G}^{\cup}[\![\mathsf{Bool}_g]\!] \quad \varepsilon_2 = \mathcal{G}[\![\widetilde{U_2 <: U_2 \mathbin{\ddot{\vee}} U_3}]\!] \quad \varepsilon_3 = \mathcal{G}[\![\widetilde{U_3 <: U_2 \mathbin{\ddot{\vee}} U_3}]\!]}{\Gamma; \Sigma; g_c \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightsquigarrow \mathsf{if}\ \varepsilon_1 t_1\ \mathsf{then}\ \varepsilon_2 t_2\ \mathsf{else}\ \varepsilon_3 t_3 : (U_2 \mathbin{\widetilde{\ddot{\vee}}} U_3) \mathbin{\widetilde{\vee}} g}$$

$$(T\mathrm{assgn})\ \frac{\Gamma; \Sigma; g_c \vdash t_1 \rightsquigarrow t_1' : \mathsf{Ref}_g\ U_1 \quad \Gamma; \Sigma; g_c \vdash t_2 \rightsquigarrow t_2' : U_2 \quad \varepsilon_1 = \mathcal{G}^{\cup}[\![\mathsf{Ref}_g\ U_1]\!] \quad \varepsilon_2 = \mathcal{G}[\![U_2 \lesssim U_1]\!] \quad \varepsilon_3 = \mathcal{G}[\![\widetilde{g_c \vee g \preccurlyeq \mathit{label}(U_1)}]\!]}{\Gamma; \Sigma; g_c \vdash t_1 := t_2 \rightsquigarrow \varepsilon_1 t_1' :=_{\varepsilon_3} \varepsilon_2 t_2' : \mathsf{Unit}_\perp}$$

$$(T\mathrm{ref})\ \frac{\Gamma; \Sigma; g_c \vdash t \rightsquigarrow t' : U' \quad \varepsilon_1 = \mathcal{G}[\![U' \lesssim U]\!] \quad \varepsilon_2 = \mathcal{G}[\![g_c \mathbin{\widetilde{\preccurlyeq}} \mathit{label}(U)]\!]}{\Gamma; \Sigma; g_c \vdash \mathsf{ref}^U\ t \rightsquigarrow \mathsf{ref}^U_{\varepsilon_2}\ \varepsilon_1 t' : \mathsf{Ref}_\perp\ U} \qquad (T\mathrm{deref})\ \frac{\Gamma; \Sigma; g_c \vdash t \rightsquigarrow t' : \mathsf{Ref}_g\ U \quad \varepsilon = \mathcal{G}^{\cup}[\![\mathsf{Ref}_g\ U]\!]}{\Gamma; \Sigma; g_c \vdash !t \rightsquigarrow !\varepsilon t' : U \mathbin{\widetilde{\vee}} g}$$

$$(T::)\ \frac{\Gamma; \Sigma; g_c \vdash t \rightsquigarrow t' : U_1 \quad \varepsilon = \mathcal{G}[\![U_1 \lesssim U_2]\!]}{\Gamma; \Sigma; g_c \vdash t :: U_2 \rightsquigarrow \varepsilon t' : U_2}$$

where $\mathcal{G}^{\cup}[\![g]\!] = \mathcal{G}[\![g \mathbin{\widetilde{\preccurlyeq}} g]\!]$ and $\mathcal{G}^{\cup}[\![U]\!] = \mathcal{G}[\![U \lesssim U]\!]$

Fig. 5. $\mathrm{GSL}_{\mathrm{Ref}}$: elaboration to $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$ terms

The *bounds* function produces the label interval that corresponds to a given gradual label, *i.e.* $\mathit{bounds}(?) = [\bot, \top]$ and $\mathit{bounds}(\ell) = [\ell, \ell]$. The *interior operator intr* computes the smallest sub-intervals of its arguments that include all plausible orderings.[9] Given two intervals ι_1 and ι_2, $\mathit{intr}(\iota_1, \iota_2)$ yields the greatest pair of sub-intervals $\langle \iota_1', \iota_2' \rangle \sqsubseteq \langle \iota_1, \iota_2 \rangle$ such that each label ℓ_1 in the interval ι_1' is less than some label ℓ_1 in ι_2', and each label in ι_2' is greater than some label in ι_1'. Formally:

$$\mathit{intr}([\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}]) = \langle [\ell_{11}, \ell_{12} \wedge \ell_{22}], [\ell_{11} \vee \ell_{21}, \ell_{22}] \rangle$$

[9] In Garcia et al. [2016], the interior and initial evidence operators coincide under the name "interior" because both operate on pairs of gradual types. By distinguishing between intervals and labels, the present development induces a corresponding distinction between these notions.

This operation only changes the upper-bound of the lower interval and the lower-bound of the upper interval. The resulting intervals are well-defined because we only use this operator in $\mathcal{J}$ after consistent label ordering is already known to hold.

Similarly, the initial evidence of a consistent judgment $\widetilde{g_1 \vee g_2 \preccurlyeq g_3}$ is computed as

$$\mathcal{J}[\![\widetilde{g_1 \vee g_2 \preccurlyeq g_3}]\!] = \mathit{intr}(\mathit{bounds}(g_1) \vee \mathit{bounds}(g_2), \mathit{bounds}(g_3))$$

This definition uses join of intervals, defined as $[\ell_1, \ell_2] \vee [\ell_1', \ell_2'] = [\ell_1 \vee \ell_1', \ell_2 \vee \ell_2']$. For instance, the initial evidence for consistent judgment $\widetilde{? \vee \mathsf{H} \preccurlyeq ?}$ is:

$$\begin{aligned}\mathcal{J}[\![\widetilde{? \vee \mathsf{H} \preccurlyeq ?}]\!] &= \mathit{intr}(\mathit{bounds}(?) \vee \mathit{bounds}(\mathsf{H}), \mathit{bounds}(?)) \\ &= \mathit{intr}([\mathsf{H}, \top], [\bot, \top]) \\ &= \langle [\mathsf{H}, \top], [\mathsf{H}, \top] \rangle\end{aligned}$$

A generalized definition of $\mathcal{J}$, considering any consistent bounding judgment can be found in Fig. 18. The definition of $\mathcal{J}$ extends naturally to compute the initial evidence for consistent subtyping judgments (the complete definition can be found in Fig. 19). For instance, in the (Tif) rule, $\mathcal{J}[\![\widetilde{U_2 <: U_2 \dot{\vee} U_3}]\!]$ computes the initial evidence for the consistent lifting of the fact that the type of the first branch is a subtype of the type of the entire conditional expression.

Rule (T ::) recursively translates the subterm t, and the consistent subtyping judgment $U_1 <: U_2$ from (S ::) is replaced with $\mathcal{J}[\![U_1 \lesssim U_2]\!]$, which computes evidence ε for consistent subtyping. This evidence is eventually placed next to the translated term t'. The ascription itself is erased because it does not affect the results of the computation.

Rule (Tapp) works similarly. Since t_1 is not constrained by a consistent subtyping judgment, the rule generates evidence for *reflexive* consistent subtyping: that the type is a consistent subtype of itself, $\mathcal{J}^{\cup}[\![U_{11} \xrightarrow{g'}_{g} U_{12}]\!]$. This seemingly vacuous evidence evolves nontrivially as a program reduces. Evidence for the judgment $\widetilde{g_c \vee g \preccurlyeq g'}$ is computed as $\mathcal{J}[\![\widetilde{g_c \vee g \preccurlyeq g'}]\!]$, and placed next to the @ symbol, since it does not logically belong to any subterm.

The rest of the translation rules are analogous: each term is translated recursively, judgments are replaced by functions that determine the corresponding initial evidence, and the evidence for reflexive consistent subtyping $\mathcal{J}^{\cup}_{<:}$ is associated to otherwise unconstrained types.

As an example, consider the $\mathrm{GSL}_{\mathsf{Ref}}$ program $x{:=}\mathsf{true}_?$, with current security effect L and environment $\Gamma \triangleq x : \mathsf{Ref}_? \ \mathsf{Bool}_{\mathsf{H}}$. It elaborates to $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ as follows:

$$\text{(Tassgn)}\ \frac{\begin{array}{c}\Gamma;.;\mathsf{L} \vdash x \rightsquigarrow x : \mathsf{Ref}_? \ \mathsf{Bool}_{\mathsf{H}} \qquad \Gamma;.;\mathsf{L} \vdash \mathsf{true}_? \rightsquigarrow \mathsf{true}_? : \mathsf{Bool}_? \\ \varepsilon_1 = \mathcal{J}^{\cup}[\![\mathsf{Ref}_? \ \mathsf{Bool}_{\mathsf{H}}]\!] = \langle \mathsf{Ref}_{[\bot,\top]} \ \mathsf{Bool}_{[\mathsf{H},\mathsf{H}]}, \mathsf{Ref}_{[\bot,\top]} \ \mathsf{Bool}_{[\mathsf{H},\mathsf{H}]} \rangle \\ \varepsilon_2 = \mathcal{J}[\![\mathsf{Bool}_? \lesssim \mathsf{Bool}_{\mathsf{H}}]\!] = \langle \mathsf{Bool}_{[\bot,\mathsf{H}]}, \mathsf{Bool}_{[\mathsf{H},\mathsf{H}]} \rangle \\ \varepsilon_3 = \mathcal{J}[\![\widetilde{\mathsf{L} \vee ? \preccurlyeq \mathsf{H}}]\!] = \langle [\mathsf{L},\mathsf{H}], [\mathsf{H},\mathsf{H}] \rangle\end{array}}{\Gamma;.;\mathsf{L} \vdash x{:=}\mathsf{true}_? \rightsquigarrow \varepsilon_1 x :=_{\varepsilon_3} \varepsilon_2 \mathsf{true}_? : \mathsf{Unit}_{\bot}}$$

Evolving evidence. During reduction, evidence for consistent judgments must be combined to justify each reduction step. This combination is realized by two operators: *consistent transitivity for label ordering* and *consistent join monotonicity*.

The consistent transitivity operator $\circ^{\preccurlyeq}$ attempts to combine evidence for $g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$ and $g_2 \mathrel{\widetilde{\preccurlyeq}} g_3$ to produce evidence for $g_1 \mathrel{\widetilde{\preccurlyeq}} g_3$. Since $\mathrel{\widetilde{\preccurlyeq}}$ is not in general transitive, $\circ^{\preccurlyeq}$ is partial, giving rise to runtime errors. For instance, both $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ and $? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$ hold, but can they be combined to deduce that $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$? Of course not, otherwise high-confidence data could flow to low-confidence positions. To

understand this failure of consistent transitivity, consider the initial evidence for these judgments, $\langle [\mathsf{H},\mathsf{H}],[\mathsf{H},\top]\rangle$ and $\langle [\bot,\mathsf{L}],[\mathsf{L},\mathsf{L}]\rangle$. They cannot be combined because "they do not meet in the middle", *i.e.* the middle intervals $[\mathsf{H},\top]$ and $[\bot,\mathsf{L}]$ share no labels in common, which would justify transitivity. This intuition is formalized as follows:

$$\langle \iota_1, \iota_{21}\rangle \circ^{\preccurlyeq} \langle \iota_{22}, \iota_3\rangle = \triangle^{\preccurlyeq}(\iota_1, \iota_{21} \sqcap \iota_{22}, \iota_3)$$

$$\text{where} \quad [\ell_1,\ell_2] \sqcap [\ell_1',\ell_2'] = [\ell_1 \curlyvee \ell_1', \ell_2 \curlywedge \ell_2'] \quad \text{if} \quad \ell_1 \curlyvee \ell_1' \preccurlyeq \ell_2 \curlywedge \ell_2'$$

$$\text{and} \quad \triangle^{\preccurlyeq}([\ell_1,\ell_2],[\ell_1',\ell_2'],[\ell_1'',\ell_2'']) =$$

$$\langle [\ell_1, \ell_2 \curlywedge \ell_2' \curlywedge \ell_2''], [\ell_1 \curlyvee \ell_1' \curlyvee \ell_1'', \ell_2'']\rangle \quad \text{if} \quad \ell_1 \preccurlyeq \ell_2',\ \ell_1' \preccurlyeq \ell_2'',\ \ell_1 \preccurlyeq \ell_2''$$

The meet operator $\sqcap$ denotes the intersection of two intervals. Given three intervals $\iota_1, \iota_2, \iota_3$, the $\triangle^{\preccurlyeq}$ operator calculates, if possible, a pair of intervals $\langle \iota_1', \iota_3'\rangle \sqsubseteq \langle \iota_1, \iota_3\rangle$ such that transitivity of label ordering through elements of ι_2 is always plausible. Both operators are undefined if their side conditions do not hold.

The consistent join monotonicity operator $\widetilde{\curlyvee}$ reflects another facet of reasoning about consistent ordering relationships. Recall from Fig. 2 that during reduction, labels are sometimes joined, either for stamping values or for augmenting the security effect. Similarly, in $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ evidence must be combined to support new consistent judgments that involve these joined labels. Consistent join monotonicity combines evidence for $g_1 \mathrel{\widetilde{\preccurlyeq}} g_2$ and $g_3 \mathrel{\widetilde{\preccurlyeq}} g_4$ to produce evidence for $\widetilde{g_1 \curlyvee g_3 \preccurlyeq g_2 \curlyvee g_4}$, the consistent lifting of the static judgment $\ell_1 \curlyvee \ell_3 \preccurlyeq \ell_2 \curlyvee \ell_4$.

$$\langle \iota_1, \iota_2\rangle \mathrel{\widetilde{\curlyvee}} \langle \iota_1', \iota_2'\rangle = \langle \iota_1 \curlyvee \iota_1', \iota_2 \curlyvee \iota_2'\rangle$$

In contrast to consistent transitivity, this operator is total.

Lifting these label operators to types is direct, albeit verbose, and can be found in Appendix A.5. These type operators inherit properties from the label operators, *e.g.* consistent transitivity of subtyping $\circ^{<:}$ is partial just like consistent transitivity of label ordering.

Reduction rules. Fig. 6 presents reduction semantics for $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$. Reduction operates on configurations $\mathbb{C}$, which consist of a term and a store, and a security effect. Specifically, $t_1 \mid \mu_1 \overset{\varepsilon\, g_c}{\longmapsto} t_2 \mid \mu_2$ denotes the reduction of term t_1 in store μ_1 to term t_2 in store μ_2 under security effect g_c; the label evidence ε confirms that the runtime security effect is a sublabel of the label that was used statically to type check the original term (and is preserved by reduction).

The semantics is defined using two notions of reduction, $\longrightarrow$ and $\longrightarrow_{<:}$. The rules directly mirror the rules of $\mathrm{SSL}_{\mathsf{Ref}}$ (Fig. 2), except that they also manage evidence at subexpression borders and combine evidence as needed to justify the preserved typing of the contractum. If evidence fails to combine, the program ends with an **error**.

A word about notation: to select evidences for sub-components of types, we use evidence inversion functions [Garcia et al. 2016]. For instance, given a function type evidence ε, $\mathit{idom}(\varepsilon)$ (resp. $\mathit{icod}(\varepsilon)$) retrieves the type evidence of the domain (resp. co-domain). Similarly, ilat retrieves latent effect evidence from the evidence for a function type, and iref performs likewise for reference types. Finally, given type evidence ε, $\mathit{ilbl}(\varepsilon)$ yields the corresponding label evidence.

We now describe each reduction rule in turn.

- Rule $(r1)$ reduces a binary operation by joining the evidence of both operands to confirm that type preservation holds.
- Rule $(r2)$ reduces a protected value by stamping the security effect of the prot on the value and joining both evidences accordingly. We stamp g_1 on the value to prevent it from leaking

$$(r1)\quad \varepsilon_1(b_1)_{g_1} \oplus \varepsilon_2(b_2)_{g_2} \mid \mu \xrightarrow{\varepsilon g_c} (\varepsilon_1 \mathbin{\widetilde{\vee}} \varepsilon_2)(b_1 \;[\![\oplus]\!]\; b_2)_{(g_1 \mathbin{\widetilde{\vee}} g_2)} \mid \mu$$

$$\boxed{\xrightarrow{\varepsilon g_c} : \mathbb{C} \times (\mathbb{C} \cup \{\,\textbf{error}\,\})}$$

$$(r2)\quad \mathrm{prot}_{\varepsilon_1 g_1 \varepsilon_2 g_2}(\varepsilon_3 u) \mid \mu \xrightarrow{\varepsilon g_c} (\varepsilon_3 \mathbin{\widetilde{\vee}} \varepsilon_1)(u \mathbin{\widetilde{\vee}} g_1) \mid \mu$$

$$(r3)\quad \varepsilon_1(\lambda^{g'} x : U.t)_g \; @_{\varepsilon_3}\; \varepsilon_2 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \mathrm{prot}_{ilbl(\varepsilon_1)g} \varepsilon_1' g_1'(icod(\varepsilon_1)([\varepsilon_2' u/x]t)) \mid \mu \\ \textbf{error} \qquad \text{if } \varepsilon_1' \text{ or } \varepsilon_2' \text{ are not defined} \end{cases}$$

where:
$$\varepsilon_1' = (\varepsilon \mathbin{\widetilde{\vee}} ilbl(\varepsilon_1)) \circ^{\preccurlyeq} \varepsilon_3 \circ^{\preccurlyeq} ilat(\varepsilon_1)$$
$$\varepsilon_2' = \varepsilon_2 \circ^{<:} idom(\varepsilon_1)$$
$$g_1' = (g_c \mathbin{\widetilde{\vee}} g)$$

$$(r4)\quad \text{if } \varepsilon_1 b_{g_1} \text{ then } t_2 \text{ else } t_3 \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \mathrm{prot}_{ilbl(\varepsilon_1)g_1} \varepsilon' g'(\varepsilon_2 t_2) \mid \mu & \text{if } b = \text{true} \\ \mathrm{prot}_{ilbl(\varepsilon_1)g_1} \varepsilon' g'(\varepsilon_3 t_3) \mid \mu & \text{if } b = \text{false} \end{cases}$$

where:
$$\varepsilon' = \varepsilon \mathbin{\widetilde{\vee}} ilbl(\varepsilon_1)$$
$$g' = g_c \mathbin{\widetilde{\vee}} g_1$$

$$(r5)\quad \mathrm{ref}^U_{\varepsilon_2}\; \varepsilon_1 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} o_\perp \mid \mu[o \mapsto \varepsilon'(u \mathbin{\widetilde{\vee}} g_c)] \\ \textbf{error} \qquad \text{if } (\varepsilon \circ^{\preccurlyeq} \varepsilon_2) \text{ is not defined} \end{cases}$$

where:
$$o \notin dom(\mu)$$
$$\varepsilon' = \varepsilon_1 \mathbin{\widetilde{\vee}} (\varepsilon \circ^{\preccurlyeq} \varepsilon_2)$$

$$(r6)\quad !\varepsilon_1 o_g \mid \mu \xrightarrow{\varepsilon g_c} \mathrm{prot}_{ilbl(\varepsilon_1)g} \varepsilon' g'(iref(\varepsilon_1) v)$$

where:
$$\mu(o) = v$$
$$\varepsilon' = \varepsilon \mathbin{\widetilde{\vee}} ilbl(\varepsilon_1)$$
$$g' = g_c \mathbin{\widetilde{\vee}} g$$

$$(r7)\quad \varepsilon_1 o_g :=_{\varepsilon_3} \varepsilon_2 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \mathrm{unit}_\perp \mid \mu[o \mapsto \varepsilon'(u \mathbin{\widetilde{\vee}} (g_c \mathbin{\widetilde{\vee}} g))] \\ \textbf{error} \qquad \text{if } \varepsilon' \text{ is not defined, or } \varepsilon \mathbin{\lfloor \preccurlyeq \rfloor} ilbl(\varepsilon'') \text{does not hold} \end{cases}$$

where:
$$\mu(o) = \varepsilon'' u'$$
$$\varepsilon' = (\varepsilon_2 \circ^{<:} iref(\varepsilon_1)) \mathbin{\widetilde{\vee}} ((\varepsilon \mathbin{\widetilde{\vee}} ilbl(\varepsilon_1)) \circ^{\preccurlyeq} \varepsilon_3 \circ^{\preccurlyeq} ilbl(iref(\varepsilon_1)))$$

$$\varepsilon_1(\varepsilon_2 u) \longrightarrow_{<:} \begin{cases} (\varepsilon_2 \circ^{<:} \varepsilon_1) u \\ \textbf{error} \qquad \text{if not defined} \end{cases}$$

$$\boxed{\longrightarrow_{<:} : \textsc{EvTerm} \times (\textsc{EvTerm} \cup \{\,\textbf{error}\,\})}$$

Fig. 6. $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$: Dynamic semantics

information to the current context when g_1 is more confidential than the current security effect g_c. Note that g_2—which represents the join between g_1 and the current security effect g_c—is not used in this rule; it is used during reduction of the protected subterm.

- Rule ($r3$) reduces a function application either to a protected body or to an error. The term reduces to an error if consistent transitivity fails to justify that the type of the actual argument is a consistent subtype of the formal argument type. This prevents an evident invalid information flow from the actual argument to the formal argument. Also, to prevent implicit flows via the store, an error is signaled if consistent transitivity fails to confirm that the latent effect of the function is greater than both the current security effect and that of the function. If the function application is valid, then the body is protected at the security level of the function. Label g_1' represents the security effect that is used to reduce the body, where ε_1' confirms that g_1' is no more confidential than the latent effect g'.
- Similarly, rule ($r4$) reduces a conditional expression by protecting the chosen branch. The resulting prot term is constructed using the dynamic information of the conditional.
- Rule ($r5$) reduces a reference term to a fresh location. To prevent invalid implicit flows, the current security effect is stamped on the stored value. The term reduces to an error if consistent transitivity fails to confirm that the current security effect is lower than the statically-determined security level of the reference content U.
- Rule ($r6$) reduces a dereference term. In the dynamic semantics of SSL_{Ref}, dereferencing a store location causes the actual security of the location to be stamped on the resulting value. Here, the term reduces instead to a protected expression, which is equivalent but simplifies the proofs.
- Rule ($r7$) is critical to ensuring noninterference. It can reduce to an error, and thereby preventing either implicit or explicit invalid flows, for three reasons:

(1) the security level of the stored value should be no more confidential than the statically-determined security level of the reference content (explicit flow).
(2) both the current security effect and the actual security level of the reference should be no more confidential than the static security level of the reference content (implicit flow).
(3) the evidence of the current security effect must denote possible labels that are *necessarily lower* than those denoted by the evidence of the stored value (implicit flow).

The third condition above, highlighted in gray in Fig. 6, is expressed with the lower-bound comparison operator $\lfloor \leq \rfloor$ between evidences:

$$\langle [\ell_1, \ell_2], [\ell_3, \ell_4] \rangle \lfloor \leq \rfloor \langle [\ell_1', \ell_2'], [\ell_3', \ell_4'] \rangle \iff \ell_3 \preccurlyeq \ell_3'$$

This check is necessary to ensure noninterference, and as explained in Sec. 6.3, it arises not from the type preservation argument, but from the noninterference argument. In Sec. 4.3 we illustrate each of these three scenarios.

The $\longrightarrow_{<:}$ reduction rule uses consistent transitivity to combine, if possible, strings of evidence that accumulate on a raw value. It fails with a runtime error if the evidence cannot be combined. Sec. 4.3 presents an example of such a reduction.

Finally, contextual term reduction is specified using *term frames* f and *evidence frames* h:

$$\begin{array}{lcl} f & ::= & h[\varepsilon[]] \\ h & ::= & \square \oplus \varepsilon t \mid \varepsilon u \oplus \square \mid \square \ @_\varepsilon \ \varepsilon t \mid \varepsilon u \ @_\varepsilon \ \square \mid \varepsilon \ \square \mid \text{if } \square \text{ then } \varepsilon t \text{ else } \varepsilon t \mid !\square \mid \square :=_\varepsilon \varepsilon t \mid \varepsilon u :=_\varepsilon \square \mid \text{ref}_\varepsilon^U \ \square \end{array}$$

The reduction rules for frames are presented in Fig. 7. Rule (Rf) reduces under term frames. Rule (R$\longrightarrow$) reduces a term to either a term or **error**, using $\longrightarrow$ from Fig. 6. Similarly Rules (Rh) and (Rprot$h$) reduce the subterm using the evidence-combining reduction $\longrightarrow_{<:}$. Rule (Rprot) allows the protected subterm to step under a higher security level, which may be a sublabel of the one

$$(\mathrm{R}{\longrightarrow})\ \frac{t \mid \mu \xrightarrow{\varepsilon g_c} r \qquad r \in \mathbb{C} \cup \{\,\mathbf{error}\,\}}{t \mid \mu \overset{\varepsilon g_c}{\longmapsto} r} \qquad (\mathrm{R}f)\ \frac{t \mid \mu \overset{\varepsilon g_c}{\longmapsto} t' \mid \mu'}{f[t] \mid \mu \overset{\varepsilon g_c}{\longmapsto} f[t'] \mid \mu'}$$

$$(\mathrm{Rprot})\ \frac{t \mid \mu \overset{\varepsilon' g'_c}{\longmapsto} t' \mid \mu'}{\mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon t) \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon t') \mid \mu'} \qquad (\mathrm{R}h)\ \frac{\varepsilon v \longrightarrow_{<:} \varepsilon' u}{h[\varepsilon v] \mid \mu \overset{\varepsilon g_c}{\longmapsto} h[\varepsilon' u] \mid \mu}$$

$$(\mathrm{Rprot}h)\ \frac{\varepsilon v \longrightarrow_{<:} \varepsilon' u}{\mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon v) \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon' u) \mid \mu} \qquad (\mathrm{R}f\mathrm{err})\ \frac{t \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathbf{error}}{f[t] \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathbf{error}}$$

$$(\mathrm{R}h\mathrm{err})\ \frac{\varepsilon v \longrightarrow_{<:} \mathbf{error}}{h[\varepsilon v] \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathbf{error}} \qquad (\mathrm{Rproterr})\ \frac{t \mid \mu \overset{\varepsilon' g'_c}{\longmapsto} \mathbf{error}}{\mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon t) \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathbf{error}}$$

$$(\mathrm{Rprot}h\mathrm{err})\ \frac{\varepsilon v \longrightarrow_{<:} \mathbf{error}}{\mathrm{prot}_{\varepsilon_1 g_1} \varepsilon' g'_c(\varepsilon v) \mid \mu \overset{\varepsilon g_c}{\longmapsto} \mathbf{error}}$$

Fig. 7. $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$: Evaluation frames and reduction

determined statically. Finally, rules (R$f$err) and (Rproterr) propagate errors when the subterm reduces to an error, and rules (R$h$err) and (Rprot$h$err) propagate errors when evidence fails to combine.

4.3 Examples of Reduction

To illustrate the runtime semantics of $\mathrm{GSL}_{\mathrm{Ref}}$ we first illustrate the three scenarios for which an assignment can fail, as per Rule (*r*7).

(1) Consider the following program, which attempts to assign a high-confidentiality value into a low-confidentiality reference, and its translation (under security effect $\bot$):

$$\bot \vdash \mathrm{ref}^{\mathsf{Int_L}}\ 20_{\mathsf{L}} := (10_{\mathsf{H}} :: \mathsf{Int}_?) \rightsquigarrow t : \mathsf{Unit}_{\bot}$$

Abbreviating $[\bot, \top]$ as $?$, $[\ell, \ell]$ as ℓ, $\langle \iota, \iota \rangle$ as $\langle \iota \rangle$, and $\_$ for irrelevant evidence, we have:

$$t \overset{\bot}{\longmapsto}{}^{*} \varepsilon_1 o_{\bot} :=_{\_} \varepsilon_2 10_{\mathsf{H}}$$

where $\varepsilon_1 = \langle \mathsf{Ref}_{\bot}\ \mathsf{Int_L} \rangle \vdash \mathsf{Ref}_{\bot}\ \mathsf{Int_L} \lesssim \mathsf{Ref}_{\bot}\ \mathsf{Int_L}$, $\varepsilon_2 = \langle \mathsf{Int_H}, \mathsf{Int}_{[\mathsf{H},\top]} \rangle \vdash \mathsf{Int_H} \lesssim \mathsf{Int}_?$. Then as $(\varepsilon_2 \circ^{<:} \mathit{iref}(\varepsilon_1)) = \langle \mathsf{Int_H}, \mathsf{Int}_{[\mathsf{H},\top]} \rangle \circ^{<:} \langle \mathsf{Int_L} \rangle$ is not defined, the term reduces to an error, as expected.

(2) The following program attempts to update a low-confidentiality reference under a high-confidentiality security effect. Considering a security effect $\bot$, a location $\vdash o_{\bot} : \mathsf{Ref}_{\bot}\ \mathsf{Int_L}$, the program and its translation are:

$$\bot \vdash \mathrm{if}\ \mathsf{true}_{\mathsf{H}} :: \mathsf{Bool}_?\ \mathrm{then}\ o_{\bot} := 10_{\mathsf{L}}\ \mathrm{else}\ \mathrm{unit} \rightsquigarrow t : \mathsf{Unit}_?$$

The conditional reduces to the first branch under a security effect H.

$$t \overset{\bot}{\longmapsto}{}^{*} \mathrm{prot}_{\_\mathsf{H}} \varepsilon_1 \mathsf{H}(\_(\varepsilon_2 o_{\bot} :=_{\varepsilon_3} \_10_{\mathsf{L}}))$$

where $\varepsilon_1 = \langle \mathsf{H}, [\mathsf{H}, \top] \rangle \vdash \bot \curlyvee \mathsf{H} \preccurlyeq \bot \curlyvee ?$ and $\varepsilon_2 = \langle \mathsf{Ref}_{\bot}\ \mathsf{Int_L} \rangle \vdash \mathsf{Ref}_{\bot}\ \mathsf{Int_L} \lesssim \mathsf{Ref}_{\bot}\ \mathsf{Int_L}$. Also, because the static security effect of the assignment is ?, we have $\varepsilon_3 = \langle [\bot, \mathsf{L}], \mathsf{L} \rangle \vdash ? \curlyvee \bot \preccurlyeq \mathsf{L}$.

Then as $((\varepsilon_1 \mathbin{\widetilde{\vee}} \mathit{ilbl}(\varepsilon_2)) \circ^{\preccurlyeq} \varepsilon_3 \circ^{\preccurlyeq} \mathit{ilbl}(\mathit{iref}(\varepsilon_2))) = \langle \mathsf{H}, [\mathsf{H}, \top] \rangle \circ^{\preccurlyeq} \langle [\bot, \mathsf{L}], \mathsf{L} \rangle \circ^{\preccurlyeq} \langle \mathsf{L} \rangle$ is not defined, the term reduces to an error, successfully preventing an invalid implicit flow.

(3) Consider a program fragment similar to the previous one, with security effect $\bot$, a variable $x : \mathsf{Bool}_{\mathsf{H}}$, and a location $\vdash o_{\bot} : \mathsf{Ref}_{\bot}\ \mathsf{Int}_{?}$:

$$\bot \vdash \mathsf{if}\ x :: \mathsf{Bool}_{?}\ \mathsf{then}\ o_{\bot} := 10_{\mathsf{H}}\ \mathsf{else}\ \mathsf{unit}_{?} \rightsquigarrow t : \mathsf{Unit}_{?}$$

Suppose as well that $\mu(o) = \varepsilon_2 0_{?}$, where $\mathit{ilbl}(\varepsilon_2) = \langle [\bot, \top], [\bot, \top] \rangle \vdash ? \mathbin{\widetilde{\preccurlyeq}} ?$ (*i.e.* the stored number and heap cell have not acquired any security commitments yet). If x is $\mathsf{true}_{\mathsf{H}}$, then the first branch is taken:

$$t \xrightarrow{-\bot}{}^{*} \mathsf{prot}_{\mathsf{H}} \varepsilon_1 \mathsf{H}(\_(\_o_{\bot} :=_{\_} \_10_{\mathsf{H}}))$$

where $\varepsilon_1 = \langle \mathsf{H}, [\mathsf{H}, \top] \rangle \vdash \overbrace{\bot \vee \mathsf{H} \preccurlyeq \bot \vee ?}$. Since $\varepsilon_1 \lfloor \leq \rfloor \mathit{ilbl}(\varepsilon_2)$ is not defined, because $\mathsf{H} \mathbin{\widetilde{\not\preccurlyeq}} \bot$, the program reduces to an error. The problem is that if x were changed to $\mathsf{false}_{\mathsf{H}}$, then the unchanged imprecisely labeled contents of o could be treated as low-security and thereby used to leak information about x, using for instance a test of $!o$ that conditionally assigns to some other low-security reference (for more see the example of Sec. 2, and Sec. 6.3).

Type-based reasoning. Finally, we revisit the *mix* and *smix* functions from Sec. 2, which illustrate how $\mathrm{GSL}_{\mathrm{Ref}}$ preserves type-based reasoning principles in the gradual setting. The desugared $\mathrm{GSL}_{\mathrm{Ref}}$ program follows:[10]

$$\begin{aligned}
&\mathit{mix} = (\lambda \mathit{pub} : \mathsf{L}.(\lambda \mathit{priv} : ?.(\mathsf{if}\ \mathit{pub} < \mathit{priv}\ \mathsf{then}\ 1_{\mathsf{L}}\ \mathsf{else}\ 2_{\mathsf{L}}) :: \mathsf{L})_{\mathsf{L}})_{\mathsf{L}} \\
&\mathit{smix} = \mathit{mix} :: \mathsf{L} \to \mathsf{H} \to \mathsf{L} \\
&\mathit{smix}\ 1_{\mathsf{L}}\ 5_{\mathsf{L}}
\end{aligned}$$

This program elaborates to the following $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$ program:

$$\begin{aligned}
&\mathit{mix} = (\lambda \mathit{pub} : \mathsf{L}.(\lambda \mathit{priv} : ?.\langle [\bot, \mathsf{L}], \mathsf{L} \rangle (\mathsf{if}\ \langle ? \rangle (\langle \mathsf{L} \rangle \mathit{pub} < \langle ? \rangle \mathit{priv})\ \mathsf{then}\ \langle \mathsf{L} \rangle 1_{\mathsf{L}}\ \mathsf{else}\ \langle \mathsf{L} \rangle 2_{\mathsf{L}}))_{\mathsf{L}})_{\mathsf{L}} \\
&\mathit{smix} = \langle \mathsf{L} \to [\mathsf{H}, \top] \to \mathsf{L}, \mathsf{L} \to \mathsf{H} \to \mathsf{L} \rangle \mathit{mix} \\
&\langle \mathsf{H} \to \mathsf{L} \rangle (\langle \mathsf{L} \to \mathsf{H} \to \mathsf{L} \rangle \mathit{smix} \mathbin{@}_{\langle [\mathsf{L}, \top] \rangle} \langle \mathsf{L} \rangle 1_{\mathsf{L}}) \mathbin{@}_{\langle [\mathsf{L}, \top] \rangle} \langle \mathsf{L}, \mathsf{H} \rangle 5_{\mathsf{L}}
\end{aligned}$$

A trace of the program is given in Fig. 8. As before, we abbreviate $[\bot, \top]$ as $?$, $[\ell, \ell]$ as ℓ, and $\langle \iota, \iota \rangle$ as $\langle \iota \rangle$. We omit the security effect of the reduction, which is always $\langle \bot \rangle \bot$, as well as the heap, since the program is pure. The program fails as expected because low-security evidence is attached to the conditional term by a static ascription, which fails to combine with the high-security evidence of the value produced by the conditional. In other words, reduction fails to prove that $\mathsf{H} \preccurlyeq \mathsf{L}$.

4.4 $\mathrm{GSL}_{\mathrm{Ref}}$: Safety and Graduality

$\mathrm{GSL}_{\mathrm{Ref}}$ satisfies a standard type safety property, whose proofs are in the companion technical report [Toro et al. 2018]. More precisely, type safety is formulated for the evidence-augmented language $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$, and hence appeals to a corresponding typing judgment. As expected, this typing judgment, denoted $\Gamma; \Sigma; \varepsilon g_c \vdash t : U$, is based on the $\mathrm{GSL}_{\mathrm{Ref}}$ typing judgment.[11] The only difference is that the security effect g_c is enriched with evidence ε. This evidence accounts for how the runtime security effect can evolve to (consistently) lower levels than the security effect originally determined by the type system.

[10] For brevity, we only show the labels of base types, and omit latent effect annotations on pure functions.

[11] The full definition of the $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$ type system can be found in Appendix A.3; the (straightforward) theorem that elaboration preserves typing is in the companion technical report [Toro et al. 2018] .

$$
\begin{aligned}
&\langle \mathsf{H} \to \mathsf{L}\rangle(\ \langle \mathsf{L} \to \mathsf{H} \to \mathsf{L}\rangle\langle \mathsf{L} \to [\mathsf{H},\top] \to \mathsf{L}, \mathsf{L} \to \mathsf{H} \to \mathsf{L}\rangle \mathit{mix}\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L}\rangle 1_{\mathsf{L}})\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L},\mathsf{H}\rangle 5_{\mathsf{L}} \\
\longmapsto &\langle \mathsf{H} \to \mathsf{L}\rangle(\ \langle \mathsf{L} \to [\mathsf{H},\top] \to \mathsf{L}, \mathsf{L} \to \mathsf{H} \to \mathsf{L}\rangle \mathit{mix}\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L}\rangle 1_{\mathsf{L}}\ )\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L},\mathsf{H}\rangle 5_{\mathsf{L}} \\
\longmapsto &\langle \mathsf{H} \to \mathsf{L}\rangle(\ \mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle [\mathsf{H},\top] \to \mathsf{L}, \mathsf{H} \to \mathsf{L}\rangle u)\ )\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L},\mathsf{H}\rangle 5_{\mathsf{L}} \\
&\quad \text{where } u = (\lambda priv : ?.\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\text{if } \langle ?\rangle(\langle \mathsf{L}\rangle\langle \mathsf{L}\rangle 1_{\mathsf{L}} < \langle ?\rangle priv) \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}}))_{\mathsf{L}} \\
&\quad \text{and } \phi' = \langle \mathsf{L},\top\rangle \mathsf{L} \\
\longmapsto &\ \langle \mathsf{H} \to \mathsf{L}\rangle(\langle [\mathsf{H},\top] \to \mathsf{L}, \mathsf{H} \to \mathsf{L}\rangle u)\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L},\mathsf{H}\rangle 5_{\mathsf{L}} \\
\longmapsto &\ \langle [\mathsf{H},\top] \to \mathsf{L}, \mathsf{H} \to \mathsf{L}\rangle u\ @_{\langle [\mathsf{L},\top]\rangle}\ \langle \mathsf{L},\mathsf{H}\rangle 5_{\mathsf{L}} \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\text{if } \langle ?\rangle(\ \langle \mathsf{L}\rangle\langle \mathsf{L}\rangle 1_{\mathsf{L}}\ < \langle ?\rangle\langle \mathsf{L},[\mathsf{H},\top]\rangle 5_{\mathsf{L}}) \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}})) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\text{if } \langle ?\rangle(\langle \mathsf{L}\rangle 1_{\mathsf{L}} < \ \langle ?\rangle\langle \mathsf{L},[\mathsf{H},\top]\rangle 5_{\mathsf{L}}\ ) \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}})) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\text{if } \langle ?\rangle(\ \langle \mathsf{L}\rangle 1_{\mathsf{L}} < \langle \mathsf{L},[\mathsf{H},\top]\rangle 5_{\mathsf{L}}\ ) \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}})) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\text{if } \ \langle ?\rangle(\langle \mathsf{L},[\mathsf{H},\top]\rangle \mathsf{true}_{\mathsf{L}})\ \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}})) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle(\ \text{if } (\langle \mathsf{L},[\mathsf{H},\top]\rangle \mathsf{true}_{\mathsf{L}}) \text{ then } \langle \mathsf{L}\rangle 1_{\mathsf{L}} \text{ else } \langle \mathsf{L}\rangle 2_{\mathsf{L}}\ )) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\langle [\bot,\mathsf{L}],\mathsf{L}\rangle\ \mathsf{prot}_{\langle \mathsf{L},[\mathsf{H},\top]\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle 1_{\mathsf{L}})\ ) \\
\longmapsto &\mathsf{prot}_{\langle \mathsf{L}\rangle \mathsf{L}} \phi'(\langle \mathsf{L}\rangle\ \langle [\bot,\mathsf{L}],\mathsf{L}\rangle\langle \mathsf{L},[\mathsf{H},\top]\rangle 1_{\mathsf{L}}\ ) \\
\longmapsto &\textbf{error} \qquad \langle \mathsf{L},[\mathsf{H},\top]\rangle \circ^{\preccurlyeq} \langle [\bot,\mathsf{L}],\mathsf{L}\rangle \text{ is undefined}
\end{aligned}
$$

Fig. 8. $\mathrm{GSL}^{\varepsilon}_{\mathrm{Ref}}$: Example reduction

Proposition 4.1 (Type Safety). *If* $\cdot;\Sigma;\varepsilon g_c \vdash t : U$, *and consider* μ, *such that* $\Sigma \vdash \mu$, *then either:*

- t *is a value* v
- $t \mid \mu \overset{\varepsilon g_c}{\longmapsto}$ **error**
- $t \mid \mu \overset{\varepsilon g_c}{\longmapsto} t' \mid \mu'$ *and* $\cdot;\Sigma';\varepsilon g_c \vdash t' : U$ *for some* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \vdash \mu'$

Additionally, by design, the type system of $\mathrm{GSL}_{\mathrm{Ref}}$ is *crisply* and *smoothly* connected to that of $\mathrm{SSL}_{\mathrm{Ref}}$. First, the two typing judgments are crisply connected in that the $\mathrm{GSL}_{\mathrm{Ref}}$ judgment conservatively extends the $\mathrm{SSL}_{\mathrm{Ref}}$ one.

Proposition 4.2 (Static conservative extension). *Let* $\vdash_S$ *denote SSL*$_{Ref}$*'s type system. Then for any static language term* $t \in \textsc{Term}$, $\cdot;\Sigma;\ell_c \vdash_S t : S$ *if and only if* $\cdot;\Sigma;\ell_c \vdash t : S$.

Second, the two typing judgments are smoothly connected in that each well-typed $\mathrm{GSL}_{\mathrm{Ref}}$ program (thus each $\mathrm{SSL}_{\mathrm{Ref}}$ one) preserves well-typing as its security information is made *less* precise, a property known as the *static gradual guarantee* [Siek et al. 2015]. Precision orders the static information content of gradual type or labels from most to least. Type and label precision are defined as follows:

Definition 4.3 (Type and label precision).

$$\frac{}{g \sqsubseteq ?} \qquad \frac{}{g \sqsubseteq g} \qquad \frac{g_1 \sqsubseteq g_2}{\mathsf{Bool}_{g_1} \sqsubseteq \mathsf{Bool}_{g_2}} \qquad \frac{g_1 \sqsubseteq g_2}{\mathsf{Unit}_{g_1} \sqsubseteq \mathsf{Unit}_{g_2}}$$

$$\frac{U_{11} \sqsubseteq U_{21} \quad U_{12} \sqsubseteq U_{22} \quad g_{11} \sqsubseteq g_{21} \quad g_{12} \sqsubseteq g_{22}}{U_{11} \xrightarrow{g_{12}}_{g_{11}} U_{12} \sqsubseteq U_{21} \xrightarrow{g_{22}}_{g_{21}} U_{22}} \qquad \frac{g_1 \sqsubseteq g_2 \quad U_1 \sqsubseteq U_2}{\mathsf{Ref}_{g_1}\ U_1 \sqsubseteq \mathsf{Ref}_{g_2}\ U_2}$$

Type and label precision are naturally lifted to *term precision*.

Proposition 4.4 (Static gradual guarantee). *Suppose* $g_{c1} \sqsubseteq g_{c2}$ *and* $t_1 \sqsubseteq t_2$. *If* $\cdot;\cdot;g_{c1} \vdash t_1 : U_1$ *then* $\cdot;\cdot;g_{c2} \vdash t_2 : U_2$ *where* $U_1 \sqsubseteq U_2$.

This guarantee is best understood in reverse: if a *simply-typed* program (where all security labels are ?) has a security-typed counterpart (where all security labels are precise), then GSL_{Ref} statically accepts *every* intermediate security typing of that program: type checking is continuous with respect to security precision, so security information can be added in any order and at any rate [Siek et al. 2015].

Siek et al. [2015] also present a *dynamic* gradual guarantee, which relates the execution behavior of programs that only differ in their precision. Specifically, if a program takes a step, then the same program with less precise (or fewer) type annotations also takes a step, *i.e.* reducing precision does not introduce new runtime errors. The formal statement of the guarantee can be found in the companion technical report [Toro et al. 2018]. Unfortunately, we have uncovered a tension between the dynamic gradual guarantee and noninterference. To ensure noninterference, the dynamic semantics of GSL_{Ref} includes a specific runtime check (highlighted in gray in Fig. 6) which breaks the dynamic gradual guarantee. Dually, without this check, GSL_{Ref} satisfies the dynamic gradual guarantee, but does not enforce noninterference for all programs. We discuss this subtlety in more detail in Sec. 6.3.

Nevertheless, an interesting conservative extension result holds for the dynamic semantics. Specifically, static GSL_{Ref} terms never produce errors at runtime.

Proposition 4.5 (Static terms do not fail). *Let* StaticTerm *be the static subset of* $\text{GSL}^{\varepsilon}_{\text{Ref}}$ *terms,* i.e. *with fully-static annotations, and* StaticStore *the set of stores whose codomains are subsets of* StaticTerm. *Then consider* $t \in$ StaticTerm, $\mu \in$ StaticStore, *and* $\varepsilon\ell_c$ *such that* $\varepsilon = \mathcal{J}[\![\ell_c \mathrel{\widetilde{\preccurlyeq}} \ell'_c]\!]$. *If* $\cdot;\Sigma;\varepsilon\ell_c \vdash t : U$, *then either* t *is a value, or* $t \mid \mu_s \overset{\varepsilon\ell_c}{\longmapsto} t'_s \mid \mu'_s$, *with* $t' \in$ StaticTerm *and* $\mu' \in$ StaticStore.

4.5 Prototype Implementation

We have implemented GSL_{Ref} in an interactive prototype available online at:
https://pleiad.cl/gradual-security/.
The implementation, realized in Scala, supports all of GSL_{Ref} plus let-bindings. Given a source program, it either shows the result of the elaboration to $\text{GSL}^{\varepsilon}_{\text{Ref}}$, or reports a static type error. If the source program is well-typed, the evidence-augmented term can be explored interactively, either collapsing or expanding premises of its well-typedness, including evidences. The user can then reduce the term step by step, similarly to PLT Redex's trace facility. At each step, the full typing derivation of the term can again be explored. The reduction shows how evidences are combined by consistent subtyping transitivity, eventually ending up in a value or a runtime security error.

All examples presented in this paper are available as pre-loaded source examples.

5 $\mathsf{GSL}_{\mathsf{Ref}}$: NONINTERFERENCE

This section establishes the type *soundness* of $\mathsf{GSL}_{\mathsf{Ref}}$, *i.e.* that gradual security types ensure noninterference. Noninterference formalizes the intuition that low-security observers of a computation cannot detect changes in high-security inputs. Therefore noninterference inherently reflects a relationship between different runs of the same program with different inputs. We establish noninterference for $\mathsf{GSL}_{\mathsf{Ref}}$ using logical relations [Heintze and Riecke 1998; Zdancewic 2002]. More precisely, because general references introduce nontermination, we apply step-indexed relations [Ahmed 2004]. As standard, we focus on *termination-insensitive* noninterference: interference between two executions is only acknowledged when both terminate in values that are observably different. In line with prior work on gradual security [Disney and Flanagan 2011; Fennell and Thiemann 2013], we consider runtime check errors to be akin to non-termination, because in principle the semantics could deal with errors by diverging and directly reporting the error through a secure channel.

Observing values. The security type of a value dictates both an observation protocol and the clearance required to observe it. Consider a value $\vdash v : U_1 \to_g U_2$, and an observer with security level ℓ_o: Can ℓ_o observe the value? If so, what observations can it make? First, ℓ_o cannot make *any* observations if its security level does not subsume that of the function ($g \not\precsim \ell_o$). If clearance is granted ($g \precsim \ell_o$), then ℓ_o may make observations in accordance with the structure of v's type: it may construct another value $v' : U_1$ and apply it to the function; the observations that ℓ_o can make of the result are then dictated by the type $U_2 \mathbin{\widetilde{\vee}} g$.

The predicate obsVal_{ℓ_o}, defined formally below, intuitively captures what it means for a value v of type U to be *observable at* ℓ_o: ℓ_o must be consistently greater than the security label of U. To account for the gradual security setting, we need to extend this intuitive notion in two ways. First, observation must deal with the potential for values to carry type ascriptions, such as $v = \mathsf{true}_{\mathsf{H}} :: \mathsf{Bool}_?$. An observer at security level L must *not* observe the underlying high-security value. The key intuition is that the observation should ultimately be equivalent to applying the source language context if $\square :: \mathsf{Bool}_{\mathsf{L}}$ then $\mathsf{true}_{\mathsf{L}}$ else $\mathsf{false}_{\mathsf{L}}$ to the value, thereby asserting credentials and then using them. Doing so would trigger a runtime check error, which amounts to a non-observation. In $\mathsf{GSL}^{\varepsilon}_{\mathsf{Ref}}$, v would be represented as an evidence value $\varepsilon\mathsf{true}_{\mathsf{H}}$, where ε confirms that $\mathsf{Bool}_{\mathsf{H}} \lesssim \mathsf{Bool}_?$. We capture the observability of the underlying value by defining the notion of *observable evidence* at a given observation level. Then, an evidence value $v = \varepsilon u$ is observable if its label evidence ($\mathit{ilbl}(\varepsilon)$) is observable.

Definition 5.1 (Observable evidence). Suppose observation level ℓ_o and an evidence judgment $\varepsilon \vdash g \mathbin{\widetilde{\preccurlyeq}} g'$ for some g and g'. For the evidence ε to be observable at ℓ_o, it must be possible to confirm $g \mathbin{\widetilde{\preccurlyeq}} \ell_o$ using consistent transitivity of label ordering through g'. Formally:

$$\mathsf{obsEv}^{g'}_{\ell_o}(\varepsilon) \iff \varepsilon \circ^{\preccurlyeq} \mathcal{J}[\![g' \mathbin{\widetilde{\preccurlyeq}} \ell_o]\!] \text{ is defined}$$

Second, observation must account for dynamic security effect clearance: observation leaks a value from its context, so the observer must have the proper credentials. Recall that execution happens under a dynamic security effect g that, at runtime, can be consistently lower than the security effect originally determined by the type system. Therefore the dynamic security effect is accompanied by evidence ε that confirms that $g \mathbin{\widetilde{\preccurlyeq}} g'$, where g' is the static security effect. Observation is allowed if such evidence is observable, *i.e.* $g \mathbin{\widetilde{\preccurlyeq}} \ell_o$.

Adding these two refinements of observability to the original notion of observable value yields the following definition.

Definition 5.2 (Observable value). Given an observation level ℓ_o, we define that a value v, typed as U, is observable as:

$$\mathsf{obsVal}^{U}_{\ell_o}(v) \iff g \mathrel{\widetilde{\preccurlyeq}} \ell_o \wedge \Big((v = \varepsilon_1 u) \implies \mathsf{obsEv}^{g}_{\ell_o}(\mathit{ilbl}(\varepsilon_1))\Big) \quad \text{where } g = \mathit{label}(U)$$

Security logical relations. We define logical relations between both computations and values in Figs. 9 and 10. The notions of related values and related computations are mutually recursive, as explained below. Note that the logical relations are only defined for pairs of $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms that have the *same* type U, so simple type safety ensures that the behaviors dictated by U will produce defined behavior (including runtime error). To make the relations well-defined in the presence of nontermination, we index them on the number of steps k that the observer ℓ_o may take. If no inequivalent observations are made after k steps, the terms are deemed equivalent. Ultimately we require that ℓ_o observes equivalence for any arbitrary number of steps, which implies that nonterminating computations also respect the noninterference guarantees. This is the essence of step-indexing [Ahmed 2004].

The definition of *related values* is presented in Fig. 9. We use notation $\hat{g}_i$ to denote the evidence-augmented security context $\varepsilon_i g_i$. The notation $\Sigma; g_c \vdash \langle \hat{g}_1, v_1, \mu_1\rangle \approx^{k}_{\ell_o} \langle \hat{g}_2, v_2, \mu_2\rangle : U$ indicates that the triple of security context $\hat{g}_1$, value v_1 and store μ_1, is related to the triple of dynamic security context $\hat{g}_2$, value v_2 and store μ_2 at type U for k steps under store typing Σ and static security context g_c when observed at the security level ℓ_o. For two such triples to be related, four conditions must be satisfied:

(1) The security effects must be related under security effect g_c, meaning they denote execution contexts that are either both above ℓ_o (high-security), or both below (low-security). Formally, two security effects are related if their underlying evidences are either both observable or both not observable:

$$g_c \vdash \varepsilon_1 g_1 \approx_{\ell_o} \varepsilon_2 g_2 \iff (\mathsf{obsEv}^{g_c}_{\ell_o}(\varepsilon_1) \wedge \mathsf{obsEv}^{g_c}_{\ell_o}(\varepsilon_2)) \vee (\neg\mathsf{obsEv}^{g_c}_{\ell_o}(\varepsilon_1) \wedge \neg\mathsf{obsEv}^{g_c}_{\ell_o}(\varepsilon_2))$$

where $\varepsilon_i \vdash g_i \mathrel{\widetilde{\preccurlyeq}} g_c$.

(2) The stores must be related for k steps under store typing Σ, notation $\Sigma \vdash \mu_1 \approx^{k}_{\ell_o} \mu_2$. This means that, for locations that are common to both stores,[12] the stored values are related at $j < k$ steps. Formally:

$$\Sigma \vdash \mu_1 \approx^{k}_{\ell_o} \mu_2 \iff \forall g_c, \hat{g}_i, \varepsilon_i \vdash g_i \mathrel{\widetilde{\preccurlyeq}} g_c, g_c \vdash \hat{g}_1 \approx_{\ell_o} \hat{g}_2, j < k, \Sigma \vdash \mu_i,$$
$$\forall o \in \mathit{dom}(\mu_1) \cap \mathit{dom}(\mu_2), \Sigma; g_c \vdash \langle \hat{g}_1, \mu_1(o), \mu_1\rangle \approx^{j}_{\ell_o} \langle \hat{g}_2, \mu_2(o), \mu_2\rangle : \Sigma(U)$$

In particular, stored values must be related at *all* related security effects $\hat{g}_1, \hat{g}_2$. This generality is necessary because all reference operations involve stamping the current security effect (and its evidence) onto the stored value, and doing so must preserve relatedness. For instance, two runs of a program can update a store location with different values under a high-security effect because both will be stamped high-security, and thus indistinguishable by a low-security observer ℓ_o.

(3) The values must both have the same type U under an empty type environment and valid store type.

(4) The values must be either both observable or both not observable. If the values are not observable, they are deemed equivalent. If they are observable, then they must be related at their specific type, as specified by the auxiliary relation $\mathsf{obsRel}^{\Sigma; g_c U}_{k, \ell_o}$, defined by case analysis on U. If U is either Bool_g, Unit_g or $\mathsf{Ref}_g\ U'$, two values are related simply if their *raw values*

[12] For simplicity and without loss of generality, like Austin and Flanagan [2009], we assume that a new reference in two related executions is allocated at the same address.

$$\Sigma; g_c \vdash \langle \hat{g}_1, v_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \hat{g}_2, v_2, \mu_2 \rangle : U \iff g_c \vdash \hat{g}_1 \approx_{\ell_o} \hat{g}_2 \wedge \Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2 \wedge \ \cdot; \Sigma; \hat{g}_i \vdash v_i : U \wedge$$
$$(\mathsf{obsVal}^U_{\ell_o}(v_i) \vee \neg \mathsf{obsVal}^U_{\ell_o}(v_i)) \wedge \left((\mathsf{obsVal}^U_{\ell_o}(v_i) \wedge \mathsf{obsEv}^{g'_i}_{\ell_o}(\varepsilon_i)) \implies \mathsf{obsRel}^{\Sigma, g_c, U}_{k, \ell_o}(\hat{g}_1, v_1, \mu_1, \hat{g}_2, v_2, \mu_2)\right)$$

$$\mathsf{obsRel}^{\Sigma, g_c, U}_{k, \ell_o}(\hat{g}_1, v_1, \mu_1, \hat{g}_2, v_2, \mu_2) \iff \mathit{rval}(v_1) = \mathit{rval}(v_2) \qquad \text{if } U \in \{\mathsf{Bool}_g, \mathsf{Unit}_g, \mathsf{Ref}_g\ U'\}$$

$$\mathsf{obsRel}^{\Sigma, g_c, U_1 \xrightarrow{g_{32}}_{g_{31}} U_2}_{k, \ell_o}(\hat{g}_1, v_1, \mu_1, \hat{g}_2, v_2, \mu_2) \iff \forall j \le k, \forall U' = U'_1 \xrightarrow{g'_{32}}_{g'_{31}} U'_2, \forall U''_1,$$
$$\forall g'_c, \forall \hat{g}'_i = \varepsilon'_i g'_i, \text{ where } \varepsilon'_i \vdash g'_i \mathrel{\widetilde{\preccurlyeq}} g'_c, \text{ s.t. } \hat{g}_i \le_{\ell_o} \hat{g}'_i,$$
$$\varepsilon_{11} \vdash U_1 \xrightarrow{g_{32}}_{g_{31}} U_2 \lesssim U', \varepsilon_{12} \vdash U''_1 \lesssim U'_1, \text{ and } \varepsilon_{3i} \vdash \widetilde{g'_c \vee g'_{31} \preccurlyeq g'_{32}}, \text{ we have:}$$
$$\forall v'_i, \mu'_i, \Sigma', \Sigma \subseteq \Sigma', \Sigma'; g_c \vdash \langle \hat{g}_1, v'_1, \mu'_1 \rangle \approx^j_{\ell_o} \langle \hat{g}_2, v'_2, \mu'_2 \rangle : U''_1, \mathit{dom}(\mu_i) \subseteq \mathit{dom}(\mu'_i),$$
$$\Sigma'; g_c \vdash \langle \hat{g}_1, (\varepsilon_{11} v_1 @_{\varepsilon_{31}} \varepsilon_{12} v'_1), \mu'_1 \rangle \approx^j_{\ell_o} \langle \hat{g}_2, (\varepsilon_{11} v_2 @_{\varepsilon_{32}} \varepsilon_{12} v'_2), \mu'_2 \rangle : C(U'_2 \mathrel{\widetilde{\vee}} g'_{31})$$

Fig. 9. Related values

$$\Sigma; g_c \vdash \langle \hat{g}_1, t_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \hat{g}_2, t_2, \mu_2 \rangle : C(U) \iff g_c \vdash \hat{g}_1 \approx_{\ell_o} \hat{g}_2 \wedge \Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2 \wedge \forall \hat{g}'_i, \text{ s.t. } \hat{g}_i \le_{\ell_o} \hat{g}'_i \text{ and}$$
$$\cdot; \Sigma; \hat{g}'_i \vdash t_i : U, \forall j < k, \left(t_i \mid \mu_i \overset{\hat{g}'_i}{\longmapsto}{}^j t'_i \mid \mu'_i \implies \exists \Sigma', \Sigma \subseteq \Sigma'\right.$$
$$\left.\Sigma' \vdash \mu'_1 \approx^{k-j}_{\ell_o} \mu'_2 \wedge ((\mathit{irred}(t'_1) \wedge \mathit{irred}(t'_2)) \implies \Sigma'; g_c \vdash \langle \hat{g}_1, t'_1, \mu'_1 \rangle \approx^{k-j}_{\ell_o} \langle \hat{g}_2, t'_2, \mu'_2 \rangle : U)\right)$$

Fig. 10. Related computations

are equal (*rval* strips away checking-related information such as labels and evidences). Two functions are related if their application to two related argument values, in related stores, for $j \le k$ steps, are *related computations*, as explained below.

The definition of *related computations* is presented in Fig. 10. First, two triples of security effect, term, and store are related computations for k steps at type U if the security effects and the stores are related, as defined previously. Second, the terms must have type U under any *observationally higher* security effect $\hat{g}'$.[13] We say $\hat{g}' = \varepsilon' g'$ is observationally higher than $\hat{g} = \varepsilon g$, notation $\hat{g} \le_{\ell_o} \hat{g}'$ if $\neg \mathsf{obsEv}^{g_c}_{\ell_o}(\varepsilon) \Rightarrow \neg \mathsf{obsEv}^{g'_c}_{\ell_o}(\varepsilon')$, where $\varepsilon \vdash g \mathrel{\widetilde{\preccurlyeq}} g_c$ and $\varepsilon' \vdash g' \mathrel{\widetilde{\preccurlyeq}} g'_c$. For instance, in the static language it is the case that for any ℓ, $\mathsf{H} \le_{\ell_o} \mathsf{H} \vee \ell$, because by monotonicity of the join $\mathsf{H} \not\preccurlyeq \ell_o \Rightarrow \mathsf{H} \vee \ell \not\preccurlyeq \ell_o$. Additionally, for any $j < k$, if both terms can be reduced for at least j steps under security effect $\hat{g}'_i$, then the resulting stores should be related for the remaining $k - j$ steps. Finally, if the resulting terms are irreducible, they must be related values for the remaining $k - j$ steps at type U, as defined previously. The logical relation relates computations that do not terminate as long as the stores are also related after k steps.

Noninterference. Armed with these logical relations, we can state a semantics-driven notion of noninterference, and prove that well-typed terms of the internal language are sound with respect to it. The judgment $\Gamma; \Sigma; \hat{g} \models t : U$ says that term t is *semantically well-typed*, meaning that it respects the security protocol U for all observers, substitutions, stores, and steps [Ahmed 2004].

[13] This requirement is motivated by the proof, in order to obtain a stronger induction hypothesis [Toro et al. 2018].

Definition 5.3 (Semantic Security Typing).

$$\Gamma;\Sigma;\hat{g} \models t : U \iff \begin{array}{l}\forall\, \ell_o \in \textsc{Label}, k \geq 0, \rho_1, \rho_2 \in \textsc{Subst} \text{ and } \mu_1, \mu_2 \in \textsc{Store}, \forall g_c, \hat{g} = \varepsilon g, \\ \varepsilon \vdash g \mathrel{\widetilde{\preccurlyeq}} g_c, \text{ such that } \Sigma \vdash \mu_i \text{ and } \Gamma;\Sigma;g_c \vdash \langle \hat{g}, \rho_1, \mu_1\rangle \approx^k_{\ell_o} \langle \hat{g}, \rho_2, \mu_2\rangle, \\ \text{we have} \Sigma;g_c \vdash \langle \hat{g}, \rho_1(t), \mu_1\rangle \approx^k_{\ell_o} \langle \hat{g}, \rho_2(t), \mu_2\rangle : C(U)\end{array}$$

The definition above appeals to a notion of related substitutions. Indeed, the term t may have free variables, indicating "input parameters". The term is semantically well-typed if applying related substitutions (and stores) yields related computations at type U, for any number of steps k, and for any observer ℓ_o. Two substitutions are related if they map each variable in the term to related closed values:

Definition 5.4 (Related substitutions). Tuples $\langle \hat{g}_1, \rho_1, \mu_1\rangle$ and $\langle \hat{g}_2, \rho_2, \mu_2\rangle$ are related on k steps under Γ, Σ and g_c, notation $\Gamma;\Sigma;g_c \vdash \langle \hat{g}_1, \rho_1, \mu_1\rangle \approx^k_{\ell_o} \langle \hat{g}_2, \rho_2, \mu_2\rangle$, if $\rho_i \models \Gamma$, $\Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2$ and

$$\forall x \in dom(\Gamma).\Sigma;g_c \vdash \langle \hat{g}_1, \rho_1(x), \mu_1\rangle \approx^k_{\ell_o} \langle \hat{g}_2, \rho_2(x), \mu_2\rangle : \Gamma(x)$$

Note that because a low-security observer equates *all* high-security values, the actual substitutions and stores can be wildly different, up to the strictures that the logical relation imposes on their types.

Finally, Security Type Soundness says that the syntactic type system enforces noninterference.

Proposition 5.5 (Security Type Soundness). $\Gamma;\Sigma;\hat{g} \vdash t : U \implies \Gamma;\Sigma;\hat{g} \models t : U$

6 DERIVING GSL_{Ref} WITH AGT (ALMOST)

So far the presentation of GSL_{Ref} has focused on describing the language as it is and its properties, without explaining *how* it came to be designed that way. Several definitions in both the static and dynamic semantics may seem to come out of nowhere, and hard to accept without further justification.

This work originated in part from our desire to apply the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016] in a challenging setting. Indeed, AGT has been shown to be effective in different contexts: records and subtyping [Garcia et al. 2016], static semantics of gradual effects [Bañados Schwerter et al. 2014, 2016], gradual unions [Toro and Tanter 2017], as well as refinement types [Lehmann and Tanter 2017] and set-theoretic types [Castagna and Lanvin 2017]. But AGT has never been applied to a type discipline that denotes a relational property over multiple executions.

Therefore, we have systematically derived GSL_{Ref} from SSL_{Ref} using AGT. This methodology, which starts from considering gradual types as *abstractions* of static types, drove the entire design of GSL_{Ref}. The abstract interpretation framework of AGT provides *definitions*—semantically-defined notions—which may be hard to implement directly. From these definitions, we devise equivalent algorithmic *characterizations*—easily implementable, but hard to convincingly justify informally. AGT also explains how to derive the dynamic semantics of a gradual language based on the type safety argument of the static language. In Sec. 4 we try to convey guiding intuitions, but in this section we show how the definitions are not driven by intuition, but rather formally justified by AGT. Each algorithmic characterization from Sec. 4 is *equivalent* to its semantic *definition*, obtained using AGT and presented hereafter. These equivalences are proven in the companion technical report [Toro et al. 2018].

Before diving into the subtleties of applying AGT to security typing, we quickly describe the main elements of the AGT approach as spelled out by Garcia et al. [2016]: its inputs, steps, and outputs.

AGT in a nutshell. The AGT methodology proposes to derive the static and dynamic semantics of a gradual language in the following manner:

(1) **Deriving the statics.**
  (a) Start from a language with a fully-static typing discipline, including the particulars of its type safety proof.
  (b) Define the syntax of gradual types, and give them meaning via a concretization function, which maps gradual types to sets of static types; then define the corresponding most precise abstraction function, forming a Galois connection.
  (c) Lift type predicates and functions used in the type system of the static language through the Galois connection to obtain the gradual type system.

(2) **Deriving the dynamics.**
  (a) Define the structure of *evidence* for consistent judgments, which represents justification for why such a judgment holds; this representation depends on a Galois connection—usually the same as the one used for deriving the static semantics.
  (b) Reduce gradual programs by reducing *gradual typing derivations* decorated with evidence, mirroring reasoning steps of the static language's type safety proof, hence exploiting the correspondence between proof normalization and term reduction [Howard 1980].

Therefore, the "inputs" to AGT are only the static language, and the Galois connection(s) that give meaning to gradual types and evidences. As "output", one obtains the static and dynamic semantics of the gradual language, together with the guarantee that it is type safe, is a conservative extension of the static discipline, and satisfies the gradual guarantees.

Note that, as alluded to above, in order to achieve an implementation one must also provide algorithmic characterizations of the operators obtained through the abstract interpretation framework. Often these algorithms can be calculated by induction on types, but sometimes it requires trial-and-error. In any case, the AI-based definition provides the baseline against which to formally validate such characterizations.

Applying AGT to security typing. As mentioned above, applying AGT ensures by construction that the derived gradual language is type safe and satisfies the gradual guarantees. In prior work, we applied AGT to a *pure* language with security typing, and found the resulting language to satisfy noninterference [Garcia and Tanter 2015]. However, in this work, where the languages support mutable references, applying AGT to SSL_{Ref} yielded a gradual language that violates noninterference! By applying AGT, we surely obtained a gradual language that was type safe and satisfied the gradual guarantees, but unfortunately, the crucial semantic property of security types was broken. In brief, we had to apply two refinements. The first was proposed in the AGT methodology, though not needed in prior work. The second is novel, but conflicts with the dynamic gradual guarantee.

This section reports on these wrinkles and refinements so that future efforts to apply AGT to rich type disciplines can build on our experience. In particular:

- Sec. 6.1 sets up the basics to derive the static semantics of GSL_{Ref} with AGT, which was a successful endeavor. In the process, we identified one subtlety (about compositional lifting) that is worth highlighting.
- Sec. 6.2 explains the AGT approach to deriving the dynamic semantics of the gradual language. Here, we discover that evidence must use a more precise abstraction than the one used in the static semantics. While this possibility is briefly mentioned in [Garcia et al. 2016], it was not necessary in other applications of AGT.

- Sec. 6.3 discusses a crucial point related to enforcing noninterference in the presence of references, and hence potential implicit flows. This observation led us to add an extra check to GSL_{Ref}'s dynamic semantics. The check ensures noninterference, but breaks the dynamic gradual guarantee.

6.1 Deriving the Statics

Following the AGT approach, we give meaning to gradual security labels directly in terms of the original static security labels. The driving intuition is that the unknown label ? represents any label whatsoever, while a gradual label ℓ represents a single static security label. We formalize this with a *concretization* function.

Definition 6.1 (Label Concretization). $\gamma : \textsc{GLabel} \to \mathcal{P}(\textsc{Label})$

$$\gamma(\ell) = \{\, \ell \,\}$$
$$\gamma(?) = \textsc{Label}$$

Concretization immediately induces the notion of *precision*, which orders the static information content of gradual labels from most to least:

Definition 6.2 (Label Precision). $g_1 \sqsubseteq g_2$ if and only if $\gamma(g_1) \subseteq \gamma(g_2)$.

In order to exploit AGT to gradualize SSL_{Ref}, we also require an *abstraction* function to precisely summarize a set of static labels as a single gradual label (round hats $\widehat{x}$ denote sets of x):

Definition 6.3 (Label Abstraction). $\alpha : \mathcal{P}(\textsc{Label}) \rightharpoonup \textsc{GLabel}$:

$$\alpha(\{\, \ell \,\}) = \ell$$
$$\alpha(\emptyset) \text{ is undefined}$$
$$\alpha(\widehat{\ell}) = \ ? \text{ otherwise}$$

The γ and α functions are tightly connected by two properties that together form a Galois connection [Cousot and Cousot 1977].

Proposition 6.4 (α is Sound and Optimal). *If* $\widehat{\ell} \neq \emptyset$ *then,*

(i) $\widehat{\ell} \subseteq \gamma(\alpha(\widehat{\ell}))$.

(ii) If $\widehat{\ell} \subseteq \gamma(g)$ *then* $\alpha(\widehat{\ell}) \sqsubseteq g$.

Soundness (*i*) means that α always produces a gradual label whose concretization over-approximates the original set. Optimality (*ii*) means that α always yields the best (*i.e.* least) sound approximation that gradual labels can represent.

The meaning of gradual security types is derived from the meaning of gradual security labels. Therefore, we naturally define a Galois connection for gradual security types (see Appendix A.3).

Lifting predicates and functions. Following AGT, we exploit the Galois connections to *lift* all predicates and functions over labels and types from SSL_{Ref} to obtain the definition of their counterparts in GSL_{Ref}. In essence, each gradual entity (label, type) represents some set of static entities, so a consistent predicate holds among gradual entities so long as the underlying static predicate could *plausibly* hold. For instance, consistent ordering on gradual labels is defined as follows:

Definition 6.5 (Consistent label ordering). $g_1 \mathrel{\widetilde{\preccurlyeq}} g_2 \iff \ell_1 \preccurlyeq \ell_2$ for some $(\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2)$.

Consistent ordering conservatively extends static label ordering because each static label, when treated as a gradual label, concretizes to a singleton set that contains only itself; conservative extension is central to the concept of graduality [Siek et al. 2015]. On the other hand, consistent ordering holds universally for the unknown label ?, since it concretizes to all possible static labels.

Similarly, the join of two gradual labels is defined by lifting static label join:

Definition 6.6 (Gradual label join). $g_1 \mathbin{\widetilde{\vee}} g_2 = \alpha(\{\, \ell_1 \vee \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \,\})$

The gradual join of two gradual labels is the best abstraction of the set of all plausible static joins. For more insight, recall its equational characterization in Sec. 4: the unknown label disappears when joined with $\top$, while it otherwise survives all joins. This is an emergent property of lifting: we did not anticipate it.

Compositional vs. aggregate lifting. One unanticipated subtlety observed in Sec. 4 involves the compound premises of the (Sapp) and (Sref) rules, such as $\ell_c \vee \ell \preccurlyeq \ell'$. One might be tempted to lift this premise compositionally as $g_c \mathbin{\widetilde{\vee}} g \mathbin{\widetilde{\preccurlyeq}} g'$. But Garcia et al. [2016] explicitly warn against blindly lifting static predicates compositionally: compositional lifting must be proven (for instance, they show that lifting their subtyping premises compositionally yields the same result as lifting them aggregately). Here it matters! Consider the definition induced by AGT:

Definition 6.7 (Consistent bounding).

$$\widetilde{g_1 \vee g_2 \preccurlyeq g_3} \iff \ell_1 \vee \ell_2 \preccurlyeq \ell_3 \text{ for some } (\ell_1, \ell_2, \ell_3) \in \gamma(g_1) \times \gamma(g_2) \times \gamma(g_3)$$

This definition is *not* equivalent to compositional lifting. For instance, the relation $\mathsf{H} \mathbin{\widetilde{\vee}} ? \mathbin{\widetilde{\preccurlyeq}} \mathsf{L}$ holds, but we know that no static label ℓ satisfies $\mathsf{H} \vee \ell \preccurlyeq \mathsf{L}$ (because $\mathsf{H} \vee \ell$ must be at least as high as H).[14] In fact, precise lifting becomes critical when we reason about combining such lattice relations in the dynamic semantics. To the best of our knowledge, this is the first instance of aggregate lifting affecting the application of AGT.

6.2 Deriving the Dynamics

Garcia et al. [2016] derive the dynamic semantics of a gradual language by reduction of *gradual typing derivations* (augmented with evidence), thereby exploiting the correspondence between proof normalization and term reduction [Howard 1980]. This approach, which directly exploits the proof of syntactic type safety for the static language ($\mathrm{SSL}_{\mathsf{Ref}}$ in our case), provides the direct runtime semantics of gradual programs, instead of the usual approach by translation to some internal cast calculus [Siek and Taha 2006].

Since writing down reduction rules over (two-dimensional) derivation trees is unwieldy, Garcia et al. [2016] use intrinsically-typed terms [Church 1940] as a convenient flat notation for derivation trees. Intrinsic terms are heavy notationally because they carry all type annotations, yielding to reduction rules that are hard to read. To alleviate this burden, we have chosen to present the dynamic semantics by reducing *evidence-augmented terms*, which are more lightweight notationally, and establish a more direct connection with the traditional translational approach. The counterpart of this choice is that we had to present a translation from source $\mathrm{GSL}_{\mathsf{Ref}}$ terms to evidence-augmented $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms. Apart from this cosmetic difference, the central approach to reduction is the same: evidence is combined during reduction, producing either new evidence to support the plausibility of the contractum, or a runtime error if *no evidence remains, thereby refuting type safety.*

[14] To be honest, despite the warning of Garcia *et al.*, we first overlooked the issue and applied compositional lifting, assuming it would hold. We then observed that the resulting design loses enough precision to miss some evident inconsistencies, with dramatic consequences for security.

In essence, $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms are intrinsic terms from which computationally irrelevant static annotations have been erased. Proofs of theorems about $\mathrm{GSL}_{\mathsf{Ref}}$'s dynamic semantics need these annotations, so they use intrinsic terms. The companion technical report formalizes the relationship between intrinsic terms and evidence-augmented terms by giving a translation from intrinsic terms to evidence-augmented terms [Toro et al. 2018]. We show that, intrinsic terms can always be erased to $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms, and that the process can be reversed for well-typed $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms. Furthermore, related intrinsic and $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms either reduce to related terms or yield errors. Therefore the theorems about intrinsic terms transfer to $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ terms.

Reduction and consistent deductions. All instances of combining evidence in the reduction rules are dictated by $\mathrm{SSL}_{\mathsf{Ref}}$'s type safety proof. To illustrate this deep connection, we now analyze a case of the $\mathrm{SSL}_{\mathsf{Ref}}$ type safety proof and describe how to lift the argument to $\mathrm{GSL}_{\mathsf{Ref}}$. Consider the assignment case of $\mathrm{SSL}_{\mathsf{Ref}}$'s preservation proof, which in essence reduces a type derivation $\mathscr{D}$ to a new one and updates the program counter ℓ_c and store μ.

$$\mathscr{D} = \dfrac{\dfrac{o : S \in \Sigma}{\cdot; \Sigma; \ell_c \vdash o_\ell : \mathsf{Ref}_\ell\, S} \quad \mathscr{D}_1 \quad \cdot; \Sigma; \ell_c \vdash v : S_2 \quad S_2 <: S \quad \ell_c \curlyvee \ell \preccurlyeq \mathsf{label}(S)}{\cdot; \Sigma; \ell_c \vdash o_\ell{:=}v : \mathsf{Unit}_\bot}$$

The relevant reduction rule (Fig. 2) follows:

$$o_\ell{:=}v \mid \mu \xrightarrow{\ell'_c} \mathsf{unit}_\bot \mid \mu[o \mapsto v \curlyvee \ell'_c \curlyvee \ell].$$

The fact that $\mathscr{D}$ reduces to $\dfrac{}{\cdot; \Sigma; \ell_c \vdash \mathsf{unit}_\bot : \mathsf{Unit}_\bot}$ is immediate, but we must also prove that the stored value $v \curlyvee \ell'_c \curlyvee \ell$ respects the store type, *i.e.* $S_2 \curlyvee \ell'_c \curlyvee \ell <: S$. Since $\cdot; \Sigma; \ell_c \vdash v : S_2$ and $S_2 <: S$, it suffices to show that $\ell'_c \curlyvee \ell \preccurlyeq \mathsf{label}(S)$. We do so as follows. Since $\curlyvee$ is monotone with respect to $\preccurlyeq$ in both arguments, we can combine $\ell'_c \preccurlyeq \ell_c$ (assumed in the statement of preservation) and $\ell \preccurlyeq \ell$ (deduced by $\preccurlyeq$ reflexivity) to deduce $\ell'_c \curlyvee \ell \preccurlyeq \ell_c \curlyvee \ell$. Finally, since $\preccurlyeq$ is transitive, we combine the above with the $\ell_c \curlyvee \ell \preccurlyeq \mathsf{label}(S)$ to deduce $\ell'_c \curlyvee \ell \preccurlyeq \mathsf{label}(S)$. To recap, this "reduction" applies reasoning steps with a computational flavor: it composes $\preccurlyeq$ relations to deduce new ones, using both *join monotonicity* and *order transitivity*.

In the gradual setting, transitivity of ordering of gradual labels does not always hold: *e.g.* $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ and $? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$ but $\mathsf{H} \not\mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$. As such, transitivity of consistent ordering is *plausible* but not *definite*, so we have to check. How? Here is the key intuition: recall that a consistent judgment like $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ means that $\ell_1 \preccurlyeq \ell_2$ holds for *some* pair of labels (ℓ_1, ℓ_2) drawn from the concretizations $\gamma(\mathsf{H}) = \{\, \mathsf{H} \,\}$ and $\gamma(?) = \textsc{Label}$ respectively. We do not know *which* pair, so we must consider all *plausible* ones, *i.e.* $\{\, (\mathsf{H}, \mathsf{H}), (\mathsf{H}, \top) \,\}$: the rest are surely wrong so we discard them. Similarly, the plausible pairs for $? \mathrel{\widetilde{\preccurlyeq}} \top$ are $\{\, (\ell, \top) \mid \ell \preccurlyeq \top \,\}$. Now, given these two sets of plausible orderings, is *transitivity* plausible? Yes, because two plausible deductions arise: 1) $\mathsf{H} \preccurlyeq \mathsf{H}$ and $\mathsf{H} \preccurlyeq \top$ implies $\mathsf{H} \preccurlyeq \top$; and 2) $\mathsf{H} \preccurlyeq \top$ and $\top \preccurlyeq \top$ implies $\mathsf{H} \preccurlyeq \top$. When collected, the deduced pairings collapse to the singular expected result: $\{\, (H, \top) \,\}$. If we replay the same reasoning for $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ and $? \mathrel{\widetilde{\preccurlyeq}} \mathsf{L}$, however, we deduce $\emptyset$, which means that transitivity is *not* plausible: it has been refuted. An analogous process applies for join monotonicity, as well as transitivity of consistent subtyping, yielding sets of pairs of candidate subtypings.

In both of the above deductions, we reason imprecisely yet still deduce definite results: a single possibility in one, and none in the other. But in general, imprecision begets imprecision. The main source of complication is that static safety arguments deduce ordering relationships by interleaving transitivity and monotonicity arguments, so corresponding consistent deductions must mirror them. Furthermore, it would be especially burdensome to explicitly track sets of pairs of labels at runtime, let alone the sets of pairs of types that arise when reasoning about consistent

subtyping. This is where AGT suggests to use an *abstraction* of the possible static candidates, evidence. Evidence of a consistent judgment is a pair of abstractions of sets of static entities that justify a consistent judgment. Which abstraction to use turns out to be a crucial decision in order to preserve noninterference, as discussed next.

Problems with evidence as gradual labels. The "natural" abstraction of sets of labels are gradual labels, as used in the static semantics. In fact, Garcia et al. [2016] use the same abstraction to represent both runtime evidence and static gradual types; we initially followed suit. However, the first major subtlety we uncovered while deriving GSL_{Ref}'s dynamic semantics is that using gradual labels (and consequently, gradual types) for evidence yields a design that achieves both type safety and the gradual criteria, but violates noninterference!

This problem manifested in two parts of the noninterference proof. First, the noninterference proof relies on the *associativity* of consistent transitivity.[15] However, consistent transitivity of label ordering is not associative if gradual labels are used to represent evidence. Recall the program $\text{true}_? :: \text{Bool}_\text{H} :: \text{Bool}_? :: \text{Bool}_\text{L}$, introduced in Sec. 4.2, which we expect to fail at runtime, and which ultimately involves combining three consistent label ordering judgments: $\varepsilon_1 \vdash ? \mathrel{\widetilde{\preccurlyeq}} \text{H}$, $\varepsilon_2 \vdash \text{H} \mathrel{\widetilde{\preccurlyeq}} ?$, and $\varepsilon_3 \vdash ? \mathrel{\widetilde{\preccurlyeq}} \text{L}$. If we use a pair of gradual labels to represent evidence, eventually we have to calculate $(\varepsilon_1 \circ^{<:} \varepsilon_2) \circ^{<:} \varepsilon_3$. But $\varepsilon_1 = \langle ?, \text{H}\rangle$, $\varepsilon_2 = \langle \text{H}, ?\rangle$, and $\varepsilon_3 = \langle ?, \text{L}\rangle$, then $\varepsilon_1 \circ^{\preccurlyeq} \varepsilon_2 = \langle ?, ?\rangle$ and $\langle ?, ?\rangle \circ^{\preccurlyeq} \varepsilon_3 = \langle ?, \text{L}\rangle$, so no runtime error is produced. Note that $\varepsilon_1 \circ^{<:} (\varepsilon_2 \circ^{<:} \varepsilon_3)$ fails as expected, because $\varepsilon_2 \circ^{<:} \varepsilon_3$ is not defined, but this is not the composition order that arises at runtime.

Second, the proof of noninterference relies on the *observational completeness* of the consistent join operator:

Lemma 6.8. *Suppose* $\varepsilon_1 \vdash g_1' \mathrel{\widetilde{\preccurlyeq}} g_1$ *and* $\varepsilon_2 \vdash g_2' \mathrel{\widetilde{\preccurlyeq}} g_2$ *such that* $\varepsilon_1 \mathbin{\widetilde{\curlyvee}} \varepsilon_2 \vdash \overbrace{g_1' \curlyvee g_2'} \preccurlyeq g_1 \curlyvee g_2$.
Then $(\neg\text{obsEv}^{g_1}_{\ell_o}(\varepsilon_1) \vee \neg\text{obsEv}^{g_2}_{\ell_o}(\varepsilon_2)) \iff \neg\text{obsEv}^{g_1 \widetilde{\curlyvee} g_2}_{\ell_o}(\varepsilon_1 \mathbin{\widetilde{\curlyvee}} \varepsilon_2)$.

The analogous static lemma, *i.e.* $(\neg\text{obsEv}^{\ell_1}_{\ell_o}(\ell_1) \vee \neg\text{obsEv}^{\ell_2}_{\ell_o}(\ell_2)) \iff \neg\text{obsEv}^{\ell_1 \curlyvee \ell_2}_{\ell_o}(\ell_1 \curlyvee \ell_2)$, holds trivially by the very definition of the join, but this property fails to hold in the presence of the unknown label. Suppose $\varepsilon_1' \vdash \text{H} \mathrel{\widetilde{\preccurlyeq}} ?$ and $\varepsilon_2' \vdash ? \mathrel{\widetilde{\preccurlyeq}} ?$. If we use a pair of gradual labels to represent evidence, then $\varepsilon_1' = \langle \text{H}, ?\rangle$, $\varepsilon_2' = \langle ?, ?\rangle$, and $\varepsilon_1' \mathbin{\widetilde{\curlyvee}} \varepsilon_2' = \langle ?, ?\rangle$ losing information about H . But $\neg\text{obsEv}^{?}_{\text{L}}(\langle \text{H}, ?\rangle)$ and $\text{obsEv}^{?}_{\text{L}}(\langle ?, ?\rangle)$, therefore invalidating the lemma.

Representing evidence as intervals. These observations forced us to seek a more precise abstraction whose composition (through consistent transitivity) is associative and preserves the observational completeness of consistent join. Since it suffices to know whether the upper- and lower-bounds of the plausible static labels overlap to deduce the plausibility of consistent ordering, *intervals* seem to be a fitting abstraction.[16] Indeed, this abstraction is sufficiently precise to guarantee the desired properties.

[15] Note that associativity of cast composition is also critical for space-efficient semantics of gradual typing, *e.g.* Siek and Wadler [2010]. We conjecture that associativity may be a fundamentally desirable property, and intend to pursue this question.

[16] One could design a gradual security language that uses label intervals instead of gradual labels right from the start, including in the static semantics. While this would unify the abstractions used in the statics and dynamics, it would yield a gradual type system that rejects more secure programs than GSL_{Ref} does. For instance, the program $(\text{if false}_\text{L} :: ? \text{ then } 1_\text{H} \text{ else } 2_\text{L}) :: \text{L}$, is accepted and runs without errors in GSL_{Ref}. But if we use intervals in the static semantics, then the security level of the conditional expression which boils down to the join between ?, H and L, would be [L, H], therefore the program would be rejected statically. Applying a ? ascription to 1_H would fix this program.

Definition 6.9 (Interval Concretization). $\gamma_\iota : \textsc{Interval} \to \mathcal{P}(\textsc{Label})$, where $\textsc{Interval} = \{[\ell_1, \ell_2] \in \textsc{Label}^2 \mid \ell_1 \preccurlyeq \ell_2\}$

$$\gamma_\iota([\ell_1, \ell_2]) = \{\ell \mid \ell \in \textsc{Label}, \ell_1 \preccurlyeq \ell \preccurlyeq \ell_2\}.$$

Definition 6.10 (Interval Abstraction). $\alpha_\iota : \mathcal{P}(\textsc{Label}) \to \textsc{Interval}$

$$\alpha_\iota(\emptyset) \text{ is undefined} \qquad \alpha_\iota(\{\overline{\ell_i}\}) = [\curlywedge\overline{\ell_i}, \curlyvee\overline{\ell_i}] \text{ otherwise}$$

With evidence based on intervals, $(\varepsilon_1 \circ^{\preccurlyeq} \varepsilon_2) \circ^{\preccurlyeq} \varepsilon_3$ and $\varepsilon_1 \circ^{\preccurlyeq} (\varepsilon_2 \circ^{\preccurlyeq} \varepsilon_3)$ are equivalent. Back to the example, now $\varepsilon_1 = \langle[\bot, \mathsf{H}], [\mathsf{H}, \mathsf{H}]\rangle, \varepsilon_2 = \langle[\mathsf{H}, \mathsf{H}], [\mathsf{H}, \top]\rangle$ and $\varepsilon_3 = \langle[\bot, \mathsf{L}], [\mathsf{L}, \mathsf{L}]\rangle$, then $\varepsilon_1 \circ^{\preccurlyeq} \varepsilon_2 = \langle[\bot, \mathsf{H}], [\mathsf{H}, \top]\rangle$. Because $\langle[\bot, \mathsf{H}], [\mathsf{H}, \top]\rangle \circ^{\preccurlyeq} \varepsilon_3$ is undefined, a runtime error is raised, avoiding the breach of noninterference. Also, the observational-monotonicity of the join is preserved. Now $\varepsilon_1' = \langle[\mathsf{H}, \mathsf{H}], [\mathsf{H}, \top]\rangle$ and $\varepsilon_2' = \langle[\bot, \top], [\bot, \top]\rangle$, then $\varepsilon_1' \mathbin{\widetilde{\curlyvee}} \varepsilon_2' = \langle[\mathsf{H}, \top], [\mathsf{H}, \top]\rangle$ and now $\neg\mathsf{obsEv}^?_{\mathsf{L}}(\langle[\mathsf{H}, \top], [\mathsf{H}, \top]\rangle)$ as expected.

Lifting consistent lattice relations. We now explain how the definitions of consistent transitivity and join monotonicity are semantically justified. As discussed in Sec. 6.1, premises such as $\ell_c \curlyvee \ell \preccurlyeq \ell'$ must be lifted as aggregates. In fact, such a judgment is likely the consequence of similar deductions from earlier reduction steps. For instance ℓ must be some *lattice expression* $F(\overline{\ell_i})$ comprising joins (and meets) of source program labels $\overline{\ell_i}$. Therefore, to mirror static type safety reasoning steps at runtime, and catch inconsistencies if they arise, we must generalize each ordering premise in a derivation and consider it as some *lattice relation* $F_1(\overline{\ell_i}) \preccurlyeq F_2(\overline{\ell_j})$. The notion of evidence must consequently account for the plausibility of *consistent lattice relations*:

$$\langle \iota_1, \iota_2 \rangle \vdash \widetilde{F_1(\overline{g_i}) \preccurlyeq F_2(\overline{g_j})}$$

The definitions of consistent join monotonicity and consistent transitivity then follow directly from AGT by consistent lifting.

Definition 6.11 (Consistent transitivity for label ordering).

$$\circ^{\preccurlyeq} : \textsc{Interval}^2 \times \textsc{Interval}^2 \rightharpoonup \textsc{Interval}^2$$

$$\langle \iota_1, \iota_{21} \rangle \circ^{\preccurlyeq} \langle \iota_{22}, \iota_3 \rangle = \alpha_\iota^2(\{\langle \ell_1, \ell_3 \rangle \in \gamma_\iota^2(\langle \iota_1, \iota_3 \rangle) \mid \exists \ell \in \gamma_\iota(\iota_{21}) \cap \gamma_\iota(\iota_{22}).\ell_1 \preccurlyeq \ell \wedge \ell \preccurlyeq \ell_3\})$$

Consistent transitivity produces evidence for all plausible instances of consistent ordering that can be deduced using transitivity from the plausible instances of ordering represented by the two inputs. By design, $\alpha_\iota^2(\emptyset)$ is undefined, so consistent transitivity is also undefined if no plausible pairings remain to support a deduction.

Definition 6.12 (Consistent join monotonicity). $\widetilde{\curlyvee} : \textsc{Interval}^2 \times \textsc{Interval}^2 \rightharpoonup \textsc{Interval}^2$

$$\varepsilon_1 \widetilde{\curlyvee} \varepsilon_2 = \quad \alpha_\iota^2(\{\langle \ell_1, \ell_2 \rangle) \mid \exists \langle \ell_{11}, \ell_{12} \rangle \in \gamma_\iota^2(\varepsilon_1), \langle \ell_{21}, \ell_{22} \rangle \in \gamma_\iota^2(\varepsilon_2).\ell_1 = \ell_{11} \curlyvee \ell_{21},\ \ell_2 = \ell_{12} \curlyvee \ell_{22},\ \ell_1 \preccurlyeq \ell_2\})$$

Consistent join monotonicity is analogous, but note that due to lattice and interval properties, consistent join monotonicity is really a total function. Also, the $\ell_1 \preccurlyeq \ell_2$ condition is superfluous; we present the definition in this form to preserve the general structure of consistent deduction definitions.

The algorithmic characterizations from Sec. 4.2 are equivalent to the above definitions. More importantly, we can prove that these operators indeed yield valid evidence for the combined consistent judgments.

Proposition 6.13. *Suppose* $\varepsilon_1 \vdash \widetilde{F_{11}(\overline{g_i}) \preccurlyeq F_{12}(\overline{g_j})}$ *and* $\varepsilon_2 \vdash \widetilde{F_{21}(\overline{g_i}) \preccurlyeq F_{22}(\overline{g_j})}$
Then $\varepsilon_1 \mathbin{\widetilde{\curlyvee}} \varepsilon_2 \vdash \widetilde{F_{11}(\overline{g_i}) \curlyvee F_{21}(\overline{g_i}) \preccurlyeq F_{12}(\overline{g_j}) \curlyvee F_{22}(\overline{g_j})}$

Proposition 6.14. *Suppose* $\varepsilon_1 \vdash \widetilde{F_1(\overline{g_i}) \preccurlyeq F_2(\overline{g_j})}$ *and* $\varepsilon_2 \vdash \widetilde{F_2(\overline{g_j}) \preccurlyeq F_3(\overline{g_k})}$. *If* $\varepsilon_1 \circ^{\preccurlyeq} \varepsilon_2$ *is defined, then* $\varepsilon_1 \circ^{\preccurlyeq} \varepsilon_2 \vdash \widetilde{F_1(\overline{g_i}) \preccurlyeq F_3(\overline{g_k})}$

From labels to types. Finally, in addition to reasoning about consistent label ordering, the dynamic semantics must track and check the plausibility of consistent subtyping. Since (consistent) subtyping is induced by (consistent) ordering, the reasoning in question arises by lifting the same constructions to gradual security types, consistent subtyping, and consistent subtyping join and meet.

Just as we extend gradual labels g to gradual security types U (*e.g.* Int_g) in the source language, so do we extend label intervals ι point-wise to *type intervals* E (*e.g.* Int_ι) and corresponding notions of evidence for consistent subtyping ε (*e.g.* $\langle \mathsf{Int}_{\iota_1}, \mathsf{Int}_{\iota_2} \rangle$), which represent sets of pairs of candidates for plausible subtyping. We introduce evidence judgments $\varepsilon \vdash U_1 \lesssim U_2$ to associate runtime evidence with particular consistent subtyping judgments. The entire development mirrors the one for labels, and does not convey any new insights (see Appendix A.5).

6.3 Policing Dynamic Heap Updates

Although adopting label intervals for evidence of consistent label judgments addressed some aspects of the noninterference proof, this refinement alone is not sufficient.

To illustrate the remaining problem, recall the example of implicit flows from Sec. 2, in particular the second version of the example, which has some missing static annotations.

```
1  fun x: Bool_H =>
2    let y: Ref Bool_? = ref true_?
3    let z: Ref Bool_L = ref true_L
4    if x then y := false_? else unit
5    if !y then z := false_L else unit
6    !z
```

This program is accepted statically and also runs without errors: if x is true_H then the program reduces to true_L, and if x is $\mathsf{false}_\mathsf{H}$ it reduces to $\mathsf{false}_\mathsf{L}$: a clear breach of noninterference!

To understand the problem, consider what happens for the different values of x. When x is true_H the assignment in line 4 under security effect H is valid, because $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$. In that moment we know that the security level of the content of y, must be higher than H. But when x is $\mathsf{false}_\mathsf{H}$, in line 5 we assume that the security level of the content of y is lower than L. In other words, under supposedly-related executions we get contradictory evidence for y. Notice that in the assignment at line 4, the judgment $\mathsf{H} \mathrel{\widetilde{\preccurlyeq}} ?$ holds, but so does its negation $\mathsf{H} \not\mathrel{\widetilde{\preccurlyeq}} ?$. To preserve noninterference, we must ensure that its negation never holds.

To recover noninterference, we add an extra check to the assignment reduction rule ($r7$) from Fig. 6:

$$\varepsilon_1 o_g :=_{\varepsilon_3} \varepsilon_2 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \mathsf{unit}_\perp \mid \mu[o \mapsto \varepsilon'(u \mathbin{\widetilde{\vee}} (g_c \mathbin{\widetilde{\vee}} g))] \\ \textbf{error} \qquad \text{if } \varepsilon' \text{ is not defined, or } \varepsilon \lfloor\leq\rfloor \mathit{ilbl}(\varepsilon'') \text{ does not hold} \end{cases}$$

where $\mu(o) = \varepsilon'' u'$. The highlighted check ensures that if the security effect is not observable, then the content of the heap to be replaced must also be not observable.[17] This concept is formalized in the following lemma, which is used in the noninterference proof:

[17] This check is analogous to the no-sensitive-upgrade check introduced by Austin and Flanagan [2009], taken to the gradual context, and hence involving unknown labels, evidences and consistent judgments.

Lemma 6.15. *Consider* $\varepsilon_1 \vdash g_1' \mathrel{\widetilde{\preccurlyeq}} g_1$ *and* $\varepsilon_2 \vdash g_2' \mathrel{\widetilde{\preccurlyeq}} g_2$. *Then* $(\neg\mathsf{obsEv}^{g_1}_{\ell_o}(\varepsilon_1) \wedge \varepsilon_1 \lfloor\leq\rfloor \varepsilon_2) \Rightarrow \neg\mathsf{obsEv}^{g_2}_{\ell_o}(\varepsilon_2)$.

With the additional check, if x is true_H, the program fails at runtime, preserving noninterference.

The necessity of the check shows up in the noninterference proof for the if case. When two computations have related non-observable conditionals, the booleans can be different. This may lead to two related computations that reduce different branches under a high-security context. At that point, we must enforce that those different executions only write high-security values to the heap. In other words, as long as both executions reduce under high-security contexts, their executions can desynchronize only on private information. Formally, the following lemma should hold:

Lemma 6.16. *Consider* $\cdot; \Sigma; \varepsilon g_c \vdash t : U$, g_c' *and* μ *such that,* $\varepsilon \vdash g_c \mathrel{\widetilde{\preccurlyeq}} g_c'$, $\neg\mathsf{obsEv}^{g_c'}_{\ell_o}(\varepsilon)$ *and* $\Sigma \vdash \mu$, *and* $\forall k > 0$, *such that* $t \mid \mu \overset{\varepsilon g_c}{\longmapsto}{}^{k} t' \mid \mu'$,

(1) $\forall o \in dom(\mu') \setminus dom(\mu)$, $\neg\mathsf{obsVal}^{U}_{\ell_o}(\mu'(o))$.
(2) $\forall o \in dom(\mu') \cap dom(\mu)$ *where* $\mu'(o) \neq \mu(o)$,
  (a) $\neg\mathsf{obsVal}^{U}_{\ell_o}(\mu(o))$, *and*
  (b) $\neg\mathsf{obsVal}^{U}_{\ell_o}(\mu'(o))$.

Without the additional check in rule (*r*7), we cannot prove (2.a): before updating a reference, the current content should be non observable. And as we can see in the example above, without the check, the reference before the assignment would be observable, hence breaking the Lemma.

In its current formulation [Garcia et al. 2016], AGT derives the dynamic semantics of the gradual language from the *type safety* argument of the static language. Here, we are facing a typing discipline in which type safety *does not* imply type soundness (*i.e.* noninterference), and hence, the methodology falls short of naturally preserving that property. This suggests that extending AGT to ensure type soundness of the derived gradual language might require adapting the conceptual framework to take the purely static type soundness proof as a source of design insight.

Noninterference vs. Dynamic gradual guarantee. Although the extra check above allows $\mathsf{GSL}_\mathsf{Ref}$ to ensure noninterference, it sacrifices the dynamic gradual guarantee. Recall that this guarantee says that removing a static security annotation cannot introduce new runtime errors.

Consider the following example:

```
1  fun x: Bool_H =>
2    let y: Ref Bool_H = ref true_H
3    if x then y := false_H else unit
```

The program is accepted statically and runs without error as it does not break noninterference. If we remove the type annotations on line 2:

```
1  fun x: Bool_H =>
2    let y: Ref Bool_? = ref true_?
3    if x then y := false_H else unit
```

then the program is conservatively rejected at runtime, because of the additional check for assignments. This behavior violates the dynamic gradual guarantee.[18]

To sum up, if decreasing the precision of a type annotation results in performing an assignment to a reference whose content now has an unknown security label, and that assignment occurs under a non-public security effect, a runtime error can be raised, whereas the more precise program did

[18] Removing the additional check on assignments recovers the dynamic gradual guarantee, but it breaks noninterference: there is no free lunch in presence of mutable references.

not fail. More precisely, even in such situations, a runtime error will only be raised if the dynamic security information about the stored value up to the point of the actual assignment is lower than the current security effect. For instance, in our example above, if we modify the security level of the boolean in line 2 to H (leaving the type of y as it is), then the program performs a valid assignment on a reference whose content has a statically-unknown security level, but dynamically H; therefore no runtime error is raised. Unfortunately, beyond pure and read-only programs, it seems impossible to provide any useful *syntactic* characterization of the programs for which the dynamic gradual guarantee holds, because both the current security effect and the accumulated evidence about a given value are essentially dynamic information.

7 RELATED WORK

Static and dynamic information-flow control techniques have been extensively studied in the literature. The area is too vast to exhaustively review here: we refer to [Hedin and Sabelfeld 2012b; Russo and Sabelfeld 2010; Sabelfeld and Myers 2003] for broad overviews of the area. This section first focuses on security type systems, as well as some specific approaches to dynamic information flow control, given the static-to-dynamic spectrum that gradual security typing covers. We also discuss existing proposals that combine static and dynamic checking. Finally we relate our work to other efforts to gradualize advanced type disciplines.

Static information flow control. Volpano et al. [1996] present one of the first type systems for information flow analysis, developed for a first-order imperative language with conditionals and loops. They present and formalize the first soundness result for a security-typed language, namely that altering the initial values of locations cannot affect resulting values of locations with a lesser security level.

Subsequently, Heintze and Riecke [1998] present a security-typed higher-order language called the Secure Lambda Calculus (SLam). SLam is a functional language extended with sums, products, and recursion, that supports both confidentiality and its dual notion, integrity [Biba 1977]. They introduce the prot expression, which we also use, to increase the ambient security level for the dynamic extent of evaluating a term. The noninterference proof for SLam is also based on logical relations. The authors extend SLam with concurrency and references. They prove that the resulting language is type safe, but they do not prove noninterference, deemed too problematic in a concurrent setting. SSL_{Ref} is also a higher-order language with references, but it does not support sums, products, recursion and concurrency. We prove noninterference for both GSL_{Ref} and SSL_{Ref}. Extending GSL_{Ref} to richer types and concurrency is a challenge worth addressing in future work.

To consolidate different related efforts, Abadi et al. [1999] develop the Dependency Core Calculus (DCC), an extension of the lambda calculus that tracks dependencies such as security, partial evaluation, program slicing and call-tracking. In particular, they show that different languages such as SLam can be translated to DCC. They present a semantic model of DCC that helps to provide a simple proof of noninterference. It would be interesting to study the application of AGT to DCC, to provide a general account of gradual dependency tracking.

JFlow [Myers 1999; Myers and Liskov 1997], which later evolved into Jif [Myers and Liskov 2000], is a practical extension of the Java language that protects both confidentiality and integrity of sensitive data. Jif supports statically-checked information flow annotations, a decentralized label model with principals, automatic label inference, and security label polymorphism, all integrated with object-oriented features like class inheritance, as well as exceptions, among other features. Jif supports runtime label tests that can be used to encode explicit security casts, although such casts break type-based reasoning about noninterference. Scaling up GSL_{Ref} to cover the feature set of Jif would open the door to a practical implementation of gradual security typing.

Zdancewic [2002] proposes λ^{SEC}, a simple security language similar to SLam, and proves noninterference using logical relations. He then extends the language with references, yielding λ^{SEC}_{REF}, which was the starting point for our design of SSL_{Ref}. Unlike SSL_{Ref}, the operational semantics of λ^{SEC}_{REF} includes additional checks to control whether it is safe to assign to references; the type system then makes these checks redundant. In SSL_{Ref}, we omit these checks, and the runtime only *tracks* security levels. The runtime checks needed in the gradual setting arise as evidence combination. Also, Zdancewic does not prove noninterference for λ^{SEC}_{REF} directly, but instead by a CPS translation to a lower-level imperative language with explicit continuations, for which noninterference is established [Zdancewic and Myers 2001]. This setting permits studying information flow with concurrency and as such could be a judicious starting point to study the interaction of gradual security typing and concurrency.

Much work on static information flow analysis focuses on *declassification*, which is the limited, intentional, and controlled release of confidential information. Declassification is outside the scope of this work, though a very interesting perspective for future work; we refer to [Sabelfeld and Sands 2009] for an introductory survey.

An important distinction in information flow analysis is whether an analysis is *flow-sensitive*, *i.e.* whether memory cells are allowed to store values of different security levels at different times. Hunt and Sands [2006] explore families of sound flow-sensitive type systems, indexed by the choice of the security lattice. In particular, they show that every program typeable in a flow-sensitive static type system can be translated to an equivalent program typeable in a flow-insensitive type system. SSL_{Ref} is a flow-insensitive purely static analysis; GSL_{Ref} inherits flow-insensitivity for its static semantics. However, at runtime the security level of references is allowed to vary (through evidence composition) within the bounds imposed by the static type of the reference. This means that a reference that is created with an unknown security label can store values of any security level at different times. This leads us to sharing challenges faced by dynamic information-flow control techniques, discussed hereafter.

Dynamic information flow control. Russo and Sabelfeld [2010] show that static mechanisms can be more precise than dynamic ones about certain kinds of information flows. Indeed, noninterference can be characterized as a 2-safety property, meaning that it can only be refuted by observing two different executions of the same program with different inputs. This makes it particularly challenging for dynamic information flow control, which traditionally makes decisions based on a single execution. Most work on dynamic information flow analysis therefore monitors a 1-safety property that conservatively approximates noninterference, but has the advantage of being observable in a single execution. Such approximations necessarily introduce false alarms, especially when mutable references are involved.

To avoid implicit leaks through the heap in a purely dynamic information-flow analysis, Austin and Flanagan [2009] introduce a *no-sensitive-upgrade check* to prevent implicit security leaks through partially-leaked data, *i.e.* data produced from updates to public heap data that depend on private information. We adapt this approach to GSL_{Ref}, imposing an extra check when assigning to references. Subsequently, Austin and Flanagan [2010] propose a more permissive analysis, where partially-leaked data is allowed, but carefully tracked to ensure that it is upgraded before being used in conditional tests. This allows programmers to iteratively add security upgrades to partially-leak data only when needed, through multiple executions of a program.

Later, Austin and Flanagan [2012] introduce a completely different approach: *faceted execution*, which simulates multiple executions of a program for different security levels in a single run. A faceted execution yields a faceted value, which in a traditional two-point lattice is a pair of a public

and a private value. This novel approach enables a characterization of noninterference as a 1-safety property, without introducing false alarms. It does however raise questions regarding how to efficiently implement such faceted executions, especially in the presence of complex security lattices. Faceted execution was recently extended to support dynamic information flow with exceptions, declassification and clearance [Austin et al. 2017]. It would be interesting to explore whether basing GSL_{Ref} on faceted execution might yield a gradual security language that fully respects the dynamic gradual guarantee, by avoiding the extra runtime check in assignments.

Stefan et al. [2017] present a dynamic information-flow control system called LIO. Contrary to most approaches to dynamic information flow, LIO does not modify the underlying language runtime semantics, being implemented as a Haskell library. LIO supports both mutable references and exceptions. Exceptions are used to recover from security monitor failures, preserving both confidentiality and integrity. The possibility of securely recovering from runtime security exceptions is an interesting perspective to study in the context of gradual security typing. More generally, recovering from runtime type errors raises a number of questions about the metatheory of gradual typing, because doing so can directly affect the dynamic gradual guarantee as well as type-based reasoning (*e.g.* it becomes possible to encode explicit type tests).

Hybrid information flow control. To resolve the tension between flexibility and soundness of flow-sensitive analyses, Russo and Sabelfeld [2010] propose a general *hybrid* approach, in which a static effect analysis is used to dynamically upgrade the security level of variables of untaken branches of conditionals, thereby preventing implicit leaks through the heap. This hybrid approach is developed on top of a (first-order) imperative language. Moore and Chong [2011] later show how to implement this hybrid approach more efficiently using additional static analyses.

A variety of hybrid information-flow control systems have been investigated, whose designs combine static and dynamic techniques that buttress one another to balance permissiveness and efficiency. Note that although gradual typing also combines static and dynamic techniques, hybrid approaches differ essentially from gradual ones. The key specificity of gradual typing is to smoothly support the continuum between static and dynamic checking based on the (programmer-controlled) *precision* of type annotations [Siek and Taha 2006; Siek et al. 2015]. This central notion of type precision is absent from hybrid approaches, in which the balance between static and dynamic checking is often driven by other concerns—such as the (un)decidability of a static predicate [Knowles and Flanagan 2010], or the need to pre-compute information for enhancing runtime checking.

Chandra and Franz [2007] implement hybrid security information flow control for the Java Virtual Machine. The operational semantics permits policies to change during execution. To prevent invalid implicit flows through the heap, they perform a static analysis of effects similar to Russo and Sabelfeld [2010]. Information about conditionals is gathered ahead of execution, then used to update labels at runtime, as if all branching alternatives had been taken. They also statically determine when the current security effect can be lowered again after a conditional. Performing an effect analysis statically to drive runtime monitoring is appealing as it could obviate the extra assignment check in GSL_{Ref} that compromised the dynamic gradual guarantee. However, in the setting of a higher-order imperative language, the effect analysis could easily become too conservative or too demanding for programmers. Combining gradual security and gradual effects [Bañados Schwerter et al. 2016] may temper this issue, but represents a considerable challenge in itself.

Shroff et al. [2007] present a dynamic information flow system based on runtime tracking of indirect dependencies between program points, allowing a lazier, hence more flexible, detection of implicit flows. In particular, they track indirect dependency between dereference points and branching points. They present two languages, one that captures dependencies statically, and one that uses multiple executions of a program to record dependencies. This is yet another approach

to runtime tracking that is worth considering in order to achieve a more flexible gradual security language that fully respects the dynamic gradual guarantee.

Hybrid approaches can also support programmer-controlled flexibility. Buiras et al. [2015] propose Hybrid LIO (HLIO), a flexible monadic information-flow control library for Haskell. HLIO is not gradual in the sense that it does not include an unknown security label; instead, HLIO provides a primitive to explicitly and selectively *defer* label-ordering checks to runtime. Their approach to defer static typing constraints to runtime can even be exploited to postpone type checks beyond security label constraints, opening the door to hybrid type checking in Haskell. In contrast, as a gradual security language, GSL_{Ref} supports a notion of unknown security information and implicitly mediates the interactions between static and dynamic security checking.

Gradual security typing. Most directly related to our proposal is prior work on gradual security typing, which combines static and dynamic checking with the express intent of supporting a smooth migration between both checking disciplines by introducing a *dynamic* (*i.e.* statically unknown) security label. Disney and Flanagan [2011] and Fennell and Thiemann [2013] pioneered what we describe in Sec. 1 as a check-driven approach to gradual security typing, starting from dynamic checking. Both develop notions of blame tracking and prove blame theorems for their semantics. It is important to recall that these approaches, while dubbed "gradual", are based on *explicit* security casts, and are therefore more akin to cast calculi than to gradual languages. In particular, this means that these languages do not respect the gradual guarantees *by design*, including the static one, because changing the precision of type annotations requires adding/removing explicit casts. Additionally, as discussed in the introduction, both proposals break type-based reasoning about noninterference.

Recently, Fennell and Thiemann [2016] extend their prior work on gradual security typing with references to the object-oriented setting, in a language called LJGS. Like Jif, LJGS performs local inference of security labels, and supports polymorphic security signatures. Local variables in LJGS are typed in a flow-sensitive manner, whereas both SSL_{Ref} and GSL_{Ref} are flow insensitive regarding security levels. Although LJGS is based on explicit casts like prior work, its semantics differ in important ways. For instance, recall the example given in Sec. 1:

```
let mix : Int_L →_L Int_H →_L Int_L =
  fun pub priv => if pub < (Int_L ⇐ Int_H)priv then 1_L else 2_L
mix 1_L 5_L
```

This example does not type check in LJGS because the target type of a security cast cannot be less secure than the source type. The only way to write this example is to go through the dynamic security level explicitly:

```
let mix : Int_L →_L Int_H →_L Int_L =
  fun pub priv => if pub < (Int_L ⇐ Int_?) (Int_? ⇐ Int_H) priv then 1_L else 2_L
mix 1_L 5_L
```

This well-typed program fails at runtime because $(\text{Int}_? \Leftarrow \text{Int}_H)$ upgrades 5_L to 5_H, but $(\text{Int}_L \Leftarrow \text{Int}_?)5_H$ is not defined. This approach to upgrade the security level of values that are cast to the dynamic label using the *statically-determined* source label seems to restore type-based reasoning about noninterference in LJGS. Interestingly, the change in semantics in LGJS is solely motivated by the design goal to avoid having to dynamically track security labels of statically-typed program fragments, so the relation with type-based reasoning appears to be accidental.

Similar to the approach of Russo and Sabelfeld [2010] and Shroff et al. [2007] discussed above, LJGS relies on a side-effect analysis to tracks the updated variables in method bodies. More precisely, when typing a method, LJGS generates a set of constraints that represent the information flow

dependencies between parameters and return values, as well as two sets of effects: a local effect that lists the variables modified in branches of a conditional, used to update local variables of untaken branches; and a global effect that records the security types whose fields may be updated with sensitive information. This type analysis and constraint/effect inference is facilitated by the fact that classes in LJGS are not first-class entities, *i.e.* all class definitions are top-level and known ahead-of-time. This means in particular that at every call site, one statically knows the precise inferred constraints and effects of methods (modulo a standard subsumption criteria to account for subtyping). In a setting with higher-order types, this information would be more complex to track. Additionally, the inferred global effect of a method is insufficient information *per se* for the dynamic information flow control part of LJGS. Therefore, LJGS also appeals to an external effect analysis (left opaque) to obtain precise information about heap write effects.

Gradualizing expressive typing disciplines. Since the initial formulation of gradual typing [Siek and Taha 2006], there has been many efforts to gradualize advanced typing disciplines, like typestates [Garcia et al. 2014; Wolff et al. 2011], ownership types [Sergey and Clarke 2012], annotated type systems [Thiemann and Fennell 2014], effects [Bañados Schwerter et al. 2014, 2016; Toro and Tanter 2015], refinement types [Jafery and Dunfield 2017; Lehmann and Tanter 2017], parametric polymorphism [Ahmed et al. 2017; Igarashi et al. 2017], and the security type systems discussed above, among others.

Since the formulation of the refined criteria for gradually-typed languages [Siek et al. 2015], however, only refinement types [Jafery and Dunfield 2017; Lehmann and Tanter 2017] have been shown to fully respect such guarantees. This work contributes to the general research agenda of gradual typing disciplines by explicitly attempting to achieve both the gradual guarantees and a rich semantic property, like noninterference. Indeed, noninterference is *not* implied by type safety; in contrast, soundness of refinement types directly follows from type safety. We have shown that GSL_{Ref} does respect the static gradual guarantee (as opposed to other gradual security type systems); but GSL_{Ref} must sacrifice the dynamic gradual guarantee due to a modification of the runtime semantics that is necessary to enforce noninterference in the presence of mutable references.

Initial work on gradual parametricity [Igarashi et al. 2017] also suggests that parametricity may be incompatible with the dynamic gradual guarantee, unless one is willing to tweak the type precision relation; even then, the dynamic gradual guarantee is left as a conjecture. Ahmed et al. [2017] prove parametricity for a polymorphic cast calculus—not a source language—and also leave the gradual guarantees as an open question. Therefore, further work is needed to fully understand if and how the gradual guarantees can be reconciled with rich semantic typing disciplines, and if additional design criteria for such gradual languages should be devised.

8 CONCLUSION

We develop a novel, *type-driven* approach to gradual security typing, in which gradual security types provide strong security invariants, while admitting flexible programming idioms. This is the first work to address the gradualization of a rich typing discipline in which type safety does not imply type soundness, while pursuing the most elaborate formulation of criteria for gradually-typed languages [Siek et al. 2015], and preserving type-based reasoning principles. This means that the amount of static checking is entirely driven by the precision of static security annotations, and that programmers can reason modularly about the noninterference guarantees of program fragments by just looking at types.

Using the AGT methodology [Garcia et al. 2016] to derive the gradual security language GSL_{Ref}, this work sheds light on key semantic issues in the design of gradual languages. AGT was central in our endeavor to separate the elements of the design that follow by systematically following the

methodology from those that require careful consideration. In particular, we identify a tension between the smooth continuum on the static-to-dynamic spectrum that the gradual guarantees mandate, and the semantic property of noninterference, which manifests in GSL_{Ref} because of mutable references. This tension also raises interesting questions for the principled design of gradually-typed languages, whenever the semantics of types has a relational flavor. In particular, while we have addressed noninterference, relational parametricity remains to be addressed. Overall, this work suggests that it might be necessary to extend AGT to integrate the purely static type soundness proof—as opposed to only the type safety proof—as a source for the design of the dynamic semantics of a gradual language.

Within the context of gradual security typing, our work leaves open the question of whether it is possible to reconcile both noninterference and the dynamic gradual guarantee. Specifically, it would be informative to study whether other approaches to sound dynamic information flow control could help us recover the dynamic gradual guarantee. We believe that there might be an inherent incompatibility between the strictness required to enforce a hyper-property like noninterference, and the optimistic flexibility dictated by the dynamic gradual guarantee.

Another interesting track for future work is to explore a "pay-as-you-go" [Siek and Taha 2006] semantics, which only introduces runtime checks for imprecisely-typed expressions, as well as scaling the security discipline to other language-based security features such as integrity, flow sensitivity and declassification. Additionally, we want to explore the applicability of Garcia and Cimini [2015]'s approach to type inference in gradual languages to address security label inference [Pottier and Simonet 2003] in GSL_{Ref}.

Acknowledgments. We thank the anonymous reviewers of this paper and previous submissions for their helpful comments, questions, and detailed readings. We also thank Alison M. Clark, Joshua Dunfield, Chris Martens, and Jeremy Siek.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 147–160.

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. 2011. Blame for all. In *38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM Press, Austin, Texas, USA, 201–214.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. In *22th ACM SIGPLAN Conference on Functional Programming (ICFP 2017)*. ACM Press, Oxford, United Kingdom, 39:1–39:28.

Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. 113–124.

Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 2010 Workshop on Programming Languages and Analysis for Security (PLAS 2010)*. 3:1–3:12.

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. *ACM SIGPLAN Notices* 47, 1 (Jan. 2012), 165–178.

Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Transactions on Programming Languages and Systems* 39, 3 (July 2017), 10:1–10:56.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*. ACM Press, Gothenburg, Sweden, 283–295.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual Type-and-Effect Systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Kenneth J. Biba. 1977. *Integrity considerations for secure computer systems.* Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA, USA.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: mixing static and dynamic typing for information-flow control in Haskell. In *20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*. ACM Press, Vancouver,

Canada, 289–301.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. In *22th ACM SIGPLAN Conference on Functional Programming (ICFP 2017)*. ACM Press, Oxford, United Kingdom, 41:1–41:28.

D. Chandra and M. Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. 463–475.

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symbolic Logic* 5, 2 (06 1940), 56–68.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 77)*. ACM Press, Los Angeles, CA, USA, 238–252.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243.

Tim Disney and Cormac Flanagan. 2011. Gradual information flow typing. In *International Workshop on Scripts to Programs*.

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Computer Security Foundations Symposium*. 224–239.

Luminous Fennell and Peter Thiemann. 2016. LJGS: Gradual Security Types for Object-Oriented Languages. In *30th European Conference on Object-oriented Programming (ECOOP 2016) (LNCS)*. Springer-Verlag, Rome, Italy.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM Press, 303–315.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St Petersburg, FL, USA, 429–442.

Ronald Garcia and Éric Tanter. 2015. Deriving a Simple Gradual Security Language. available on arXiv. http://arxiv.org/abs/1511.01399

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems* 36, 4, Article 12 (Oct. 2014), 12:1–12:44 pages.

David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM Press, Cambridge, MA, USA, 28–38.

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. 11–20.

Daniel Hedin and Andrei Sabelfeld. 2012a. Information-Flow Security for a Core of JavaScript. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*. 3–18.

Daniel Hedin and Andrei Sabelfeld. 2012b. A Perspective on Information-Flow Control. In *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 319–347.

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1998)*. ACM, New York, NY, USA, 365–377.

William A. Howard. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley (Eds.). Academic Press, New York, 479–490. Reprint of 1969 article.

Sebastian Hunt and David Sands. 2006. On Flow-Sensitive Security Types. In *33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*. ACM Press, Charleston, SC, USA, 79–90.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. In *22th ACM SIGPLAN Conference on Functional Programming (ICFP 2017)*. ACM Press, Oxford, United Kingdom, 40:1–40:29.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 804–817.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010).

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 775–788.

Scott Moore and Stephen Chong. 2011. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF 2011)*. Cernay-la-Ville, France, 146–160.

Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*. ACM Press, San Antonio, TX, USA, 228–241.

Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *16th ACM Symposium on Operating System Principles (SOSP)*. 129–142.

Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology* 9 (Oct. 2000), 410–442. Issue 4.

François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems* 25, 1 (Jan. 2003), 117–158.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF '10)*. IEEE Computer Society, Washington, DC, USA, 186–199.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003).

Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548.

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *21st European Symposium on Programming Languages and Systems (ESOP 2012) (LNCS)*, Helmut Seidl (Ed.), Vol. 7211. Springer-Verlag, Tallinn, Estonia, 579–599.

Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic Dependency Monitoring to Secure Information Flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF '07)*. IEEE Computer Society, Washington, DC, USA, 203–217.

Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *21st European Conference on Object-oriented Programming (ECOOP 2007) (LNCS)*, Erik Ernst (Ed.). Springer-Verlag, Berlin, Germany, 2–27.

Jeremy Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM Press, 365–376.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. 274–293.

Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming* 27 (2017).

Peter Thiemann and Luminous Fennell. 2014. Gradual Typing for Annotated Type Systems. In *23rd European Symposium on Programming Languages and Systems (ESOP 2014) (LNCS)*, Zhong Shao (Ed.), Vol. 8410. Springer-Verlag, Grenoble, France, 47–66.

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. *Type-Driven Gradual Security with References: Complete Definitions and Proofs*. Technical Report TR/DCC-2018-4. University of Chile.

Matías Toro and Éric Tanter. 2015. Customizable Gradual Polymorphic Effects for Scala. In *30th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2015)*. ACM Press, Pittsburgh, PA, USA, 935–953.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017) (Lecture Notes in Computer Science)*, Vol. 10422. Springer-Verlag, New York City, NY, USA, 382–404.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996), 167–187.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. ACM, New York, NY, USA, 347–359.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual Typestate. In *25th European Conference on Object-oriented Programming (ECOOP 2011) (LNCS)*, Mira Mezini (Ed.), Vol. 6813. Springer-Verlag, Lancaster, UK, 459–483.

Steve Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University.

Steve Zdancewic and Andrew C. Myers. 2001. Secure information flow and CPS. In *10th European Symposium on Programming*, Vol. 2028. 46–61.

Lantian Zheng and Andrew C. Myers. 2007. Dynamic Security Labels and Noninterference. *International Journal of Information Security* 6, 2 (March 2007), 67–84.

A ADDITIONAL DEFINITIONS

In this appendix we present additional definitions that were not included in the main body of the paper. Proofs are in the companion technical report [Toro et al. 2018].

A.1 SSL_{Ref}: Static semantics

In this section we present additional definitions of the static semantics of SSL_{Ref}. The join between types and labels is defined as follows

$$\begin{gathered}\text{Bool}_\ell \curlyvee \ell' = \text{Bool}_{(\ell \curlyvee \ell')} \\ (S_1 \xrightarrow{\ell_c}_\ell S_2) \curlyvee \ell' = S_1 \xrightarrow{\ell_c}_{(\ell \curlyvee \ell')} S_2 \\ \text{Ref}_\ell\ S \curlyvee \ell' = \text{Ref}_{(\ell \curlyvee \ell')}\ S\end{gathered}$$

Figure 11 presents the join and meet type functions.

$$\boxed{S \mathbin{\dot{\curlyvee}} S, S \mathbin{\dot{\curlywedge}} S}$$

$$\begin{gathered}\dot{\curlyvee} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type} \\ \text{Bool}_\ell \mathbin{\dot{\curlyvee}} \text{Bool}_{\ell'} = \text{Bool}_{(\ell \curlyvee \ell')} \\ (S_{11} \xrightarrow{\ell_c}_\ell S_{12}) \mathbin{\dot{\curlyvee}} (S_{21} \xrightarrow{\ell'_c}_{\ell'} S_{22}) = (S_{11} \mathbin{\dot{\curlywedge}} S_{21}) \xrightarrow{\ell_c \curlywedge \ell'_c}_{(\ell \curlyvee \ell')} (S_{12} \mathbin{\dot{\curlyvee}} S_{22}) \\ \text{Ref}_\ell\ S \mathbin{\dot{\curlyvee}} \text{Ref}_{\ell'}\ S = \text{Ref}_{(\ell \curlyvee \ell')}\ S \\ S \mathbin{\dot{\curlyvee}} S \text{ undefined otherwise}\end{gathered}$$

$$\begin{gathered}\dot{\curlywedge} : \textsc{Type} \times \textsc{Type} \rightharpoonup \textsc{Type} \\ \text{Bool}_\ell \mathbin{\dot{\curlywedge}} \text{Bool}_{\ell'} = \text{Bool}_{(\ell \curlywedge \ell')} \\ (S_{11} \xrightarrow{\ell_c}_\ell S_{12}) \mathbin{\dot{\curlywedge}} (S_{21} \xrightarrow{\ell'_c}_{\ell'} S_{22}) = (S_{11} \mathbin{\dot{\curlyvee}} S_{21}) \xrightarrow{\ell_c \curlyvee \ell'_c}_{(\ell \curlywedge \ell')} (S_{12} \mathbin{\dot{\curlywedge}} S_{22}) \\ \text{Ref}_\ell\ S \mathbin{\dot{\curlywedge}} \text{Ref}_{\ell'}\ S = \text{Ref}_{(\ell \curlywedge \ell')}\ S \\ S \mathbin{\dot{\curlywedge}} S \text{ undefined otherwise}\end{gathered}$$

Fig. 11. SSL_{Ref}: Join and meet type functions

Definition A.1 (Valid Type Sets).

$$\frac{}{valid(\{\,\overline{\text{Bool}_{\ell_i}}\,\})} \qquad \frac{valid(\{\,\overline{S_{i1}}\,\}) \qquad valid(\{\,\overline{S_{i2}}\,\})}{valid(\{\,\overline{S_{i1} \xrightarrow{\ell_{c\,i}}_{\ell_i} S_{i2}}\,\})} \qquad \frac{valid(\{\,\overline{S_i}\,\})}{valid(\{\,\overline{\text{Ref}_{\ell_i}\ S_i}\,\})}$$

$$\frac{}{valid(\{\,\overline{\text{Unit}_{\ell_i}}\,\})}$$

A.2 SSL_{Ref}: Noninterference definitions

In this section we present definitions and properties of noninterference for SSL_{Ref}. Figure 12 presents the full definition of step-indexed logical relations.

Definition A.2. Let ρ be a substitution, Γ and Σ a type substitutions. We say that substitution ρ satisfy environment Γ and Σ, written $\rho \models \Gamma; \Sigma$, if and only if $dom(\rho) = \Gamma$ and $\forall x \in dom(\Gamma), \forall \ell_c, \Gamma; \Sigma; \ell_c \vdash \rho(x) : S'$, where $S' <: \Gamma(x)$.

Definition A.3 (Related substitutions). Tuples $\langle \ell_1, \rho_1, \mu_1 \rangle$ and $\langle \ell_2, \rho_2, \mu_2 \rangle$ are related on k steps, notation $\Gamma; \Sigma \vdash \langle \ell_1, \rho_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_2, \rho_2, \mu_2 \rangle$, if $\rho_i \models \Gamma; \Sigma, \Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2$ and

$$\forall x \in \Gamma. \Sigma \vdash \langle \ell_1, \rho_1(x), \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_2, \rho_2(x), \mu_2 \rangle : \Gamma(x)$$

$$\Sigma \vdash \langle \ell_1, v_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_2, v_2, \mu_2 \rangle : S \iff \ell_1 \approx_{\ell_o} \ell_2 \wedge \Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2 \wedge \Sigma; \ell_i \vdash v_i : S'_i, S'_i <: S,$$
$$\wedge \left(\mathsf{obs}_{\ell_o}(\ell_i, S) \implies \mathsf{obsRel}^{\Sigma,S}_{k,\ell_o}(\ell_1, v_1, \mu_1, \ell_2, , v_2, \mu_2) \right)$$

$$\mathsf{obsRel}^{\Sigma,S}_{k,\ell_o}(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) \iff (\mathit{rval}(v_1) = \mathit{rval}(v_2)) \qquad \text{if } S \in \{\mathsf{Bool}_g, \mathsf{Unit}_g, \mathsf{Ref}_g\ S'\}$$

$$\mathsf{obsRel}^{\Sigma,S_1 \xrightarrow{\ell'}_{\ell} S_2}_{k,\ell_o}(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) \iff \forall j \le k.\, \forall \Sigma \subseteq \Sigma', \Sigma' \vdash \langle \ell_1, v'_1, \mu'_1 \rangle \approx^j_{\ell_o} \langle \ell_2, v'_2, \mu'_2 \rangle : S_1,$$
$$\Sigma' \vdash \langle \ell_1, v_1\ v'_1, \mu'_1 \rangle \approx^j_{\ell_o} \langle \ell_2, v_2\ v'_2, \mu'_2 \rangle : \mathcal{C}(S_2 \mathbin{\widetilde{\vee}} g)$$

$$\Sigma \vdash \langle \ell_1, t_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_2, t_2, \mu_2 \rangle : \mathcal{C}(S) \iff \ell_1 \approx_{\ell_o} \ell_2 \wedge \Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2 \wedge \Sigma; \ell_i \vdash t_i : S'_i, S'_i <: S, \forall j < k$$
$$\left(t_i \mid \mu_i \xrightarrow{\ell_i}{}^{j} t'_i \mid \mu'_i \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu'_1 \approx^{k-j}_{\ell_o} \mu'_2 \wedge \right.$$
$$\left. (\mathit{irred}(t'_i) \implies \Sigma' \vdash \langle \ell_1, t'_1, \mu'_1 \rangle \approx^{k-j}_{\ell_o} \langle \ell_2, t'_2, \mu'_2 \rangle : S) \right)$$

$$\begin{array}{rcl}
\Sigma \vdash \mu_1 \approx^k_{\ell_o} \mu_2 & \iff & \Sigma \vdash \mu_i \wedge \forall \ell_i, \ell_1 \approx_{\ell_o} \ell_2, j < k, \forall o \in \mathit{dom}(\mu_1) \cap \mathit{dom}(\mu_2) \\
& & \Sigma \vdash \langle \ell_1, \mu_1(o), \mu_1 \rangle \approx^j_{\ell_o} \langle \ell_2, \mu_2(o), \mu_2 \rangle : \Sigma(o) \\
\ell_1 \approx_{\ell_o} \ell_2 & \iff & \mathsf{obs}_{\ell_o}(\ell_i) \vee \neg \mathsf{obs}_{\ell_o}(\ell_i) \\
\mu_1 \rightarrowtriangle \mu_2 & \iff & \mathit{dom}(\mu_1) \subseteq \mathit{dom}(\mu_2) \\
\mathsf{obs}_{\ell_o}(\ell, S) & \iff & \mathsf{obs}_{\ell_o}(\ell) \wedge \mathsf{obs}_{\ell_o}(\mathit{label}(S)) \\
\mathsf{obs}_{\ell_o}(\ell) & \iff & \ell \preccurlyeq \ell_o
\end{array}$$

Fig. 12. Security logical relations

Definition A.4 (Semantic Security Typing).

$$\begin{array}{rcl}
\Gamma; \Sigma; \ell_c \models t : S & \iff & \forall\ \ell_o \in \textsc{Label}, k \ge 0, \rho_1, \rho_2 \in \textsc{Subst} \text{ and } \mu_1, \mu_2 \in \textsc{Store} \\
& & \text{such that } \Sigma \vdash \mu_i \text{ and } \Gamma; \Sigma \vdash \langle \ell_c, \rho_1, \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_c, \rho_2, \mu_2 \rangle\text{, we have} \\
& & \Sigma \vdash \langle \ell_c, \rho_1(t), \mu_1 \rangle \approx^k_{\ell_o} \langle \ell_c, \rho_2(t), \mu_2 \rangle : \mathcal{C}(S)
\end{array}$$

Proposition A.5 (Security Type Soundness). *If* $\Gamma; \Sigma; \ell_c \vdash t : S'_i \implies \forall S, S'_i <: S, \Gamma; \Sigma; \ell_c \models t : S$

A.3 GSL_{Ref}: Static semantics

In this section we present some additional definitions needed in gradualizing SSL_{Ref}.

Definition A.6 (Type Concretization). $\gamma_S : \text{GType} \to \mathcal{P}(\text{Type})$

$$\gamma_S(\text{Bool}_g) = \{\, \text{Bool}_\ell \mid \ell \in \gamma(g) \,\} \qquad \gamma_S(U_1 \xrightarrow{g} U_2) = \gamma_S(U_1) \xrightarrow{\gamma(g')}_{\gamma(g)} \gamma_S(U_2)$$

$$\gamma_S(\text{Unit}_g) = \{\, \text{Unit}_\ell \mid \ell \in \gamma(g) \,\} \qquad \gamma_S(\text{Ref}_g\ U) = \{\, \text{Ref}_\ell\ S \mid \ell \in \gamma(g), S \in \gamma_S(U) \,\}$$

Type concretization induces notions of precision and abstraction.

Definition A.7 (Type Precision). $U_1 \sqsubseteq U_2$, if and only if $\gamma_S(U_1) \subseteq \gamma_S(U_2)$.

Definition A.8 (Type Abstraction). $\alpha_S : \mathcal{P}(\text{Type}) \to \text{GType}$

$$\alpha_S(\{\, \overline{\text{Bool}_{\ell_i}} \,\}) = \text{Bool}_{\alpha(\{\, \overline{\ell_i} \,\})} \qquad \alpha_S(\{\, \overline{\text{Unit}_{\ell_i}} \,\}) = \text{Unit}_{\alpha(\{\, \overline{\ell_i} \,\})}$$

$$\alpha_S(\{\, \overline{S_{i1} \xrightarrow{\ell'_i} \ell_i S_{i2}} \,\}) = \alpha_S(\{\, \overline{S_{i1}} \,\}) \xrightarrow[\alpha(\{\, \overline{\ell_i} \,\})]{\alpha(\{\, \overline{\ell'_i} \,\})} \alpha_S(\{\, \overline{S_{i2}} \,\}) \qquad \alpha_S(\{\, \overline{\text{Ref}_{\ell_i}\ S_i} \,\}) = \text{Ref}_{\alpha(\{\, \overline{\ell_i} \,\})}\ \alpha_S(\{\, \overline{S_i} \,\})$$

$\alpha_S(\widehat{S})$ is undefined otherwise

Proposition A.9 (α_S is Sound and Optimal). *Assuming* $\widehat{S}$ **valid***:*
(i) $\widehat{S} \subseteq \gamma_S(\alpha_S(\widehat{S}))$ *(ii) If* $\widehat{S} \subseteq \gamma_S(U)$ *then* $\alpha_S(\widehat{S}) \sqsubseteq U$.

Definition A.10 (Gradual label meet).
$g_1 \mathbin{\widetilde{\wedge}} g_2 = \alpha(\{\, \ell_1 \wedge \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \,\})$.

$$\boxed{U \mathbin{\widetilde{\sqcap}} U}$$

$$\begin{aligned}
&\widetilde{\sqcap} : \text{Type} \times \text{Type} \rightharpoonup \text{Type} \\
&\text{Bool}_g \mathbin{\widetilde{\sqcap}} \text{Bool}_{g'} = \text{Bool}_{(g \mathbin{\widetilde{\wedge}} g')} \\
&(U_{11} \xrightarrow{g_c}_g U_{12}) \mathbin{\widetilde{\sqcap}} (U_{21} \xrightarrow{g'_c}_{g'} U_{22}) = (U_{11} \mathbin{\widetilde{\vee}} U_{21}) \xrightarrow[(g \mathbin{\widetilde{\wedge}} g')]{g_c \mathbin{\widetilde{\vee}} g'_c} (U_{12} \mathbin{\widetilde{\sqcap}} U_{22}) \\
&\text{Ref}_g\ U \mathbin{\widetilde{\sqcap}} \text{Ref}_{g'}\ U' = \text{Ref}_{(g \mathbin{\widetilde{\wedge}} g')}\ U \sqcap U' \\
&U \mathbin{\widetilde{\sqcap}} U \text{ undefined otherwise}
\end{aligned}$$

Fig. 13. GSL_{Ref}: consistent meet

Definition A.11 (Gradual label join). $g_1 \mathbin{\widetilde{\vee}} g_2 = \alpha(\{\, \ell_1 \vee \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \,\})$

Definition A.12 (Label Meet). $g_1 \sqcap g_2 = \alpha(\gamma(g_1) \cap \gamma(g_2))$.

Definition A.13 (Type Meet). $U_1 \sqcap U_2 = \alpha_S(\gamma_S(U_1) \cap \gamma_S(U_2))$.

Also, we introduce a function *label*, which yields the security label of a given type:

$$\mathit{label} : \text{GType} \to \text{Label}$$

$$\mathit{label}(\text{Bool}_g) = g \qquad \mathit{label}(\text{Unit}_g) = g \qquad \mathit{label}(U_1 \to_g U_2) = g \qquad \mathit{label}(\text{Ref}_g\ U) = g$$

$$\text{(Ix)}\ \frac{x : U \in \Gamma}{\Gamma; \Sigma; \varepsilon g_c \vdash x : U} \qquad \text{(Ib)}\ \frac{}{\Gamma; \Sigma; \varepsilon g_c \vdash b_g : \mathsf{Bool}_g} \qquad \text{(Iu)}\ \frac{}{\Gamma; \Sigma; \varepsilon g_c \vdash \mathsf{unit}_g : \mathsf{Unit}_g}$$

$$\text{(Il)}\ \frac{o : U \in \Sigma}{\Gamma; \Sigma; \varepsilon g_c \vdash o_g : \mathsf{Ref}_g\ U} \qquad \text{(I}\lambda\text{)}\ \frac{\Gamma, x : U_1; \Sigma; \varepsilon' g' \vdash t : U_2 \qquad \varepsilon' = \mathcal{G}^{\cup}_{\preccurlyeq}(g')}{\Gamma; \Sigma; \varepsilon g_c \vdash (\lambda^{g'} x : U_1.t)_g : U_1 \xrightarrow{g'}_g U_2}$$

$$\text{(Iprot)}\ \frac{\Gamma; \Sigma; \varepsilon' g'_c \vdash t : U' \qquad \varepsilon_1 \vdash U' \lesssim U \qquad \varepsilon_2 \vdash g' \mathrel{\widetilde{\preccurlyeq}} g}{\Gamma; \Sigma; \varepsilon g_c \vdash \mathsf{prot}_{\varepsilon_2 g'} \varepsilon' g'_c (\varepsilon_1 t) : U \mathrel{\widetilde{\curlyvee}} g} \qquad \text{(I}\varepsilon\text{)}\ \frac{\Gamma; \Sigma; \varepsilon g_c \vdash t : U_1 \qquad \varepsilon_1 \vdash U_1 \lesssim U_2}{\Gamma; \Sigma; \varepsilon g_c \vdash \varepsilon_1 t : U_2}$$

$$\text{(Iapp)}\ \frac{\Gamma; \Sigma; \varepsilon g_c \vdash t_i : U_i \qquad \varepsilon_1 \vdash U_1 \lesssim U_{11} \xrightarrow{g'}_g U_{12} \qquad \varepsilon_2 \vdash U_2 \lesssim U_{11} \qquad \varepsilon_3 \vdash \widetilde{g'_c \curlyvee g \preccurlyeq g'}}{\Gamma; \Sigma; \varepsilon g_c \vdash \varepsilon_1 t_1 \ @_{\varepsilon_3}\ \varepsilon_2 t_2 : U_{12} \mathrel{\widetilde{\curlyvee}} g}$$

$$\text{(Iif)}\ \frac{\begin{array}{c}\Gamma; \Sigma; \varepsilon g_c \vdash t_1 : U_1 \qquad \varepsilon_1 \vdash U_1 \lesssim \mathsf{Bool}_g \qquad \varepsilon' g'_c = (\varepsilon \mathrel{\widetilde{\curlyvee}} \mathit{ilbl}(\varepsilon_1))(g_c \mathrel{\widetilde{\curlyvee}} g) \\ \Gamma; \Sigma; \varepsilon' g'_c \vdash t_2 : U_2 \qquad \varepsilon_2 \vdash U_2 \lesssim U_2 \mathrel{\widetilde{\veebar}} U_3 \qquad \Gamma; \Sigma; \varepsilon' g'_c \vdash t_3 : U_3 \qquad \varepsilon_3 \vdash U_3 \lesssim U_2 \mathrel{\widetilde{\veebar}} U_3\end{array}}{\Gamma; \Sigma; \varepsilon g_c \vdash \mathsf{if}\ \varepsilon_1 t_1\ \mathsf{then}\ \varepsilon_2 t_2\ \mathsf{else}\ \varepsilon_3 t_3 : (U_2 \mathrel{\widetilde{\veebar}} U_3) \mathrel{\widetilde{\curlyvee}} g}$$

$$\text{(I}\oplus\text{)}\ \frac{\begin{array}{c}\Gamma; \Sigma; \varepsilon g_c \vdash t_1 : U_1 \qquad \varepsilon_1 \vdash U_1 \lesssim \mathsf{Bool}_{g_1} \\ \Gamma; \Sigma; \varepsilon g_c \vdash t_2 : U_2 \qquad \varepsilon_2 \vdash U_2 \lesssim \mathsf{Bool}_{g_2}\end{array}}{\Gamma; \Sigma; \varepsilon g_c \vdash \varepsilon_1 t_1 \oplus \varepsilon_2 t_2 : \mathsf{Bool}_{g_1 \mathrel{\widetilde{\curlyvee}} g_2}} \qquad \text{(Iref)}\ \frac{\begin{array}{c}\Gamma; \Sigma; \varepsilon g_c \vdash t : U' \\ \varepsilon_1 \vdash U' \lesssim U \qquad \varepsilon_2 \vdash g'_c \mathrel{\widetilde{\preccurlyeq}} \mathit{label}(U)\end{array}}{\Gamma; \Sigma; \varepsilon g_c \vdash \mathsf{ref}^U_{\varepsilon_2}\ \varepsilon_1 t : \mathsf{Ref}_\perp\ U}$$

$$\text{(Ideref)}\ \frac{\Gamma; \Sigma; \varepsilon g_c \vdash t : U' \qquad \varepsilon' \vdash U' \lesssim \mathsf{Ref}_g\ U}{\Gamma; \Sigma; \varepsilon g_c \vdash !\varepsilon' t : U \mathrel{\widetilde{\curlyvee}} g}$$

$$\text{(Iassgn)}\ \frac{\begin{array}{c}\Gamma; \Sigma; \varepsilon g_c \vdash t_1 : \mathsf{Ref}_{g'}\ U'_1 \qquad \varepsilon_1 \vdash \mathsf{Ref}_{g'}\ U'_1 \lesssim \mathsf{Ref}_g\ U_1 \\ \Gamma; \Sigma; \varepsilon g_c \vdash t_2 : U_2 \qquad \varepsilon_2 \vdash U_2 \lesssim U_1 \qquad \varepsilon_3 \vdash \widetilde{g'_c \curlyvee g \preccurlyeq \mathit{label}(U_1)}\end{array}}{\Gamma; \Sigma; \varepsilon g_c \vdash \varepsilon_1 t_1 :=_{\varepsilon_3} \varepsilon_2 t_2 : \mathsf{Unit}_\perp}$$

Every type rule has the extra premise $\varepsilon \vdash g_c \mathrel{\widetilde{\preccurlyeq}} g'_c$.

Fig. 14. $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Static Semantics

$$\langle \iota_1, \iota_2 \rangle \sqcap \langle \iota'_1, \iota'_2 \rangle = \langle \iota_1 \sqcap \iota'_1, \iota_2 \sqcap \iota'_2 \rangle \qquad \langle \iota_1, \iota_2 \rangle \mathrel{\widetilde{\curlywedge}} \langle \iota'_1, \iota'_2 \rangle = \langle \iota_1 \curlywedge \iota'_1, \iota_2 \curlywedge \iota'_2 \rangle$$

Fig. 15. $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Auxiliary functions for the dynamic semantics (Labels)

A.4 $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Static semantics

The static semantics of $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$ is presented in Figure 14.

A.5 $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Dynamic semantics

In this section we present additional definition of the dynamic semantics of $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$. Auxiliary functions for evidence for labels are presented in Figure 15. Auxiliary functions for evidence for types are shown in Figure 16, and the inversion functions for evidence are in Figure 17.

$$\mathsf{Bool}_\iota \sqcap \mathsf{Bool}_{\iota'} = \mathsf{Bool}_{\iota \sqcap \iota'} \qquad \mathsf{Ref}_\iota\ E_1 \sqcap \mathsf{Ref}_{\iota'}\ E_2 = \mathsf{Ref}_{\iota \sqcap \iota'}\ E_1 \sqcap E_2$$

$$(E_{11} \xrightarrow{\iota_2}_{\iota_1} E_{12}) \sqcap (E_{21} \xrightarrow{\iota'_2}_{\iota'_1} E_{22}) = (E_{11} \sqcap E_{21}) \xrightarrow[\iota_1 \sqcap \iota'_1]{\iota_2 \sqcap \iota'_2} (E_{12} \sqcap E_{22}) \qquad E \sqcap E' \text{ undefined otherwise}$$

$$\mathsf{Bool}_{\iota_1} \mathbin{\tilde{\curlyvee}} \iota_2 = \mathsf{Bool}_{(\iota_1 \tilde{\curlyvee} \iota_2)} \qquad E_1 \xrightarrow{\iota_2}_{\iota_1} E_2 \mathbin{\tilde{\curlyvee}} \iota_3 = E_1 \xrightarrow{\iota_2}_{(\iota_1 \tilde{\curlyvee} \iota_3)} E_2 \qquad \mathsf{Ref}_{\iota_1}\ E \mathbin{\tilde{\curlyvee}} \iota_2 = \mathsf{Ref}_{(\iota_1 \tilde{\curlyvee} \iota_2)}\ E$$

$$\mathsf{Bool}_{\iota_1} \mathbin{\tilde{\curlywedge}} \iota_2 = \mathsf{Bool}_{(\iota_1 \tilde{\curlywedge} \iota_2)} \qquad E_1 \xrightarrow{\iota_2}_{\iota_1} E_2 \mathbin{\tilde{\curlywedge}} \iota_3 = E_1 \xrightarrow{\iota_2}_{(\iota_1 \tilde{\curlywedge} \iota_3)} E_2 \qquad \mathsf{Ref}_{\iota_1}\ E \mathbin{\tilde{\curlywedge}} \iota_2 = \mathsf{Ref}_{(\iota_1 \tilde{\curlywedge} \iota_2)}\ E$$

$$\langle E_1, E_2 \rangle \mathbin{\tilde{\curlyvee}} \langle \iota_1, \iota_2 \rangle = \langle E_1 \mathbin{\tilde{\curlyvee}} \iota_1, E_2 \mathbin{\tilde{\curlyvee}} \iota_2 \rangle \qquad \langle E_1, E_2 \rangle \mathbin{\tilde{\curlywedge}} \langle \iota_1, \iota_2 \rangle = \langle E_1 \mathbin{\tilde{\curlywedge}} \iota_1, E_2 \mathbin{\tilde{\curlywedge}} \iota_2 \rangle$$

$$\mathsf{Bool}_{\iota_1} \mathbin{\tilde{\curlyvee}} \mathsf{Bool}_{\iota_2} = \mathsf{Bool}_{(\iota_1 \tilde{\curlyvee} \iota_2)} \qquad E_1 \xrightarrow{\iota_2}_{\iota_1} E_2 \mathbin{\tilde{\curlyvee}} E'_1 \xrightarrow{\iota'_2}_{\iota'_1} E'_2 = E_1 \mathbin{\tilde{\curlywedge}} E'_1 \xrightarrow[(\iota_1 \tilde{\curlyvee} \iota'_1)]{\iota_2 \tilde{\curlywedge} \iota'_2} E_2 \mathbin{\tilde{\curlyvee}} E'_2$$

$$\mathsf{Ref}_{\iota_1}\ E_1 \mathbin{\tilde{\curlyvee}} \mathsf{Ref}_{\iota'_1}\ E'_1 = \mathsf{Ref}_{(\iota_1 \tilde{\curlyvee} \iota'_1)}\ E_1 \sqcap E'_1 \qquad \mathsf{Bool}_{\iota_1} \mathbin{\tilde{\curlywedge}} \mathsf{Bool}_{\iota_2} = \mathsf{Bool}_{(\iota_1 \tilde{\curlywedge} \iota_2)}$$

$$E_1 \xrightarrow{\iota_2}_{\iota_1} E_2 \mathbin{\tilde{\curlywedge}} E'_1 \xrightarrow{\iota'_2}_{\iota'_1} E'_2 = E_1 \mathbin{\tilde{\curlyvee}} E'_1 \xrightarrow[(\iota_1 \tilde{\curlywedge} \iota'_1)]{\iota_2 \tilde{\curlyvee} \iota'_2} E_2 \mathbin{\tilde{\curlywedge}} E'_2 \qquad \mathsf{Ref}_{\iota_1}\ E_1 \mathbin{\tilde{\curlywedge}} \mathsf{Ref}_{\iota'_1}\ E'_1 = \mathsf{Ref}_{(\iota_1 \tilde{\curlywedge} \iota'_1)}\ E_1 \sqcap E'_1$$

$$\langle E_1, E_2 \rangle \mathbin{\tilde{\curlyvee}} \langle E'_1, E'_2 \rangle = \langle E_1 \mathbin{\tilde{\curlyvee}} E'_1, E_2 \mathbin{\tilde{\curlyvee}} E'_2 \rangle \qquad \langle E_1, E_2 \rangle \mathbin{\tilde{\curlywedge}} \langle E'_1, E'_2 \rangle = \langle E_1 \mathbin{\tilde{\curlywedge}} E'_1, E_2 \mathbin{\tilde{\curlywedge}} E'_2 \rangle$$

$$\frac{\triangle^{\leqslant}(\iota_1, \iota_2, \iota_3) = \langle \iota'_1, \iota'_3 \rangle}{\triangle^{<:}(\mathsf{Bool}_{\iota_1}, \mathsf{Bool}_{\iota_2}, \mathsf{Bool}_{\iota_3}) = \langle \mathsf{Bool}_{\iota'_1}, \mathsf{Bool}_{\iota'_3} \rangle}$$

$$\frac{\begin{array}{c}\triangle^{<:}(E_{31}, E_{21}, E_{11}) = \langle E'_{31}, E'_{11} \rangle \qquad \triangle^{<:}(E_{12}, E_{22}, E_{32}) = \langle E'_{12}, E'_{32} \rangle \\ \triangle^{\leqslant}(\iota_1, \iota_2, \iota_3) = \langle \iota'_1, \iota'_3 \rangle \qquad \triangle^{\leqslant}(\iota_{13}, \iota_{12}, \iota_{11}) = \langle \iota'_{13}, \iota'_{11} \rangle\end{array}}{\triangle^{<:}(E_{11} \xrightarrow{\iota_{11}}_{\iota_1} E_{12}, E_{21} \xrightarrow{\iota_{12}}_{\iota_2} E_{22}, E_{31} \xrightarrow{\iota_{13}}_{\iota_3} E_{32}) = \langle E'_{11} \xrightarrow{\iota'_{11}}_{\iota'_1} E'_{12}, E'_{31} \xrightarrow{\iota'_{13}}_{\iota'_3} E'_{32} \rangle}$$

$$\frac{\triangle^{\leqslant}(\iota_1, \iota_2, \iota_3) = \langle \iota'_1, \iota'_3 \rangle \qquad E'_1 = E_1 \sqcap E_2 \qquad E'_3 = E_2 \sqcap E_3}{\triangle^{<:}(\mathsf{Ref}_{\iota_1}\ E_1, \mathsf{Ref}_{\iota_2}\ E_2, \mathsf{Ref}_{\iota_3}\ E_3) = \langle \mathsf{Ref}_{\iota'_1}\ E'_1, \mathsf{Ref}_{\iota'_3}\ E'_3 \rangle}$$

$$\langle E_1, E_{21} \rangle \circ^{<:} \langle E_{22}, E_3 \rangle = \triangle^{<:}(E_1, E_{21} \sqcap E_{22}, E_3)$$

Fig. 16. $\mathsf{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Auxiliary functions for the dynamic semantics (Types)

Definition A.14 (Type Evidence Concretization). Let $\gamma_E : \textsc{GEType} \to \mathcal{P}(\textsc{Type})$ be defined as follows:

$$\gamma_E(\mathsf{Bool}_\iota) = \{\, \mathsf{Bool}_\ell \mid \ell \in \gamma_\iota(\iota) \,\}$$

$$\gamma_E(E_1 \xrightarrow{\iota_2}_{\iota_1} E_2) = \gamma_E(E_1) \xrightarrow{\gamma_\iota(\iota_2)}_{\gamma_\iota(\iota_1)} \gamma_E(E_2)$$

$$\gamma_E(\mathsf{Ref}_\iota\ E) = \{\, \mathsf{Ref}_\ell\ S \mid \ell \in \gamma_\iota(\iota), S \in \gamma_E(E) \,\}$$

where $\to$ is the set of all possible combinations of function types, using each member of the sets obtained by the γ_E and γ_ι functions.

$$\begin{aligned}
\mathit{ilbl}(\langle \mathsf{Bool}_{\iota_1}, \mathsf{Bool}_{\iota_2} \rangle) &= \langle \iota_1, \iota_2 \rangle \\
\mathit{ilbl}(\langle \mathsf{Unit}_{\iota_1}, \mathsf{Unit}_{\iota_2} \rangle) &= \langle \iota_1, \iota_2 \rangle \\
\mathit{ilbl}(\langle \mathsf{Ref}_{\iota_1}\ U_1, \mathsf{Ref}_{\iota_2}\ U_2 \rangle) &= \langle \iota_1, \iota_2 \rangle \\
\mathit{ilbl}(\langle E_1 \xrightarrow{\iota_2}_{\iota_1} E_2, E_1' \xrightarrow{\iota_2'}_{\iota_1'} E_2' \rangle) &= \langle \iota_1, \iota_1' \rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{iref}(\langle \mathsf{Ref}_{\iota_1}\ E_1, \mathsf{Ref}_{\iota_2}\ E_2 \rangle) &= \langle E_1, E_2 \rangle \\
\mathit{iref}(\langle E_1, E_2 \rangle) &= \text{undefined otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{idom}(\langle E_1 \xrightarrow{\iota_2}_{\iota_1} E_2, E_1' \xrightarrow{\iota_2'}_{\iota_1'} E_2' \rangle) &= \langle E_1', E_1 \rangle \\
\mathit{idom}(\langle E_1, E_2 \rangle) &= \text{undefined otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{icod}(\langle E_1 \xrightarrow{\iota_2}_{\iota_1} E_2, E_1' \xrightarrow{\iota_2'}_{\iota_1'} E_2' \rangle) &= \langle E_2, E_2' \rangle \\
\mathit{icod}(\langle E_1, E_2 \rangle) &= \text{undefined otherwise}
\end{aligned}$$

Fig. 17. $\mathrm{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Inversion functions for evidence

Definition A.15 (Evidence Concretization). Let $\gamma_{\varepsilon_\ell} : \mathrm{GETYPE}^2 \to \mathcal{P}(\mathrm{TYPE}^2)$ be defined as follows:

$$\gamma_{\varepsilon_\ell}(\langle E_1, E_2 \rangle) = \{\langle S_1, S_2 \rangle \mid S_1 \in \gamma_E(E_1), S_2 \in \gamma_E(E_2)\}$$

Definition A.16 (Type Evidence Abstraction). Let the abstraction function $\alpha_E : \mathcal{P}(\mathrm{TYPE}) \to \mathrm{GETYPE}$ be defined as:

$$\begin{aligned}
\alpha_E(\{\, \overline{\mathsf{Bool}_{\ell_i}} \,\}) &= \mathsf{Bool}_{\alpha_\iota(\{\, \overline{\ell_i} \,\})} \\
\alpha_E(\{\, \overline{S_{i1} \xrightarrow{\ell_{ci}} \ell_i S_{i2}} \,\}) &= \alpha_E(\{\, \overline{S_{i1}} \,\}) \xrightarrow[\alpha_\iota(\{\, \overline{\ell_i} \,\})]{\alpha_\iota(\{\, \overline{\ell_{ci}} \,\})} \alpha_E(\{\, \overline{S_{i2}} \,\}) \\
\alpha_E(\{\, \overline{\mathsf{Ref}_{\ell_i}\ S_i} \,\}) &= \mathsf{Ref}_{\alpha_\iota(\{\, \overline{\ell_i} \,\})}\ \alpha_E(\{\, \overline{S_i} \,\}) \\
\alpha_E(\widehat{S}) &\text{ is undefined otherwise}
\end{aligned}$$

Definition A.17 (Evidence Abstraction). Let $\alpha_\varepsilon : \mathcal{P}(\mathrm{TYPE}^2) \to \mathrm{GETYPE}^2$ be defined as follows:

$$\begin{aligned}
\alpha_\varepsilon(\emptyset) &\text{ is undefined} \\
\alpha_\varepsilon(\{\, \overline{\langle S_{1i}, S_{2i} \rangle} \,\}) &= \langle \alpha_E(\{\, \overline{S_{1i}} \,\}), \alpha_E(\{\, \overline{S_{2i}} \,\}) \rangle \text{ otherwise}
\end{aligned}$$

Proposition A.18 (α_ι is Sound). *If $\widehat{\ell}$ is not empty, then $\widehat{\ell} \subseteq \gamma_\iota(\alpha_\iota(\widehat{\ell}))$.*

Proposition A.19 (α_ι is Optimal). *If $\widehat{\ell}$ is not empty, and $\widehat{\ell} \subseteq \gamma_\iota(\iota)$ then $\alpha_\iota(\widehat{\ell}) \sqsubseteq \iota$.*

Proposition A.20 (α_E is Sound). *If valid$(\widehat{S})$ then $\widehat{S} \subseteq \gamma_E(\alpha_E(\widehat{S}))$.*

Proposition A.21 (α_E is Optimal). *If valid$(\widehat{S})$ and $\widehat{S} \subseteq \gamma_E(E)$ then $\alpha_E(\widehat{S}) \sqsubseteq E$.*

$$
\begin{array}{rl}
\mathit{bounds}(?) & = [\bot, \top] \\
\mathit{bounds}(\ell) & = [\ell, \ell] \\
\mathit{bounds}(x_1 \curlyvee x_2) & = \mathit{bounds}(x_1) \curlyvee \mathit{bounds}(x_2) \\
\mathit{bounds}(x_1 \curlywedge x_2) & = \mathit{bounds}(x_1) \curlywedge \mathit{bounds}(x_2) \\
\mathit{bounds}(x_1 \sqcap x_2) & = \mathit{bounds}(x_1) \sqcap \mathit{bounds}(x_2) \\
\mathit{bounds}(F_1(\overline{x_i}) \curlyvee F_2(\overline{x_i})) & = \mathit{bounds}(F_1(\overline{x_i})) \curlyvee \mathit{bounds}(F_2(\overline{x_i})) \\
\mathit{bounds}(F_1(\overline{x_i}) \curlywedge F_2(\overline{x_i})) & = \mathit{bounds}(F_1(\overline{x_i})) \curlywedge \mathit{bounds}(F_2(\overline{x_i})) \\
\mathit{bounds}(F_1(\overline{x_i}) \sqcap F_2(\overline{x_i})) & = \mathit{bounds}(F_1(\overline{x_i})) \sqcap \mathit{bounds}(F_2(\overline{\ell_i}))
\end{array}
$$

$$
\frac{\mathit{bounds}(F_1(\overline{g_i})) = [\ell_1, \ell_2] \qquad \mathit{bounds}(F_2(\overline{g_j})) = [\ell'_1, \ell'_2]}{\mathcal{J}(F_1(g_1, \ldots \widetilde{g_n) \preccurlyeq F_2(g_{n+1}, \ldots g_{n+m})}) = \langle [\ell_1, \ell_2 \curlywedge \ell'_2], [\ell_1 \ \curlyvee \ell'_1, \ell'_2] \rangle}
$$

where $F_1 : \textsc{GLabel}^n \to \textsc{GLabel}$ and $F_2 : \textsc{GLabel}^m \to \textsc{GLabel}$.

$$
\mathcal{J}^{\cup}(\widetilde{F(g_1, \ldots, g_n)}) = \mathcal{J}(F(g_1, \ldots, \widetilde{g_n) \preccurlyeq F(g_1, \ldots, g_n)})
$$

Fig. 18. $\text{GSL}^{\varepsilon}_{\text{Ref}}$: Initial evidence for gradual labels

With concretization of security type, we can now define security type precision.

Definition A.22 (Interval and Type Evidence Precision).

(1) $\imath_1$ is less imprecise than $\imath_2$, notation $\imath_1 \sqsubseteq \imath_2$, if and only if $\gamma_{\varepsilon_\ell}(\imath_1) \subseteq \gamma_{\varepsilon_\ell}(\imath_2)$; inductively:

$$
\frac{\ell_3 \preccurlyeq \ell_1 \qquad \ell_2 \preccurlyeq \ell_4}{[\ell_1, \ell_2] \sqsubseteq [\ell_3, \ell_4]}
$$

(2) E_1 is less imprecise than E_2, notation $E_1 \sqsubseteq E_2$, if and only if $\gamma_E(E_1) \subseteq \gamma_E(E_2)$; inductively:

$$
\frac{\imath_1 \sqsubseteq \imath_2}{\mathsf{Bool}_{\imath_1} \sqsubseteq \mathsf{Bool}_{\imath_2}} \qquad \frac{E_{11} \sqsubseteq E_{21} \qquad E_{12} \sqsubseteq E_{22} \qquad \imath_1 \sqsubseteq \imath_2 \qquad \imath'_1 \sqsubseteq \imath'_2}{E_{11} \xrightarrow{\imath'_1}_{\imath_1} E_{12} \sqsubseteq E_{21} \xrightarrow{\imath'_2}_{\imath_2} E_{22}} \qquad \frac{\imath_1 \sqsubseteq \imath_2 \qquad E_1 \sqsubseteq E_2}{\mathsf{Ref}_{\imath_1}\ E_1 \sqsubseteq \mathsf{Ref}_{\imath_2}\ E_2}
$$

A.6 GSL_{Ref}: Translation to $\text{GSL}^{\varepsilon}_{\text{Ref}}$

Figure 18 presents the initial evidence function for consistent label ordering. The initial evidence function for consistent subtyping is presented in Figure 19 using the following definition of operation pattern:

Definition A.23 (Operation pattern).

$$
\begin{array}{rcll}
P^T \in \textsc{GPattern}, P^\ell \in \textsc{LPattern} & & & \\
P^T & ::= & \_ \mid P^T\ op^T\ P^T & \text{(pattern on types)} \\
op^T & ::= & \dot{\curlyvee} \mid \dot{\curlywedge} \mid \sqcap & \text{(operations on types)} \\
P^\ell & ::= & \_ \mid P^\ell\ op^\ell\ P^\ell & \text{(pattern on labels)} \\
op^\ell & ::= & \curlyvee \mid \curlywedge \mid \sqcap & \text{(operations on labels)}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{liftP}(\_) & = \_ \\
\mathit{liftP}(P_1^T \dot{\vee} P_2^T) & = \mathit{liftP}(P_1^T) \curlyvee \mathit{liftP}(P_2^T) \\
\mathit{liftP}(P_1^T \dot{\wedge} P_2^T) & = \mathit{liftP}(P_1^T) \curlywedge \mathit{liftP}(P_2^T) \\
\mathit{liftP}(P_1^T \sqcap P_2^T) & = \mathit{liftP}(P_1^T) \sqcap \mathit{liftP}(P_2^T) \\
\mathit{invert}(\_) & = \_ \\
\mathit{invert}(P_1^T \dot{\vee} P_2^T) & = \mathit{invert}(P_1^T) \dot{\wedge} \mathit{invert}(P_2^T) \\
\mathit{invert}(P_1^T \dot{\wedge} P_2^T) & = \mathit{invert}(P_1^T) \dot{\vee} \mathit{invert}(P_2^T) \\
\mathit{invert}(P_1^T \sqcap P_2^T) & = \mathit{invert}(P_1^T) \sqcap \mathit{invert}(P_2^T) \\
\mathit{tomeet}(\_) & = \_ \\
\mathit{tomeet}(P_1^T \dot{\vee} P_2^T) & = \mathit{tomeet}(P_1^T) \sqcap \mathit{tomeet}(P_2^T) \\
\mathit{tomeet}(P_1^T \dot{\wedge} P_2^T) & = \mathit{tomeet}(P_1^T) \sqcap \mathit{tomeet}(P_2^T) \\
\mathit{tomeet}(P_1^T \sqcap P_2^T) & = \mathit{tomeet}(P_1^T) \sqcap \mathit{tomeet}(P_2^T)
\end{array}
$$

$$
\frac{\mathcal{G}[\![\widetilde{\mathit{liftP}(G_1)(\overline{\ell_i}) <: \mathit{liftP}(G_2)(\overline{\ell_j})}]\!] = \langle \iota_1, \iota_2 \rangle}{\mathcal{G}[\![\widetilde{G_1(\overline{\mathsf{Bool}_{g_i}}) \preccurlyeq G_2(\overline{\mathsf{Bool}_{g_j}})}]\!] = \langle \mathsf{Bool}_{\iota_1}, \mathsf{Bool}_{\iota_2} \rangle}
$$

$$
\frac{\begin{array}{c}
\mathcal{G}[\![\widetilde{\mathit{invert}(G_2)(\overline{U_{j1}}) <: \mathit{invert}(G_1)(\overline{U_{i1}})}]\!] = \langle E'_{21}, E'_{11} \rangle \qquad \mathcal{G}[\![\widetilde{G_1(\overline{U_{i2}}) <: G_2(\overline{U_{j2}})}]\!] = \langle E_{12}, E_{22} \rangle \\
\mathcal{G}[\![\widetilde{\mathit{liftP}(G_1)(\overline{\ell_{i1}}) <: \mathit{liftP}(G_2)(\overline{\ell_{j1}})}]\!] = \langle \iota_{11}, \iota_{12} \rangle \\
\mathcal{G}[\![\widetilde{\mathit{liftP}(\mathit{invert}(G_2))(\overline{\ell_{j2}}) <: \mathit{liftP}(\mathit{invert}(G_1))(\overline{\ell_{i2}})}]\!] = \langle \iota_{22}, \iota_{21} \rangle
\end{array}}{\mathcal{G}[\![\widetilde{G_1(\overline{U_{i1} \xrightarrow{g_{i2}}_{g_{i1}} U_{i2}}) <: G_2(\overline{U_{j1} \xrightarrow{g_{j2}}_{g_{j1}} U_{j2}})}]\!] = \langle E_{11} \xrightarrow{\iota_{21}}_{\iota_{11}} E_{12}, E_{21} \xrightarrow{\iota_{22}}_{\iota_{12}} E_{22} \rangle}
$$

$$
\frac{\begin{array}{c}
\mathcal{G}[\![\widetilde{\mathit{liftP}(G_1)(\overline{\ell_i}) <: \mathit{liftP}(G_2)(\overline{\ell_j})}]\!] = \langle \iota_1, \iota_2 \rangle \\
\mathcal{G}[\![\widetilde{\mathit{tomeet}(G_1)(\overline{U_i}) <: \mathit{tomeet}(G_2)(\overline{U_j})}]\!] = \langle E_1, E_2 \rangle \\
\mathcal{G}[\![\widetilde{\mathit{tomeet}(G_2)(\overline{U_j}) <: \mathit{tomeet}(G_1)(\overline{U_i})}]\!] = \langle E'_2, E'_1 \rangle
\end{array}}{\mathcal{G}[\![\widetilde{G_1(\overline{\mathsf{Ref}_{g_i}\ U_i}) <: G_2(\overline{\mathsf{Ref}_{g_j}\ U_j})}]\!] = \langle \mathsf{Ref}_{\iota_1}\ E_1 \sqcap E'_1, \mathsf{Ref}_{\iota_2}\ E_2 \sqcap E'_2 \rangle}
$$

where $G_1 : \textsc{GLabel}^n \to \textsc{GLabel}$ and $G_2 : \textsc{GLabel}^m \to \textsc{GLabel}$, and $G_1(x_1, ..., x_n) = P_1^T(x_1, ..., x_n)$, $G_2(x_1, ..., x_n) = P_2^T(x_1, ..., x_m)$.

$$
\mathcal{G}^{\cup}(\widetilde{F(U_1, ..., U_n)}) = \mathcal{G}[\![\widetilde{F(U_1, ..., U_n) <: F(U_1, ..., U_n)}]\!]
$$

Fig. 19. $\mathsf{GSL}^{\varepsilon}_{\mathsf{Ref}}$: Initial evidence for gradual types