Quest Complete: The Holy Grail of Gradual Security

TIANYU CHEN and JEREMY G. SIEK, Indiana University, USA

Languages with gradual information-flow control combine static and dynamic techniques to prevent security leaks. Gradual languages should satisfy the gradual guarantee: programs that only differ in the precision of their type annotations should behave the same modulo cast errors. Unfortunately, Toro et al. [2018] identify a tension between the gradual guarantee and information security; they were unable to satisfy both properties in the language $\mathrm{GSL}_{\mathsf{Ref}}$ and had to settle for only satisfying information-flow security. Azevedo de Amorim et al. [2020] show that by sacrificing type-guided classification, one obtains a language that satisfies both noninterference and the gradual guarantee. Bichhawat et al. [2021] show that both properties can be satisfied by sacrificing the no-sensitive-upgrade mechanism, replacing it with a static analysis.

In this paper we present a language design, $\lambda^{\star}_{\mathtt{IFC}}$, that satisfies both noninterference and the gradual guarantee without making any sacrifices. We keep the type-guided classification of $\mathrm{GSL}_{\mathsf{Ref}}$ and use the standard no-sensitive-upgrade mechanism to prevent implicit flows through mutable references. The key to the design of $\lambda^{\star}_{\mathtt{IFC}}$ is to walk back the decision in $\mathrm{GSL}_{\mathsf{Ref}}$ to include the unknown label $\star$ among the runtime security labels. We give a formal definition of $\lambda^{\star}_{\mathtt{IFC}}$, prove the gradual guarantee, and prove noninterference. Of technical note, the semantics of $\lambda^{\star}_{\mathtt{IFC}}$ is the first gradual information-flow control language to be specified using coercion calculi (a la Henglein), thereby expanding the coercion-based theory of gradual typing.

CCS Concepts: • **Theory of computation**; • **Security and privacy → Formal security models**; • **Software and its engineering → Formal software verification**; **Semantics**;

Additional Key Words and Phrases: gradual typing, information flow security, machine-checked proofs, Agda

ACM Reference Format:
Tianyu Chen and Jeremy G. Siek. 2024. Quest Complete: The Holy Grail of Gradual Security. *Proc. ACM Program. Lang.* 8, PLDI, Article 212 (June 2024), 24 pages. https://doi.org/10.1145/3656442

1 INTRODUCTION

Information-flow control (IFC) ensures that information transfers within a program adhere to a security policy, e.g., by preventing high-security data from flowing to a low-security channel. This adherence can be enforced statically using a type system [Myers 1999; Myers and Liskov 1997; Volpano et al. 1996], or dynamically using runtime monitoring [Askarov and Sabelfeld 2009; Austin and Flanagan 2009; Austin et al. 2017; Devriese and Piessens 2010; Stefan et al. 2011; Xiang and Chong 2021], or using static analysis to pre-compute information that facilitates runtime monitoring [Chandra and Franz 2007; Le Guernic 2007; Le Guernic and Jensen 2005; Moore and Chong 2011; Russo and Sabelfeld 2010; Shroff et al. 2007]. The static and dynamic approaches have complementary strengths and weaknesses; the dynamic approach requires less effort from the programmer while the static approach provides stronger guarantees and less runtime overhead.

Taking inspiration from gradual typing [Siek and Taha 2006, 2007], researchers have explored how to give programmers seamless control over which parts of the program are secured statically versus dynamically. In general, gradually typed languages support the seamless transition between

Authors' address: Tianyu Chen, chen512@iu.edu; Jeremy G. Siek, jsiek@indiana.edu, Indiana University, Bloomington, Indiana, USA, 47408.

ACM 2475-1421/2024/6-ART212
https://doi.org/10.1145/3656442

static and dynamic enforcement through the precision of type annotations. The programmer can choose when it is appropriate to increase the precision of the type annotations and put in the effort to pass the static checks and when it is appropriate to reduce the precision of type annotations, deferring the enforcement to runtime. The main property of gradually typed languages is the *gradual guarantee* [Siek et al. 2015], which states that removing type annotations should not change the runtime behavior. Adding type annotations should also result in the same behavior except that it may introduce more trapped errors because those new type annotations may contain mistakes.

The main challenge in the design of gradually typed languages is controlling the flow of values (and information) between the static and dynamic regions of code, which is accomplished using runtime casts. Typically source programs are compiled to an intermediate language, called a cast calculus, that includes explicit syntax for runtime casts. Disney and Flanagan [2011] design a cast calculus with IFC for a pure lambda calculus and prove noninterference. Fennell and Thiemann [2013] design a cast calculus named ML-GS with mutable references using the no-sensitive-upgrade (NSU) runtime checks of Austin and Flanagan [2009]. Fennell and Thiemann [2015] design a cast calculus for an imperative, object-oriented language.

Similar to gradual security-typed languages, Hybrid LIO (HLIO) [Buiras et al. 2015] also supports the choice of static or dynamic IFC in different parts of a single program. By default the checking is static, but a programmer can insert a `defer` clause to say that the security constraints should be checked at runtime. In gradual security-typed languages, the switch between static and dynamic is directed by types, so unlike in HLIO, the developer does not need to embed explicit casts or `defer` into the program. Moreover, the gradual guarantee relates the runtime behavior of programs that differ in the precision of type annotations, but there is no comparable theorem about adding and removing `defer` in HLIO.

Since the formulation of the gradual guarantee as a criterion for gradually typed languages [Siek et al. 2015], researchers have explored the feasibility of satisfying both the gradual guarantee and noninterference. Toro et al. [2018] identify a tension between the gradual guarantee and security enforcement. They analyze the semantics of runtime casts through the lens of Abstracting Gradual Typing [Garcia et al. 2016] and propose a type-driven semantics for gradual security. However, Toro et al. [2018] discover counterexamples to the gradual guarantee in the GSL_{Ref} language. They conjecture that it is not possible to enforce noninterference and satisfy the gradual guarantee.

Azevedo de Amorim et al. [2020] conjecture one possible source of the tension: the type-guided classification performed in GSL_{Ref} [Toro et al. 2018]. They propose a new gradually typed language, GLIO, which sacrifices type-guided classification. They prove that GLIO satisfies both noninterference and the gradual guarantee using a denotational semantics. Bichhawat et al. [2021] conjecture that *NSU checking* could be another possible source of the tension. As an alternative, they propose a hybrid approach that leverages static analysis ahead of program execution to determine the write effects in untaken branches. They study a simple imperative language with first-order stores and prove both noninterference and the gradual guarantee.

Contrary to the prior work, we show that one does not need to give up on type-guided classification or NSU checking to resolve the tension. Instead, the tension can be resolved by walking back a design choice in GSL_{Ref}, which was to allow $\star$ as a runtime security label. For example, in GSL_{Ref} one can write a literal such as $\texttt{true}_{\star}$ in a program, and at runtime the literal becomes a value of unknown security level. That design was unusual because the unknown type $\star$ is traditionally used in gradual languages to represent the lack of static information, not the lack of dynamic information. The design is also unusual when compared to dynamic systems for IFC, as those systems do not use an unknown security level [Askarov and Sabelfeld 2009; Austin and Flanagan 2009; Austin et al. 2017; Devriese and Piessens 2010; Stefan et al. 2011].

Table 1. Proposed sources of tension between security and the gradual guarantee

Language	Security (noninterference)	Gradual Guarantee	Type-guided classification	NSU checking	Runtime security labels
GSL_{Ref}	✓ Yes	✗ No	✓ Yes	✓ Yes	{low, high, ⋆}
GLIO	✓ Yes	✓ Yes	✗ No	✓ Yes	{low, high}
WHILE^{G}	✓ Yes	✓ Yes	✓ Yes	✗ No	{low, high, ⋆}
$\lambda^{\star}_{\text{IFC}}$ (this paper)	✓ Yes	✓ Yes	✓ Yes	✓ Yes	{low, high}

One might think that allowing ⋆ as a label on literals and therefore on values is necessary so that programmers can run legacy code (without any security annotations) in a gradual language, by making ⋆ the default label for literals. However, prior information-flow languages use low security as the default security label for literals [Myers et al. 2006] and for good reasons. The security of a literal is something that only the programmer can know. That is, the identification of high-security data in a program must be considered as an input to an information flow system, and not something that can or should be inferred. When migrating legacy code into a system that supports secure information flow, a necessary part of the process for the programmer is to identify whether there is any high-security information in the legacy code. Our choice of low as the default label is because most literals (if not all) in real programs are low security. In fact, it is bad practice to embed high-security literals, such as passwords, in program text.

In our design, runtime security labels do not include ⋆, only low and high.[1] On the other hand, to support gradual typing, the security labels in a type annotation may include ⋆. Surprisingly, we find that removing ⋆ from the runtime labels is sufficient to reclaim the gradual guarantee, without sacrificing type-guided classification as in GLIO or NSU checking as in WHILE^{G}. This finding is the primary contribution of this paper. In our design, the security level of a literal defaults to low, similar to systems like Jif [Myers et al. 2006] and GLIO, but different from GSL_{Ref} and WHILE^{G}. We propose a new gradual, security-typed language $\lambda^{\star}_{\text{IFC}}$, which (1) enforces information flow security, (2) satisfies the gradual guarantee, (3) enjoys type-guided classification, and (4) utilizes NSU checking to enforce implicit flows through the heap with no static analysis required.

The semantics of $\lambda^{\star}_{\text{IFC}}$ is given by translation to a new security cast calculus λ^{c}_{IFC}, for which we define a syntax, type system, and operational semantics. We compile $\lambda^{\star}_{\text{IFC}}$ into λ^{c}_{IFC} in a type-preserving way. In λ^{c}_{IFC}, *security coercions* serve as our runtime security monitor, in which we adapt ideas from the Coercion Calculus [Henglein 1994; Herman et al. 2010] to IFC.

Compared to prior work on gradual IFC languages, the λ^{c}_{IFC} cast calculus supports an additional feature called *blame tracking* [Findler and Felleisen 2002]. Blame tracking is important because it enables modular runtime error messages, e.g., they play an important role in production-quality languages such as Typed Racket [Tobin-Hochstadt and Felleisen 2008; Wilson et al. 2018].

Compared to prior work on gradual IFC based on abstracting gradual typing (AGT) [Toro et al. 2018], our use of a cast calculus makes it clear where in the program there is runtime overhead from dynamic checking (the casts). This is important for the programmer to know because one may wish to avoid runtime overhead in hot regions of a programs. In our translation from $\lambda^{\star}_{\text{IFC}}$ to λ^{c}_{IFC}, casts are only inserted where there is insufficient information during compilation to decide whether a security policy is enforced or not. In particular, casts are not inserted in statically typed regions. In contrast, the AGT mechanism for dynamic checking (called evidence) is attached to most nodes in the syntax tree.

[1] Of course, any lattice of security labels could be used in place of low and high.

This paper makes the following technical contributions:

- Identify the real cause of the tension between information flow security and the gradual guarantee in GSL_{Ref}: the inclusion of $\star$ in the runtime security levels (§ 2).
- A coercion calculus for security labels (§ 3) and a coercion calculus for secure values (§ 4). The two coercion calculi serve as our runtime IFC monitor. Our paper is the first work to apply the coercion calculus to an IFC setting.
- A cast calculus λ^{c}_{IFC} with IFC that defines the dynamic semantics of $\lambda^{\star}_{\text{IFC}}$ (§ 5). The proofs of (1) the gradual guarantee (§ 6.4), (2) compilation from $\lambda^{\star}_{\text{IFC}}$ to λ^{c}_{IFC} preserves types (§ 6.2), and (3) type safety of λ^{c}_{IFC} (§ 5.2.1) are mechanized in Agda.
- A proof of noninterference for $\lambda^{\star}_{\text{IFC}}$ (§ 6.3) through the simulation between its cast calculus λ^{c}_{IFC} and a dynamic IFC programming language (§ 5.2.3).
- The first design of a gradual security-typed language with type-guided classification that satisfies the gradual guarantee (§ 6).
- We mechanize the proof of the gradual guarantee in the Agda proof assistant.

The Appendix is in the supplementary material of this paper. The Agda code is available at:

https://github.com/Gradual-Typing/LambdaIFCStar at release `v2.0 (PLDI 2024)`

2 $\lambda^{\star}_{\text{IFC}}$ IN ACTION

In this section we present example programs that demonstrate how $\lambda^{\star}_{\text{IFC}}$ enables a gradual, smooth transition between static and dynamic information-flow control, while supporting type-based reasoning and satisfying the gradual guarantee. We briefly review the basics of gradual security typing in Section 2.1. In Section 2.2, we show that the tension between security and the gradual guarantee can be achieved by removing $\star$ from the runtime security labels. In Section 2.3, we demonstrate that $\lambda^{\star}_{\text{IFC}}$ enables the same type-based reasoning capabilities as GSL_{Ref}.

For simplicity, we use the security lattice $\langle\{\texttt{high}, \texttt{low}\}, \preccurlyeq, \vee, \wedge\rangle$, where `high` is for private data and `low` is for publicly disclosable data. The ordering is standard: $\texttt{low} \preccurlyeq \texttt{high}$ and $\texttt{high} \not\preccurlyeq \texttt{low}$. So information is allowed to flow from public sources to private sinks but not the other way around. We refer to $\{\texttt{high}, \texttt{low}\}$ as *specific security labels*.

Types in $\lambda^{\star}_{\text{IFC}}$ have security labels associated with them, for example, $\texttt{Bool}_{\texttt{high}}$ is the type for booleans with high security, $\texttt{Unit}_{\texttt{low}}$ is the type for the unit value with low security, and $\texttt{Bool}_{\star}$ is the type of a boolean whose security level is unknown at compile time. We refer to $\{\texttt{high}, \texttt{low}, \star\}$ as *security labels*. We define a *precision* ordering $\sqsubseteq$ on them, where $\star \sqsubseteq g$ for any label g and $\ell \sqsubseteq \ell$ for any specific security label ℓ. The precision ordering extends to types in a natural way, so for example, $\texttt{Bool}_{\star} \sqsubseteq \texttt{Bool}_{\texttt{low}}$. Figure 18 of the Appendix gives the definition of precision on types.

To enable information-flow control, $\lambda^{\star}_{\text{IFC}}$ allows the programmer to annotate constants, mutable references, and λ-abstractions with a specific security label and ensures that if a value is annotated with `high`, it will not flow into a sink that is `low` security. If the programmer does not annotate a value with a label, $\lambda^{\star}_{\text{IFC}}$ defaults the value's label to `low`. So `true` is shorthand for $\texttt{true}_{\texttt{low}}$.

We model I/O with two functions, `user-input` and `publish`: the former returns a high-security boolean that represents sensitive input information; the latter takes a low-security boolean and publishes it into a publicly visible channel.

2.1 Reviewing the Basics of Gradual Information Flow Security, in $\lambda^{\star}_{\text{IFC}}$

In this section we review the basic concepts of gradual information flow control using $\lambda^{\star}_{\text{IFC}}$. We start with fully static $\lambda^{\star}_{\text{IFC}}$ programs and show that $\lambda^{\star}_{\text{IFC}}$ can behave like a static security-typed language, guarding against both illegal explicit and implicit flows at compile time. We then replace some security label annotations in types with $\star$, so that the programs become partially typed and

the typing information alone is insufficient to enforce IFC. We show that coercions, our runtime security monitor, are able to capture both explicit flow and implicit flow violations at runtime, preventing information leakage and enforcing security.

Gradual IFC includes static IFC. For statically typed programs, $\lambda^{\star}_{\text{IFC}}$ behaves just like a statically typed IFC language. Consider the following well-behaved $\lambda^{\star}_{\text{IFC}}$ program that takes in a high-security user input, passes it to the function `fconst` that ignores the input and returns `false`, which is then published.

```
let fconst = λ b : Bool_high. false in
let input  = user-input () in
let result = fconst input in
  publish result
```

The program type-checks and runs without error, with no need for runtime checks to enforce security. Indeed, a malicious party cannot infer anything about the high-security input because (1) the return value of `fconst` is always the same value `false` (2) the value `false` is of low security, so the explicit flow into `publish` is allowed.

If we replace `fconst` with the identity function `fid` with parameter type $\texttt{Bool}_{\texttt{low}}$, the program becomes ill-typed because the type system of $\lambda^{\star}_{\text{IFC}}$ does not allow the explicit flow from the high-security input to `fid`, as is usual for a statically typed IFC language.

```
let fid    = λ b : Bool_low . b in
let input  = user-input () in
let result = fid input in  // static error, input is high security but fid expects low
  publish result
```

Sometimes the observable behaviors of a program can depend on its branching structure. If some of the branch conditions have a data dependency on high-security input, a malicious party might be able to infer it from the observable behaviors, giving rise to illegal *implicit flows* [Denning 1976], which must be ruled out to guarantee security.

Consider the following program in which the function `flip` contains a conditional expression, whose condition is dependent on a high-security user input. Its two branches return different low-security booleans, creating a potential implicit flow from high to low:

```
let flip : Bool_high → Bool_low = λ b : Bool_high. if b then false else true in
let input  = user-input () in
let result = flip input in
  publish result
```

As is typical of statically typed IFC languages, the type system of $\lambda^{\star}_{\text{IFC}}$ rejects this program, thereby preventing an information leak through an implicit flow. To see why, note that the branch condition is of high security, so the type of the `if` expression as a whole is $\texttt{Bool}_{\texttt{high}}$. In particular, the type checker computes the security level of an `if` to be the join of its branches (both `low`) and the condition (`high`), yielding $\texttt{low} \vee \texttt{high} = \texttt{high}$. The `flip` function is expected to return $\texttt{Bool}_{\texttt{low}}$ according to its type annotation, but returns $\texttt{Bool}_{\texttt{high}}$ because of the conditional, $\texttt{high} \not\leq \texttt{low}$, so the program is ill-typed.

To summarize, $\lambda^{\star}_{\text{IFC}}$ behaves just like a static security-typed language in the above examples. When everything is statically typed, the type system of $\lambda^{\star}_{\text{IFC}}$ guards against illegal information flows, whether explicit or implicit.

Gradual IFC enables a mixture of static and dynamic IFC. We have seen that security labels (`low` and `high`) can appear in type annotations in a program, such as $\texttt{Bool}_{\texttt{low}}$ and $\texttt{Bool}_{\texttt{high}}$. $\lambda^{\star}_{\text{IFC}}$ also

provides the *unknown security label*, written ⋆, for use in type annotations. We explain how the unknown security label works in the following discussion.

We return to the `fconst` example except this time the type of parameter `b` is $\texttt{Bool}_\star$.

```
1  let fconst = λ b : Bool⋆ . false in
2  let input  = user-input () in
3  let result = fconst input in
4    publish result
```

The type system of $\lambda^\star_{\texttt{IFC}}$ accepts this program because, in the call `fconst input`, it allows an implicit conversion from the type of `input`, which is $\texttt{Bool}_{\texttt{high}}$, to the parameter type $\texttt{Bool}_\star$. This program runs to completion and publishes `false`.

Now suppose we again replace `fconst` with `fid`, but keep the parameter type of $\texttt{Bool}_\star$.

```
1  let fid    = λ b : Bool⋆ . b in
2  let input  = user-input () in
3  let result = fid input in
4    publish result
```

The type system of $\lambda^\star_{\texttt{IFC}}$ still accepts this program. The type of `result` changes to $\texttt{Bool}_\star$ but in the call `publish result`, the type system allows an implicit conversion from $\texttt{Bool}_\star$ to $\texttt{Bool}_{\texttt{low}}$. The security leak in this program is not caught statically; instead it is caught dynamically.

The dynamic semantics of $\lambda^\star_{\texttt{IFC}}$ is defined by compilation into $\lambda^c_{\texttt{IFC}}$ by inserting casts. In $\lambda^c_{\texttt{IFC}}$, explicit casts are represented as *security coercions* that monitor the flow of information. We use the standard syntax for coercions [Henglein 1994] but with adaptations to handle IFC. A coercion whose target is ⋆ (and source is not ⋆) is an *injection*, and is indicated by an exclamation mark. A coercion whose source is ⋆ (and target is not ⋆) is a *projection*, and is indicated by a question mark. The projections perform runtime checks that may fail.

The translation from $\lambda^\star_{\texttt{IFC}}$ to $\lambda^c_{\texttt{IFC}}$ inserts a security coercion wherever an implicit cast occurred in the type checking of the $\lambda^\star_{\texttt{IFC}}$ term. Here is the result of cast insertion on the above program:

```
1  let fid    = λ b. b in
2  let input  = user-input () in
3  let result = fid (input ⟨high!⟩) in
4    publish (result ⟨low?ᵖ⟩)
```

The coercion on Line 3 (`high!`) is an injection, casting from `high` to ⋆. The coercion on line 4 ($\texttt{low}?^p$) is a projection, casting from ⋆ to `low`. At runtime, a projection checks whether the incoming value has a security level that is less than or equal to the target security level. Now suppose we run the above example with input `true`. The injection on line 3 will create an injected value $\texttt{true}_{\texttt{high}}\ \langle\texttt{high}!\rangle$. This value is passed to and returned from `fid`, and then projected to `low`. Because `high` is greater than `low`, the projection fails.

Each projection is annotated with an identifier called a blame label (p). In case a projection fails, it raises a cast error, called *blame*, that contains its blame label. In this way, the programmer knows which cast is causing the problem. This feature is often referred to as *blame tracking* [Findler and Felleisen 2002; Wadler and Findler 2009]. Blame tracking is especially useful during the software development process, but in the context of IFC, one may not want blame to be observable in a production system as it could reveal information. This can be handled by causing the program to diverge whenever blame is detected, possibly sending a private error message to the software developer.

Next we return to the `flip` example to see how gradual IFC prevents implicit flows. Suppose that we change the parameter type of the λ from $\texttt{Bool}_{\texttt{high}}$ to $\texttt{Bool}_{\star}$. The return type remains $\texttt{Bool}_{\texttt{low}}$, to conform with the signature of `publish`. Line 1 thus becomes:

```
let flip : Bool⋆ → Bool_low = λ b : Bool⋆ . if b then false else true in
```

This change makes the program well-typed in $\lambda^{\star}_{\texttt{IFC}}$. The IFC enforcement of the implicit flow is deferred until runtime because the branch condition now has type $\texttt{Bool}_{\star}$, with an unknown security level. Next we discuss the runtime behavior of this program.

The result of cast insertion on this program is the following $\lambda^{c}_{\texttt{IFC}}$ term:

```
1 let flip   = λ b. ((if⋆ b then (false ⟨low!⟩) else (true ⟨low!⟩)) ⟨low ?p⟩) in
2 let input  = user-input () in
3 let result = flip (input ⟨high!⟩) in
4   publish result
```

where the `if` is changed to `if⋆` because the condition expression has static security level $\star$. The type checking rule for `if⋆` requires the two branches to have security level $\star$ and the security level of the `if⋆` is a whole is also $\star$. If we run the program with `true` or `false` as input, the $\lambda^{c}_{\texttt{IFC}}$ term reduces to `blame` p with either input, thus capturing the illegal implicit flow. Consider running this program with input `true`. First, the λ is bound to `flip` and the input `true` is bound to the `input` variable. We then inject the input, producing the value $(\texttt{true}_{\texttt{high}}\ \langle \texttt{high}!\rangle)$. We call `flip` and the `if⋆` branches on this boolean, evaluating the "then" branch to the result $(\texttt{false}_{\texttt{low}}\ \langle \texttt{low}!\rangle)$ and then, to protect against implicit flows, the `if⋆` upgrades the result to `high` to match the runtime security level of the condition $(\texttt{true}_{\texttt{high}}\ \langle \texttt{high}!\rangle)$, producing $(\texttt{false}_{\texttt{low}}\ \langle \uparrow; \texttt{high}!\rangle)$. The subtype coercion $\uparrow$ sends the security level of `false` from `low` to `high`. The last step in the body of `flip` is to apply the coercion $\texttt{low}?^{p}$ to the value $(\texttt{false}_{\texttt{low}}\ \langle \uparrow; \texttt{high}!\rangle)$, which errors because `high` is greater than `low`.

2.2 Implicit Flow, NSU Checks, Unknown Security, and the Gradual Guarantee

The tension between information-flow security and the gradual guarantee arises from an interaction between implicit flows and the use of no-sensitive-upgrade checks to guard writes to mutable references. In brief, when $\star$ is allowed as a runtime security label, some NSU checks have to conservatively trigger an error to preserve noninterference, even though no error would occur if the label was instead `high`. But the gradual guarantee says that if a program with a precise annotation runs without error, it should also run without error when that annotation is changed to $\star$.

In preparation to discuss this scenario in more detail, we review the no-sensitive-upgrade technique [Austin and Flanagan 2009] that protects against illegal implicit flows through writes to mutable references. We then show how allowing $\star$ as a runtime security label leads to a situation where a language designer is forced to choose between noninterference and the gradual guarantee. Finally, we show how this problem is resolved in the $\lambda^{\star}_{\texttt{IFC}}$ language by walking back the choice of allowing $\star$ as a runtime security label.

The main idea of no-sensitive-upgrades is to prevent data leaks through the mutable references by terminating execution whenever the program attempts to modify a low-security heap cell in a high-security execution context. In $\lambda^{\star}_{\texttt{IFC}}$, NSU checks happen at runtime when type information is insufficient to statically decide whether a heap-modifying operation is secure or not. Consider the following well-typed program in $\lambda^{\star}_{\texttt{IFC}}$:

```
1 let input : Bool⋆ = user-input () in
2 let a     = ref low  true in
3 let _     = if input then a := false else a := true in
4   publish (! a)
```

The assignments to a in the two branches try to write different low-security booleans into the cell at the address in a, depending on a branch condition whose security level is statically unknown. However, at runtime the user-input function returns a high-security boolean, so the branch condition is actually high security, and if the writes were successful, the program would leak information via an implicit flow. Fortunately, if we run this program, it reduces to blame regardless of the input. The way NSU checking works in $\lambda^{\star}_{\mathtt{IFC}}$ is that a security level is associated with the current program counter. At the point of every write that requires an NSU, the system projects the program counter's security level to the level of the memory location, making sure that the later is at least as high as the former. In the above example, the NSU check fails because the program counter's security is high but the write is to a low security location.

In $\mathrm{GSL}_{\mathsf{Ref}}$ [Toro et al. 2018], the dynamic enforcement of IFC through the heap is also based on NSU checks. Consider the following pair of programs adapted from Section 6.3 of their paper. The program on the left is derived from the program on the right by replacing some of the high annotations with the unknown label ⋆. Both variants of the program type check but the more precise variant runs to completion while the less precise variant triggers an error, thus violating the gradual guarantee. Let us examine their runtime behavior in further detail.

Left: less precise, more dynamic

```
let x = user-input () in                          1
let y = ref Bool⋆ true⋆ in                        2
  if x then (y := false_high) else ()             3
```

Right: more precise, more static

```
let x = user-input () in
let y = ref Bool_high true_high in
  if x then (y := false_high) else ()
```

The program on the right runs without error in $\mathrm{GSL}_{\mathsf{Ref}}$ because, at the assignment on line 3, variable y references a memory cell of high security and the PC's security level is also high, so the assignment is allowed.

In contrast, when the program on the left is run with input $\mathtt{true}_{\mathtt{high}}$, the assignment is conservatively rejected by the NSU check. This is because $\mathrm{GSL}_{\mathsf{Ref}}$ considers ⋆ as corresponding to the interval [low, high], and the lower bound of this interval is not greater than or equal to the high PC label. So we see that this more precise program (right) runs successfully while the less precise one (left) errors in $\mathrm{GSL}_{\mathsf{Ref}}$.

In $\lambda^{\star}_{\mathtt{IFC}}$, the ⋆ security label can be used in type annotations, as one would expect of a gradually-typed language, but ⋆ is not allowed as a runtime security label and therefore also not allowed as a label on program literals and other introduction forms. The following adapts the above examples from $\mathrm{GSL}_{\mathsf{Ref}}$ to $\lambda^{\star}_{\mathtt{IFC}}$. The fully static variant on the right is nearly identical to its $\mathrm{GSL}_{\mathsf{Ref}}$ counterpart. To obtain the less-precise program on the left we change the type annotation on variable y to model the similar loss of precision in the $\mathrm{GSL}_{\mathsf{Ref}}$ counterpart. We do not change the labels on the ref or true to ⋆ because that is not allowed in $\lambda^{\star}_{\mathtt{IFC}}$ as we just mentioned.

```
let x = user-input () in                          1
let y : (Ref Bool⋆)⋆ = ref high true_high in      2
  if x then (y := false_high) else ()             3
```

```
let x = user-input () in
let y : (Ref Bool_high)_high = ref high true_high in
  if x then (y := false_high) else ()
```

Branching on high-security input and assigning to a high-security memory location should be allowed. Indeed, both variants reduce to the unit value regardless of the input, thereby not violating the gradual guarantee. The fully annotated version (right) evaluates to unit without any overhead from runtime checking.

In the less-precise program (left), the type annotation $(\mathtt{Ref}\ \mathtt{Bool}_{\star})_{\star}$ replaces $(\mathtt{Ref}\ \mathtt{Bool}_{\mathtt{high}})_{\mathtt{high}}$, meaning that both the security level of the memory and the security of the reference itself are statically unknown and should be checked at runtime. When executing the program, at the assignment on line 3, an NSU check happens and the assignment to high-security memory under high PC is allowed. As a result, the less-precise program also evaluates successfully to unit.

One might worry that the less precise program has a heavy annotation burden. However, as we mentioned, the default security label is low so the programmer does not have to label constants in $\lambda^{\star}_{\text{IFC}}$. So we can remove the labels on constants to obtain the following program, which also reduces successfully to unit:

```
1  let x = user-input () in
2  let y : (Ref Bool⋆)⋆ = ref high true in
3    if x then (y := false) else ()
```

The low-security true is classified as high security during reference creation because the security level of the cell is high (line 2). Similarly, during assignment the false is also classified as high security because the security level of the cell (line 3). Assigning to a high-security memory cell is allowed under a high PC by the NSU check, so the program evaluates successfully to unit.

One might think that requiring the programmer to annotate the reference creation with high (line 2) is still a burden and that GSL_{Ref} is better in this regard. However, while GSL_{Ref} allows the unannotated version of this program to compile, it errors during program execution. We believe that it is better to require the programmer to annotate references during program development than to have the programs compile but then fail during program execution, perhaps only detected after the program is deployed.

2.3 Type-Based Reasoning in $\lambda^{\star}_{\text{IFC}}$

Type-based reasoning in gradual IFC languages arises from two language design choices: vigilance and type-guided classification. Vigilance gives us type-based reasoning for explicit flows, while type-guided classification provides type-based reasoning about implicit flows. We show in this section that $\lambda^{\star}_{\text{IFC}}$ is both vigilant and performs typed-guided classification, so it enables type-based reasoning in the sense of Toro et al. [2018].

2.3.1 $\lambda^{\star}_{\text{IFC}}$ *is Vigilant.* A language with casts is *vigilant* if it checks whether all the casts that are applied to the same value are consistent with each other, and triggers an error if they are not.

Toro et al. [2018] present the following example to demonstrate how vigilance is needed for type-based reasoning and free theorems in the sense of Wadler [1989]. The example involves casts from low (line 3, the label annotation on 5_{low}) to high (line 1, the type annotation Int_{high} in the signature of mix) and then back to low via the unknown security level $\star$ (line 2, the nested type annotations $\text{Int}_{\star}$ and Int_{low}):

```
1  let mix : Int_low → Int_high → Int_low =
2      λ pub priv . if pub < (priv : Int⋆ : Int_low) then 1 else 2 in
3  mix 1_low 5_low
```

The type signature of mix should guarantee the free theorem that either (1) the result of mix, which is low security, never depends on the high-security priv argument or (2) mix produces a runtime error. In this case, the output of mix does depend on priv via an implicit flow, so the free theorem says that an error should be triggered at runtime. Let us focus on the three casts applied to 5_{low}, where the first cast sends the security level from low to high because of the type annotation on line 1, the second cast is an injection due to the first type annotation on line 2, and the third cast is a projection to low due to the second type annotation on line 2:

$$5_{\text{low}} \left\langle \text{low} \Rightarrow \text{high} \right\rangle \left\langle \text{high} \Rightarrow \star \right\rangle \left\langle \star \Rightarrow \text{low} \right\rangle$$

In $\lambda^{\star}_{\text{IFC}}$, these casts produce the sequence of coercions: $5_{\text{low}} \left\langle \uparrow ; \text{high}! ; \text{low}?^{p} \right\rangle$, which trigger an error when high collides with low, blaming label p.

Similarly, the interval refinement mechanism of GSL_{Ref} detects the conflict between the intermediate cast to high and the final cast to low. Surprisingly, in GLIO and in systems prior to GSL_{Ref} [Disney and Flanagan 2011; Fennell and Thiemann 2013], the program runs successfully and produces 1_{low} because they are forgetful [Greenberg 2014] regarding the intermediate cast of 5_{low} to high and only check that 5_{low} can be cast to low[2].

2.3.2 $\lambda^{\star}_{\text{IFC}}$ Performs Typed-Guided Classification. A gradual IFC language employs *typed-guided classification* if the security-level of a value can be changed when the value flows through a cast.

The following example from Section 2 of Toro et al. [2018] demonstrates how type-guided classification interacts with implicit flow and type-based reasoning. Type-based reasoning tells us that the `smix` function below should either fail or return a value that does not depend on its high-security parameter `priv`. However, `smix` calls `mix` and there is an implicit flow from `priv` to the result value, so this program should fail.

```
1 let mix  : Int_low → Int_⋆    → Int_low = λ pub priv. if pub < priv then 1 else 2 in
2 let smix : Int_low → Int_high → Int_low = λ pub priv. mix pub priv in
3 smix 1_low 5_low
```

In $\lambda^{\star}_{\text{IFC}}$ the program produces an error as type-based reasoning suggests. Security coercions in λ^{c}_{IFC} classify values, so when 5_{low} is passed into `smix` and then `mix`, it is wrapped in a coercion: $5_{\text{low}}\ \langle\uparrow; \text{high}!\rangle$ and is classified as high-security. Consequently, the `if` reduces to its then-branch protected with high. This implicit flow affects the result value of the then-branch, by (1) inserting a subtype coercion before the injection and (2) promoting the source of the injection to high to preserve types. So the result of the `if` is $1_{\text{low}}\ \langle\uparrow; \text{high}!\rangle$. The injection, whose source is high, collides with a projection to low (to match the Int_{low} return type of `mix`), causing the program to error as expected, blaming the projection.

3 A COERCION CALCULUS FOR SECURITY LABELS

In this section we present a coercion calculus on security labels. We show that we can use coercion composition and stamping to represent explicit flows and implicit flows respectively (Section 3.1). We define how these coercions act on security labels by defining a language of label expressions whose meaning is defined by a reduction relation (Section 3.2). Finally, we explore meta-theoretic properties of this coercion calculus (Section 3.3), establishing the intuition that the security coercion calculus can be used to enforce gradual IFC, while satisfying the gradual guarantee.

As we have seen in Section 2, gradual information flows can be modeled as casts. The cast sequence $\text{high} \Rightarrow \star \Rightarrow \text{low}$ should be statically accepted but dynamically rejected, while the sequence $\text{low} \Rightarrow \star \Rightarrow \text{high}$ should be statically and dynamically accepted, promoting the security of data to high. Such sequences of casts can be arbitrarily long (for example, $\text{low} \Rightarrow \text{high} \Rightarrow \star \Rightarrow \star \Rightarrow \text{low}$), which motivates us to model the casts on security labels as coercions. In $\lambda^{\star}_{\text{IFC}}$, the source security label of a coercion sequence comes from literals, while the sink is whatever security level that the observer has: for example, the `publish` function of Section 2 is of low. Coercions can be easily sequenced and composed. Checking information flow at runtime is accomplished by reducing coercion sequences to their normal forms.

There are two noteworthy benefits of the coercion representation for IFC. First, coercions can be used to represent NSU checking while satisfying the gradual guarantee. In brief, whenever a memory location is written to, the current PC is coerced to the security level of that location. We are going to formally introduce label expressions as our representation for PC in Section 3.2 and

[2]Those systems still satisfy noninterference, because the labels *on values* track their security similar to fully dynamic IFC. It is just that the security labels on *type annotations* will not trigger errors at runtime, even if they are inconsistent.

$$
\begin{array}{rrcl}
\text{specific security labels} & \ell & \in & \{\mathtt{low}, \mathtt{high}\} \\
\text{security labels} & g & ::= & \star \mid \ell \\
\text{blame labels} & p, q & & \\
\text{security coercions} & c, d & ::= & \mathbf{id}(g) \mid \uparrow \mid \ell\,! \mid \ell\,?^p \mid \bot^p \\
\text{coercion sequences} & \bar{c}, \bar{d} & ::= & \mathbf{id}(g) \mid \bot^p\ g_1\ g_2 \mid \bar{c}; c
\end{array}
$$

$\boxed{\vdash c : g_1 \Rightarrow g_2}$

$$
\frac{}{\vdash \mathbf{id}(g) : g \Rightarrow g} \qquad \frac{}{\vdash \uparrow : \mathtt{low} \Rightarrow \mathtt{high}} \qquad \frac{}{\vdash \ell\,! : \ell \Rightarrow \star} \qquad \frac{}{\vdash \ell\,?^p : \star \Rightarrow \ell} \qquad \frac{}{\vdash \bot^p : \mathtt{high} \Rightarrow \mathtt{low}}
$$

$\boxed{\vdash \bar{c} : g_1 \Rightarrow g_2}$

$$
\frac{}{\vdash \mathbf{id}(g) : g \Rightarrow g} \qquad \frac{\vdash \bar{c} : g_1 \Rightarrow g_2 \qquad \vdash c : g_2 \Rightarrow g_3}{\vdash \bar{c}; c : g_1 \Rightarrow g_3} \qquad \frac{}{\vdash \bot^p\ g_1\ g_2 : g_1 \Rightarrow g_2}
$$

$\boxed{\mathbf{NF}\ \bar{c}}$

$$
\frac{}{\mathbf{NF}\ \mathbf{id}(g)} \qquad \frac{}{\mathbf{NF}\ \mathbf{id}(\star); \ell\,?^p} \qquad \frac{\mathbf{NF}\ \bar{c}}{\mathbf{NF}\ \bar{c}; \ell\,!} \qquad \frac{\mathbf{NF}\ \bar{c}}{\mathbf{NF}\ \bar{c}; \uparrow}
$$

$\boxed{c\ ;\ c \longrightarrow c}$

$$
?\text{-}id\ \frac{}{\ell\,!; \ell\,?^p \longrightarrow \mathbf{id}(\ell)} \qquad ?\text{-}\uparrow\ \frac{}{\mathtt{low}\,!; \mathtt{high}\,?^p \longrightarrow \uparrow} \qquad ?\text{-}\bot\ \frac{}{\mathtt{high}\,!; \mathtt{low}\,?^p \longrightarrow \bot^p}
$$

$\boxed{\bar{c} \longrightarrow \bar{d}}$

$$
id\ \frac{\mathbf{NF}\ \bar{c}}{\bar{c}; \mathbf{id}(g) \longrightarrow \bar{c}} \qquad \bot\ \frac{\mathbf{NF}\ \bar{c} \qquad \vdash \bar{c} : g_1 \Rightarrow g_2}{\bar{c}; \bot^p \longrightarrow \bot^p\ g_1\ \mathtt{low}} \qquad \xi\text{-}\bot\ \frac{\vdash c : g_2 \Rightarrow g_3}{\bot^p\ g_1\ g_2; c \longrightarrow \bot^p\ g_1\ g_3}
$$

$$
\xi_L\ \frac{\bar{c} \longrightarrow \bar{d}}{\bar{c}; c \longrightarrow \bar{d}; c} \qquad \xi_R\ \frac{\mathbf{NF}\ \bar{c} \qquad c; d \longrightarrow c'}{\bar{c}; c; d \longrightarrow \bar{c}; c'}
$$

Fig. 1. Syntax, typing, normal forms, and semantics of security coercions and coercion sequences

discuss NSU in detail in Section 5.1.2. Second, the coercion representation benefits mechanization because it enables modular reasoning. The main simulation lemma (Lemma 13) depends on the simulation results of coercion sequences and label expressions, which are stated as separate lemmas and reasoned independently in our Agda code.

The syntax and typing for security coercions and coercion sequences is defined in Figure 1. A security label is either $\mathtt{low}$, $\mathtt{high}$, or statically unknown ($\star$). There are five security coercions: identity ($\mathbf{id}(g)$), subtype ($\uparrow$), injection ($\ell\,!$), and projection ($\ell\,?^p$), and blame ($\bot^p$). Projection, which corresponds to the notion of a runtime check, is the only one responsible for blame, so it carries a blame label p. A coercion sequence $\bar{c}$ starts with either success $\mathbf{id}(g)$ or failure ($\bot^p\ g_1\ g_2$). Each coercion has a source and target type $g_1 \Rightarrow g_2$. The $\mathbf{id}(g)$ casts the label g to itself; $\uparrow$ promotes security from $\mathtt{low}$ to $\mathtt{high}$; injection casts to $\star$ from a specific label ℓ and projection does the opposite. Appending a single coercion to a coercion sequence makes the target security label that of the single coercion.

Information flow is enforced in the reduction semantics of security coercions, shown in Figure 1. Injection followed by projection to the same label collapses to the identity (*?-id*). Flowing from $\mathtt{low}$ to $\mathtt{high}$ is allowed, so an injection from $\mathtt{low}$ followed by a projection to $\mathtt{high}$ collapses into the $\uparrow$ coercion (?- $\uparrow$). An information flow from $\mathtt{high}$ to $\mathtt{low}$ is prohibited, so an injection to $\mathtt{high}$ followed by a projection to $\mathtt{low}$ triggers an error that blames the projection (?-$\bot$). The predicate **NF** that specifies the normal forms of coercion sequences. The reduction rules for coercion sequences are also defined in Figure 1. Appending $\mathbf{id}(g)$ onto a coercion sequence reduces to the

$\boxed{\bar{c} \fatsemi \bar{c} = \bar{c}}$

$$\bar{c} \fatsemi \bot^{p}\ g_2\ g_3 = \bot^{p}\ g_1\ g_3 \quad \text{,where} \vdash \bar{c} : g_1 \Rightarrow g_2$$
$$\bar{c} \fatsemi \mathbf{id}(g) = \bar{c};\ \mathbf{id}(g)$$
$$\bar{c}_1 \fatsemi (\bar{c}_2;\ c) = (\bar{c}_1 \fatsemi \bar{c}_2);\ c$$

$\boxed{\mathit{stamp}\ \bar{c}\ \ell = \bar{c}}$

$$\mathit{stamp}\ \bar{c}\ \texttt{low} = \bar{c}$$
$$\mathit{stamp}\ \mathbf{id}(\texttt{low})\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow$$
$$\mathit{stamp}\ \mathbf{id}(\texttt{high})\ \texttt{high} = \mathbf{id}(\texttt{high})$$
$$\mathit{stamp}\ (\mathbf{id}(\texttt{low});\ \texttt{low}!)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$
$$\mathit{stamp}\ (\mathbf{id}(\texttt{high});\ \texttt{high}!)\ \texttt{high} = \mathbf{id}(\texttt{high});\ \texttt{high}!$$
$$\mathit{stamp}\ (\mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$
$$\mathit{stamp}\ (\mathbf{id}(\texttt{low});\ \uparrow)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow$$

$\boxed{\mathit{stamp}!\ \bar{c}\ \ell = \bar{c}}$

$$\mathit{stamp}!\ \bar{c}\ \texttt{low} = \begin{cases} \bar{c} & \text{, if} \vdash \bar{c} : \ell \Rightarrow \star \\ \bar{c};\ \ell_2\,! & \text{, if} \vdash \bar{c} : \ell_1 \Rightarrow \ell_2 \end{cases}$$
$$\mathit{stamp}!\ \mathbf{id}(\texttt{low})\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$
$$\mathit{stamp}!\ \mathbf{id}(\texttt{high})\ \texttt{high} = \mathbf{id}(\texttt{high});\ \texttt{high}!$$
$$\mathit{stamp}!\ (\mathbf{id}(\texttt{low});\ \texttt{low}!)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$
$$\mathit{stamp}!\ (\mathbf{id}(\texttt{high});\ \texttt{high}!)\ \texttt{high} = \mathbf{id}(\texttt{high});\ \texttt{high}!$$
$$\mathit{stamp}!\ (\mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$
$$\mathit{stamp}!\ (\mathbf{id}(\texttt{low});\ \uparrow)\ \texttt{high} = \mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}!$$

Fig. 2. Composing and stamping coercions

that sequence (*id*). The failure coercions annihilate the other coercions in the sequence ($\bot$ and ξ-$\bot$). We choose the evaluation order in a coercion sequence to be from left to right (ξ_L and ξ_R), because that corresponds to the direction of information flow from source to sink: in the example above, $\texttt{low} \Rightarrow \texttt{high} \Rightarrow \star \Rightarrow \star \Rightarrow \texttt{low}$, we validate that low can flow to high before we check the flow from high through $\star$ to low.

3.1 Monitoring Explicit and Implicit Flows

We model explicit flow using security coercions. We can compose two coercion sequences ($\bar{c} \fatsemi \bar{d}$), where $\bar{c} : g_1 \Rightarrow g_2$ and $\bar{d} : g_2 \Rightarrow g_3$, to form a flow from g_1 to g_3, which is defined in Figure 2.

The stamping operation captures the intuition of an implicit flow from the security level ℓ' to a coercion sequence $\bar{c}$. We define the stamping operation in Figure 2 as two functions, $\mathit{stamp}(\bar{c}, \ell)$ and $\mathit{stamp}!(\bar{c}, \ell)$. Both function require $\bar{c}$ to be in normal form and that its source label is not $\star$. The $\mathit{stamp}!$ operator promotes the security of the coercion $\bar{c}$ to be at least ℓ and then injects the coercion if necessary, while *stamp* only promotes the security but does not inject. These stamping operations satisfy the gradual guarantee, because when stamping on a more precise coercion sequence and a less precise coercion sequence, stamping preserves the precision relation (Section 3.3.2) between them (Lemma 9). The stamping operations of coercion sequences are used in the stamping operations of (1) label expressions, which are our representation of PC and (2) values in the cast calculus. Those three types of stamping together formalize the notion of implicit flow in $\lambda^{\star}_{\texttt{IFC}}$.

3.2 Security Label Expressions

In this section we introduce security label expressions, which we use to model the security level of the PC. Security label expressions are crucial for implementing NSU checking in a way that satisfies the gradual guarantee.

A label expression is either (1) a specific security label, (2) blame (to signify an error), or (3) a coercion applied to a label expression (Figure 3). A label expression is in normal form (**NF**) if it is either (1) a specific security label or (2) an irreducible coercion applied to a specific security label. (A coercion is irreducible if it is a non-identity coercion in normal form). *PC* ranges over label expressions in normal form. The reduction relation for label expressions steps a label expression

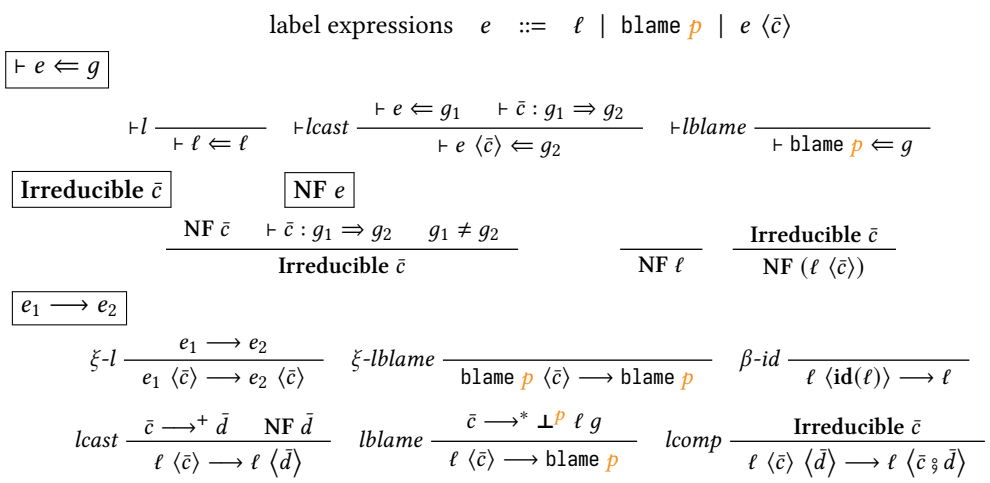

Fig. 3. Syntax, typing, normal forms, and semantics of label expressions

towards its normal form. The idea is that given a label expression of the form $e\ \langle \bar{d} \rangle$, we first reduce e to normal form and then apply the coercion $\bar{d}$. For example, if e reduces to a label wrapped in coercion $\ell\ \langle \bar{c} \rangle$, then the *lcomp* rule says to reduce by composing the two coercions, producing $\ell\ \langle \bar{c} \fatsemi \bar{d} \rangle$. Furthermore, in a label expression of the form $e\ \langle \bar{d} \rangle$, the coercion $\bar{d}$ may also need to be reduced, which is accomplished by the *lcast* rule that refers to the reduction relation for coercion sequences (Figure 1). If the coercion reduces to an identity, then the coercion application goes away (β-*id*), whereas if the coercion reduces to a failure, then the label expression reduces to blame (*lblame*).

The stamping and security level operators for label expressions are defined in Figure 13 of the Appendix. They both require their input to be in normal form, which can be either (1) a specific security label ℓ, or (2) a label wrapped with an irreducible coercion sequence $\ell\ \langle \bar{c} \rangle$. For (1), stamping `low` with `high` results in `low` $\langle \uparrow \rangle$, otherwise the label expression remains unchanged; for (2), we directly stamp the coercion sequence using *stamp* for coercion sequences defined in Figure 2. The definition of *stamp!* is analogous, except that it turns to the *stamp!* operator of coercion sequences. The security level operator $|-|$ is defined such that (1) a specific security label indicates the security level for itself and (2) the security of the coercion sequence $|\bar{c}|$ records the security level for $\ell\ \langle \bar{c} \rangle$.

We describe in Section 5.1 how label expressions are used to implement NSU checks, which enforce the heap policy for write operations.

3.3 Properties of Coercion Calculus on Labels

3.3.1 Tracking Information Flow via Composition and Stamping. We show that composing coercions tracks explicit flows, while stamping tracks implicit flows. We first define of the security level of a coercion sequence with the $|-|$ operator [3]:

Definition 1 (Security level of a coercion). *Given a coercion sequence $\bar{c}$ in normal form with type $\vdash \bar{c} : \ell \Rightarrow g$, then its security is given by the following $|-|$ operator:*

$$|\mathbf{id}(\ell)| = \ell \qquad |\mathbf{id}(\ell);\ \ell\,!| = \ell \qquad |\mathbf{id}(\texttt{low});\ \uparrow;\ \texttt{high}\,!| = \texttt{high} \qquad |\mathbf{id}(\texttt{low});\ \uparrow\ | = \texttt{high}$$

[3] In a larger lattice, the subtype coercion $\uparrow$ would be parameterized by two labels: $\vdash (\uparrow^{\ell_2}_{\ell_1}) : \ell_1 \Rightarrow \ell_2$ where $\ell_1 \prec \ell_2$. The security would be the greater one: $|\uparrow^{\ell_2}_{\ell_1}| = \ell_2$. Here the lattice only consists of `low` and `high`, so $|\uparrow| = |\uparrow^{\texttt{high}}_{\texttt{low}}| = \texttt{high}$.

We reason about explicit flows first. If we compose one coercion sequence with another and then reduce the result to normal form, the security of the resulting coercion sequence should be greater than or equal to that of the first sequence.

Lemma 2 (Composition models explicit flow).
If **NF** $\bar{c}$ *and* $\bar{c} \fatsemi \bar{d} \longrightarrow^* \bar{c}'$ *and* **NF** $\bar{c}'$, *then* $|\bar{c}| \preccurlyeq |\bar{c}'|$.

Next we show that stamping models implicit flow correctly, promoting the security of the stamped coercion by joining it with the stamped label:

Lemma 3 (Stamping models implicit flow).
If **NF** $\bar{c}$, *then* $|stamp\ \bar{c}\ \ell| = |\bar{c}| \vee \ell$ *and* $|stamp!\ \bar{c}\ \ell| = |\bar{c}| \vee \ell$.

3.3.2 Simulation Between More and Less Precise Coercion Sequences. Our goal is to prove the gradual guarantee for $\lambda^{\star}_{\mathtt{IFC}}$. The proof depends on a simulation lemma between more and less precise terms of the cast calculus $\lambda^{c}_{\mathtt{IFC}}$. We use coercion sequences as the IFC monitor in $\lambda^{c}_{\mathtt{IFC}}$. Reducing a coercion sequence can result in a blame which errors the program. So we would like to prove that the simulation lemma holds for the coercion calculus on security labels. The precision relation on security coercions is defined in Figure 20 of the Appendix. The precision relation between two coercion sequences $\bar{c}, \bar{d}$ takes the form $\vdash \bar{c} \sqsubseteq \bar{d}$. Recall that the gradual guarantee states that replacing type annotations with $\star$ (decreasing type precision) should result in the same value for a correctly running program while adding annotations (increasing type precision) may trigger more runtime errors. The precision relation is a syntactical characterization of the runtime behaviors of programs of different type precision. We explain the intuition with two examples.

Example 4. Consider the following two programs that are related by precision because the first one has a $\star$ annotation where the second one has high.

$$\mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\star} : \mathtt{Bool}_{\star} \qquad \text{and} \qquad \mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\mathtt{high}} : \mathtt{Bool}_{\star}$$

At runtime, the less precise program on the left produces value $(\mathtt{true}\ \langle \mathbf{id}(\mathtt{low}); \mathtt{low}\,! \rangle)$ and the more precise program on the right produces $(\mathtt{true}\ \langle \mathbf{id}(\mathtt{low}); \uparrow; \mathtt{high}\,! \rangle)$. The trues are straightforwardly related; we need to show the two coercion sequences are also related:

$$\vdash \mathbf{id}(\mathtt{low}); \mathtt{low}\,! \sqsubseteq \mathbf{id}(\mathtt{low}); \uparrow; \mathtt{high}\,!$$

Starting at the beginning of the two sequences, we have $\mathbf{id}(\mathtt{low}) \sqsubseteq \mathbf{id}(\mathtt{low})$ because $\sqsubseteq$ is reflexive. Next we have $\mathtt{low}\,! \sqsubseteq \uparrow$, which makes sense because the source and targets of the two coercions are related by precision: $\mathtt{low} \sqsubseteq \mathtt{low}$ and $\star \sqsubseteq \mathtt{high}$. Finally, the coercion $\mathtt{high}\,!$ can be added to the end of the more precise sequence because both its source and target type or more precise than the target of the left-hand sequence. That is, $\star \sqsubseteq \mathtt{high}$ and $\star \sqsubseteq \star$. The injections at the ends of the two sequences, $\mathtt{low}\,!$ and $\mathtt{high}\,!$, cannot be directly related via precision because $\mathtt{low} \not\sqsubseteq \mathtt{high}$. Instead, $\mathtt{low}\,!$ is related with $\uparrow$. This underlines the indispensability of explicit subtype coercion $\uparrow$ for the purposes of proving the gradual guarantee.

Example 5. Next we consider an example where the less precise program produces a value but the more precise program encounters an error. This situation is allowed by the gradual guarantee, but the opposite one is not. We extend the example with a cast to low on the more precise side.

$$\mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\star} : \mathtt{Bool}_{\star} : \mathtt{Bool}_{\star} \qquad \text{and} \qquad \mathtt{true}_{\mathtt{low}} : \mathtt{Bool}_{\mathtt{high}} : \mathtt{Bool}_{\star} : \mathtt{Bool}_{\mathtt{low}}$$

The first program again produces value $(\mathtt{true}\ \langle \mathbf{id}(\mathtt{low}); \mathtt{low}\,! \rangle)$. The second, on the other hand, reduces to $(\mathtt{true}\ \langle \mathbf{id}(\mathtt{low}); \uparrow; \mathtt{high}\,!; \mathtt{low}\,?^{p} \rangle) \longrightarrow^* (\mathtt{true}\ \langle \bot^{p}\ \mathtt{low}\ \mathtt{low} \rangle)$, because of the contradicting annotations high and low (note that high is in the middle of the sequence and both the

$$
\begin{array}{rrcl}
\text{base types} & \iota & ::= & \texttt{Unit} \mid \texttt{Bool} \\
\text{raw types} & T, S & ::= & \iota \mid A \xrightarrow{g^c} B \mid \texttt{Ref}\ (T_g) \\
\text{types} & A, B & ::= & T_g \\
\text{raw coercions} & c_r, d_r & ::= & \mathbf{id}(\iota) \mid \mathbf{Ref}\ \boldsymbol{c}\ \boldsymbol{d} \mid (\bar{d}, \boldsymbol{c} \rightarrow \boldsymbol{d}) \\
\text{coercions} & \boldsymbol{c}, \boldsymbol{d} & ::= & c_r, \bar{c}
\end{array}
$$

$\boxed{V\ \langle \boldsymbol{c} \rangle \longrightarrow M}$

$$
\textit{cast}\ \frac{\bar{c} \longrightarrow^+ \bar{d} \quad \mathbf{NF}\ \bar{d}}{V_r\ \langle c_r, \bar{c} \rangle \longrightarrow V_r\ \langle c_r, \bar{d} \rangle} \qquad \textit{cast-blame}\ \frac{\bar{c} \longrightarrow^* \bot^p\ g_1\ g_2}{V_r\ \langle c_r, \bar{c} \rangle \longrightarrow \texttt{blame}\ p}
$$

$$
\textit{cast-id}\ \frac{}{V_r\ \langle \mathbf{id}(\iota), \mathbf{id}(g) \rangle \longrightarrow V_r} \qquad \textit{cast-comp}\ \frac{\textbf{Irreducible } \boldsymbol{c}}{V_r\ \langle \boldsymbol{c} \rangle\ \langle \boldsymbol{d} \rangle \longrightarrow V_r\ \langle \boldsymbol{c} \fatsemi \boldsymbol{d} \rangle}
$$

Fig. 4. Syntax and semantics of coercions on values.

source and target labels of the blame coercion are low so that types are preserved). The failure is then propagated out and the term further reduces to blame p. The precision of coercion sequences relates $\bot$ on the right-hand side to any coercion sequence on the left so long as the respective source and target types are related via precision, in this case $\texttt{low} \sqsubseteq \texttt{low}$ and $\star \sqsubseteq \texttt{low}$.

Consider the security levels (Definition 1) of both sides of Example 4, which are low on the less precise side and high on the more precise side. We observe that $\lambda^{\star}_{\texttt{IFC}}$ programs related by precision may produce values of different security: a less precise value may have lower security than a more precise value. Indeed, we prove the following for coercions in normal form:

Lemma 6 (Security is monotonic with respect to precision). *Suppose* $\mathbf{NF}\ \bar{c}$ *and* $\mathbf{NF}\ \bar{d}$. *If* $\vdash \bar{c} \sqsubseteq \bar{d}$, *then* $|\bar{c}| \preccurlyeq |\bar{d}|$.

Next we prove a catch-up lemma for coercion sequences, where the less precise side catches up with a more precise sequence that is in normal form. The proof is by casing on $\mathbf{NF}\ \bar{d}$ first and then performing induction on the precision relation in each case.

Lemma 7 (Catching up to a more precise coercion sequence). *If* $\mathbf{NF}\ \bar{d}$ *and* $\vdash \bar{c} \sqsubseteq \bar{d}$, *there exists* $\bar{c}'$ *such that* $\bar{c} \longrightarrow^* \bar{c}'$ *and* $\vdash \bar{c}' \sqsubseteq \bar{d}$.

Using Lemma 7, we then prove the following simulation lemma for coercion sequences:

Lemma 8 (Simulation between related coercion sequences). *If* $\vdash \bar{c} \sqsubseteq \bar{d}$ *and* $\bar{d} \longrightarrow \bar{d}'$, *there exists* $\bar{c}'$ *such that* $\bar{c} \longrightarrow^* \bar{c}'$ *and* $\vdash \bar{c}' \sqsubseteq \bar{d}'$.

We also prove that stamping on coercion sequences preserves precision:

Lemma 9 (Stamping preserves precision of coercion sequences). *If* $\vdash \bar{c} \sqsubseteq \bar{d}$, *then* $\vdash \mathit{stamp}\ \bar{c}\ \ell \sqsubseteq \mathit{stamp}\ \bar{d}\ \ell$ *and* $\vdash \mathit{stamp!}\ \bar{c}\ \ell_1 \sqsubseteq \mathit{stamp!}\ \bar{d}\ \ell_2$ *and* $\vdash \mathit{stamp!}\ \bar{c}\ \ell_1 \sqsubseteq \mathit{stamp}\ \bar{d}\ \ell_2$ *if* $\ell_1 \preccurlyeq \ell_2$.

4 A SECURITY COERCION CALCULUS ON VALUES

In this section we define a second coercion calculus whose purpose is to cast a program value from one type to another type. We use this coercion calculus as the representation of casts in the intermediate language $\lambda^{c}_{\texttt{IFC}}$. These coercions on values make use of the coercions on security labels that we defined in Section 3 because the types in $\lambda^{c}_{\texttt{IFC}}$ are annotated with security labels, as is usual for a static and gradually-typed IFC languages.

We begin with the definition of types in Figure 4, which is standard for gradual security type systems: each type has a security label ascription on it, which is either a specific label ℓ or $\star$. Function types have one extra label g^c, which is a static approximation of the security level of the PC while executing the function body. In a reference type $(\texttt{Ref}\ T_{\hat{g}})_g$, the label $\hat{g}$ of the referenced type also doubles as the security level of the memory cell.

The syntax and semantics for coercions on values is defined in Figure 4. Each coercion $\boldsymbol{c}$ consists of a raw coercion c_r that casts the type of the value and the label coercion $\bar{c}$ that casts the security label of the value. There are three kinds of raw coercions: identity coercions $\mathbf{id}(g)$, coercions between reference types ($\mathbf{Ref}\ \boldsymbol{c}\ \boldsymbol{d}$), and coercions between function types $(\bar{d},\ \boldsymbol{c} \rightarrow \boldsymbol{d})$. In the coercion on functions, the $\bar{d}$ casts the PC of the function. (The syntax for values is not defined until the next section, so here we remark that V ranges over values, which can either be a raw value V_r (constant, address, or λ) or an irreducible coercion applied to a raw value: $V_r\ \langle \boldsymbol{c} \rangle$, where there is no M such that $V_r\ \langle \boldsymbol{c} \rangle \longrightarrow M$. The definition of irreducible coercion is in Figure 15 of the Appendix.)

The reduction rules in Figure 4 apply a coercion to a value, yielding a value or triggering blame. The *cast* rule normalizes the coercion $\bar{c}$ on the security label. If it normalizes to a failure coercion, the rule *cast-blame* triggers blame. We reduce identity coercions using rule *cast-id*. Finally, if the value is wrapped with an irreducible coercion, we compose the coercion with the coercion being applied (rule *cast-comp*). The composition operator $- \fatsemi -$ is also defined in Figure 15 of the Appendix; the intuition of the composition operator is that $V_r\ \langle \boldsymbol{c} \rangle\ \langle \boldsymbol{d} \rangle$ must be contextual equivalent to $V_r\ \langle \boldsymbol{c} \fatsemi \boldsymbol{d} \rangle$. There are no reduction rules specific to reference coercions $\mathbf{Ref}\ \boldsymbol{c}\ \boldsymbol{d}$ or function coercions $(\bar{d},\ \boldsymbol{c} \rightarrow \boldsymbol{d})$ because they are irreducible coercions that wrap a value. Their action occurs when the value is used in an elimination form such as in a function call or a read or write to the reference, which we explain in the next section.

5 THE CAST CALCULUS $\lambda^c_{\texttt{IFC}}$

In this section we define the cast calculus $\lambda^c_{\texttt{IFC}}$ (§ 5.1), prove that $\lambda^c_{\texttt{IFC}}$ is type-safe (§ 5.2.1), and prove the main simulation lemma (§ 5.2.2) that is needed for the proof of the gradual guarantee. We conclude this section with a proof that $\lambda^c_{\texttt{IFC}}$ satisfies noninterference (§ 5.2.3).

5.1 Syntax, Typing, and Operational Semantics of $\lambda^c_{\texttt{IFC}}$

5.1.1 Syntax and Typing of $\lambda^c_{\texttt{IFC}}$. As usual, the cast calculus $\lambda^c_{\texttt{IFC}}$ is a statically-typed language that includes an explicit term for casts, written $M\ \langle \boldsymbol{c} \rangle$, where M is a term and $\boldsymbol{c}$ is a coercion to be applied to the value of M. Furthermore, many of the operators in $\lambda^c_{\texttt{IFC}}$ have two variants, a "static" one for when the pertinent security label is statically known and the "dynamic" one for when the security label is statically unknown. The operational semantics of the "dynamic" variants involve runtime checking. The syntax and typing rules for $\lambda^c_{\texttt{IFC}}$ are shown in Figure 5 (excerpt of typing rules, full version in Figure 12 in the Appendix) and described in the following paragraphs.

A value is a raw value (constant, address or λ) or an irreducible coercion applied to a raw value.

The typing rules are syntax-directed. The typing judgment is of the form $\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A$, which says that we are type-checking $\lambda^c_{\texttt{IFC}}$ term M against the expected type A, where Γ is the typing context, Σ is the heap typing context, g is the security label that PC is typed at, and ℓ is the security level of PC. Both Σ and the security level ℓ play a role during runtime. The security level of the PC is constrained in rule $\vdash$*prot*, which is for the protection term that arises during reduction (we are going to discuss this rule momentarily). In the premises for sub-terms that do not immediately reduce, such as the body of a λ and the branches of an `if`, we universally quantify the security level (as in $\forall \ell$), which helps us prove that compilation preserves types (Theorem 15). The heap context Σ is mostly standard: looking up $\Sigma(\hat{\ell}, n)$, where n is the index in part of the heap

$$
\begin{array}{rrcl}
\text{variables} & x, y, z & & \\
\text{constants} & k & \in & \{\texttt{unit}, \texttt{true}, \texttt{false}\} \\
\text{terms} & L, M, N & ::= & x \mid \$k \mid \texttt{addr}\, n \mid \lambda x.\, N \mid \texttt{app}\, L\, M\, A\, B\, \ell \mid \texttt{app}\star\, L\, M\, A\, T \\
& & \mid & \texttt{if}\, L\, A\, \ell\, M\, N \mid \texttt{if}\star\, L\, T\, M\, N \mid \texttt{let}\, x{=}M{:}A\, \texttt{in}\, N \\
& & \mid & \texttt{ref}\, \ell\, M \mid \texttt{ref?}^{p}\, \ell\, M \mid\, !\, M\, A\, \ell \mid\, !\star\, M\, T \\
& & \mid & \texttt{assign}\, L\, M\, T\, \hat{\ell}\, \ell \mid \texttt{assign?}^{p}\, L\, M\, T\, \hat{g} \\
& & \mid & \texttt{prot}\, PC\, \ell\, M\, A \mid M\, \langle c \rangle \mid \texttt{blame}\, p \\
\text{raw values} & V_r, W_r & ::= & \$k \mid \texttt{addr}\, n \mid \lambda x.\, N \\
\text{values} & V, W & ::= & V_r \mid V_r\, \langle c \rangle \text{, where } \textbf{Irreducible } \boldsymbol{c}
\end{array}
$$

$\boxed{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A}$

$$
\vdash app\ \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash L \Leftarrow (A \xrightarrow{\ell' \vee \ell} B)_{\ell} \quad \Gamma; \Sigma; \ell'; \ell'' \vdash M \Leftarrow A \quad C = stamp\ B\ \ell}{\Gamma; \Sigma; \ell'; \ell'' \vdash \texttt{app}\, L\, M\, A\, B\, \ell \Leftarrow C}
\qquad
\vdash app\star\ \frac{\Gamma; \Sigma; g; \ell \vdash L \Leftarrow (A \xrightarrow{\star} (T_{\star}))_{\star} \quad \Gamma; \Sigma; g; \ell \vdash M \Leftarrow A}{\Gamma; \Sigma; g; \ell \vdash \texttt{app}\star\, L\, M\, A\, T \Leftarrow T_{\star}}
$$

$$
\vdash assign\ \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash L \Leftarrow (\texttt{Ref}\ T_{\hat{\ell}})_{\ell} \quad \Gamma; \Sigma; \ell'; \ell'' \vdash M \Leftarrow T_{\hat{\ell}} \quad \ell' \vee \ell \preccurlyeq \hat{\ell}}{\Gamma; \Sigma; \ell'; \ell'' \vdash \texttt{assign}\, L\, M\, T\, \hat{\ell}\, \ell \Leftarrow \texttt{Unit}_{\texttt{low}}}
\qquad
\vdash assign?\ \frac{\Gamma; \Sigma; g; \ell \vdash L \Leftarrow (\texttt{Ref}\ T_{\hat{g}})_{\star} \quad \Gamma; \Sigma; g; \ell \vdash M \Leftarrow T_{\hat{g}}}{\Gamma; \Sigma; g; \ell \vdash \texttt{assign?}^{p}\, L\, M\, T\, \hat{g} \Leftarrow \texttt{Unit}_{\texttt{low}}}
$$

$$
\vdash prot\ \frac{\Gamma; \Sigma; g'; |PC| \vdash M \Leftarrow A \quad \vdash PC \Leftarrow g' \quad \ell' \vee \ell \preccurlyeq |PC| \quad B = stamp\ A\ \ell}{\Gamma; \Sigma; g; \ell' \vdash \texttt{prot}\, PC\, \ell\, M\, A \Leftarrow B}
$$

Fig. 5. Syntax and selected typing rules of the cast calculus $\lambda^{c}_{\texttt{IFC}}$. The side condition that enforces the heap policy statically during assignment is highlighted

with security $\hat{\ell}$, gives us a raw type. Each memory cell is associated with a specific security label $\hat{\ell}$, which is specified by the programmer when that cell is created.

As the typing rules always stay in checking mode, constants, addresses, and λs do not carry any label. The security of these raw values is in their types: for example, $\texttt{addr}\, n \Leftarrow (\texttt{Ref}\ T_{\hat{\ell}})_{\ell}$ says that the security of the address n itself is ℓ and it points to a memory cell labeled $\hat{\ell}$.

The typing rules for the static variants are similar to the typing rules in a static security type system. For example, in the static function application rule $\vdash app$, both top-level labels on the function type as well as the security label that the current PC is typed at are static and the rule mirrors one in a static system. On the other hand, in the dynamic version of application rule $\vdash app\star$, both top-level labels as well as the label of the co-domain type are $\star$ and PC is allowed to be typed at $\star$, indicating the presence of injections. As another example, in the static version of memory assignment rule $\vdash assign$, all labels to perform the heap policy check, including the security of the memory address itself (ℓ), the security of the memory cell that the address references ($\hat{\ell}$), and the security of PC (ℓ') are known statically and satisfy $\ell' \vee \ell \preccurlyeq \hat{\ell}$. At runtime, this static assignment can happen directly without any runtime overhead (as is shown in the example of Section 2.2). Its dynamic counterpart rule $\vdash assign?$ does not maintain these static security invariants and thus requires runtime NSU checking, which is implemented as a projection on PC.

As we shall see in the reduction rules, the semantics of the protection term `prot` performs two things: (1) `prot` promotes the security of the value reduced from its body by level ℓ (2) it uses a new PC to reduce its body. However, the new PC cannot be any label expression. It has to be one with higher security than both the current PC and the security level ℓ. We capture this invariant

$$\boxed{M \mid \mu \mid PC \longrightarrow N \mid \mu'}$$

$$\textit{prot-ctx}\ \frac{M \mid \mu \mid PC' \longrightarrow M' \mid \mu'}{\mathtt{prot}\ PC'\ \ell\ M\ A \mid \mu \mid PC \longrightarrow \mathtt{prot}\ PC'\ \ell\ M'\ A \mid \mu'} \qquad \textit{prot-val}\ \frac{}{\mathtt{prot}\ PC'\ \ell\ V\ A \mid \mu \mid PC \longrightarrow \mathit{stamp\text{-}val}\ V\ A\ \ell \mid \mu}$$

$$\textit{cast}\ \frac{V\ \langle \boldsymbol{c} \rangle \longrightarrow M}{V\ \langle \boldsymbol{c} \rangle \mid \mu \mid PC \longrightarrow M \mid \mu} \qquad \beta\ \frac{}{\mathtt{app}\ (\lambda x.\ N)\ V\ A\ B\ \ell \mid \mu \mid PC \longrightarrow \mathtt{prot}\ (\mathit{stamp}\ PC\ \ell)\ \ell\ (N[x := V])\ B \mid \mu}$$

$$\textit{app-cast}\ \frac{\mathbf{NF}\ \bar{c} \qquad (\mathit{stamp}\ PC\ \ell)\ \langle \bar{d} \rangle \longrightarrow^* PC' \qquad V\ \langle \boldsymbol{c} \rangle \longrightarrow^* W}{\mathtt{app}\ (\lambda x.\ N\ \langle \bar{d},\ \boldsymbol{c} \rightarrow \boldsymbol{d},\ \bar{c} \rangle)\ V\ C\ D\ \ell \mid \mu \mid PC \longrightarrow \mathtt{prot}\ PC'\ \ell\ ((N[x := W])\ \langle \boldsymbol{d} \rangle)\ D \mid \mu}$$

$$\textit{app}\star\textit{-cast}\ \frac{\mathbf{NF}\ \bar{c} \qquad (\mathit{stamp!}\ PC\ |\bar{c}|)\ \langle \bar{d} \rangle \longrightarrow^* PC' \qquad V\ \langle \boldsymbol{c} \rangle \longrightarrow^* W}{\mathtt{app}\star\ (\lambda x.\ N\ \langle \bar{d},\ \boldsymbol{c} \rightarrow \boldsymbol{d},\ \bar{c} \rangle)\ V\ C\ T \mid \mu \mid PC \longrightarrow \mathtt{prot}\ PC'\ |\bar{c}|\ ((N[x := W])\ \langle \boldsymbol{d} \rangle)\ (T_\star) \mid \mu}$$

$$\textit{if-true}\ \frac{}{\mathtt{if\$true}\ A\ \ell\ M\ N \mid \mu \mid PC \longrightarrow \mathtt{prot}\ (\mathit{stamp}\ PC\ \ell)\ \ell\ M\ A \mid \mu}$$

$$\textit{if}\star\textit{-true-cast}\ \frac{\mathbf{NF}\ \bar{c}}{\mathtt{if}\star\ (\$\mathtt{true}\ \langle \mathbf{id}(\mathtt{Bool}),\ \bar{c} \rangle)\ T\ M\ N \mid \mu \mid PC \longrightarrow \mathtt{prot}\ (\mathit{stamp!}\ PC\ |\bar{c}|)\ |\bar{c}|\ M\ (T_\star) \mid \mu}$$

$$\textit{ref}\ \frac{n\ \mathbf{FreshIn}\ \mu(\ell)}{\mathtt{ref}\ \ell\ V \mid \mu \mid PC \longrightarrow \mathtt{addr}\ n \mid (\mu, \ell \mapsto n \mapsto V)} \qquad \textit{ref?}\ \frac{n\ \mathbf{FreshIn}\ \mu(\ell) \qquad PC\ \langle \star \Rightarrow^p \ell \rangle \longrightarrow^* PC'}{\mathtt{ref?}^p\ \ell\ V \mid \mu \mid PC \longrightarrow \mathtt{addr}\ n \mid (\mu, \ell \mapsto n \mapsto V)}$$

$$\textit{ref?-blame}\ \frac{PC\ \langle \star \Rightarrow^p \ell \rangle \longrightarrow^* \mathtt{blame}\ q}{\mathtt{ref?}^p\ \ell\ V \mid \mu \mid PC \longrightarrow \mathtt{blame}\ q \mid \mu} \qquad \textit{assign}\ \frac{}{\mathtt{assign}\ (\mathtt{addr}\ n)\ V\ T\ \hat{\ell}\ \ell \mid \mu \mid PC \longrightarrow \$\mathtt{unit} \mid [\hat{\ell} \mapsto n \mapsto V]\ \mu}$$

$$\textit{assign?-cast}\ \frac{\mathbf{NF}\ \bar{c} \qquad \vdash \boldsymbol{c} : T_g \Rightarrow S_{\hat{\ell}} \qquad \vdash \boldsymbol{d} : S_{\hat{\ell}} \Rightarrow T_g \qquad (\mathit{stamp!}\ PC\ |\bar{c}|)\ \langle \star \Rightarrow^p \hat{\ell} \rangle \longrightarrow^* PC' \qquad V\ \langle \boldsymbol{c} \rangle \longrightarrow^* W}{\mathtt{assign?}^p\ (\mathtt{addr}\ n\ \langle \mathbf{Ref}\ \boldsymbol{c}\ \boldsymbol{d},\ \bar{c} \rangle)\ V\ T\ g \mid \mu \mid PC \longrightarrow \$\mathtt{unit} \mid [\hat{\ell} \mapsto n \mapsto W]\ \mu}$$

$$\textit{deref}\ \frac{\mu(\hat{\ell}, n) = V}{!\ (\mathtt{addr}\ n)\ T_{\hat{\ell}}\ \ell \mid \mu \mid PC \longrightarrow \mathtt{prot}\ \_\ \ell\ V\ T_{\hat{\ell}} \mid \mu}$$

$$\textit{deref}\star\textit{-cast}\ \frac{\mathbf{NF}\ \bar{c} \qquad \vdash \boldsymbol{c} : S_\star \Rightarrow T_\ell \qquad \vdash \boldsymbol{d} : T_\ell \Rightarrow S_\star \qquad \mu(\ell, n) = V}{!\star\ (\mathtt{addr}\ n\ \langle \mathbf{Ref}\ \boldsymbol{c}\ \boldsymbol{d},\ \bar{c} \rangle)\ S \mid \mu \mid PC \longrightarrow \mathtt{prot}\ \_\ |\bar{c}|\ (V\ \langle \boldsymbol{d} \rangle)\ (S_\star) \mid \mu}$$

Fig. 6. Selected semantics rules for $\lambda^c_{\mathtt{IFC}}$. NSU checks are represented using label expressions (highlighted)

in side condition $\ell' \curlyvee \ell \preccurlyeq |PC|$ in rule $\vdash \mathit{prot}$, where ℓ' is the security of the current PC and $|PC|$ is the security for the new PC. The invariant is used in the proof of noninterference.

5.1.2 Operational Semantics for $\lambda^c_{\mathtt{IFC}}$. We show the interesting rules of the operational semantics of $\lambda^c_{\mathtt{IFC}}$ in Figure 6 (full version in Figure 16 and 17 of the Appendix). The reduction relation takes the form $M \mid \mu \mid PC \longrightarrow N \mid \mu'$, which reduces the configuration of term M and heap μ under the label expression PC to another configuration N and μ'. The heap is a map $(\ell, n) \mapsto V$, where a cell is indexed by its security level ℓ and by index n among the cells of ℓ. The predicate (n **FreshIn** $\mu(\ell)$) means that the index n is fresh (not already in use) among all cells with security ℓ; when performing a lookup, $\mu(\ell, n) = V$ retrieves the value V at index n whose security level is ℓ.

Protection terms. Following standard approaches to IFC, a protection term $\mathtt{prot}\ PC'\ \ell\ M\ A$ has two functionalities: (1) it ensures that the reduction inside M does not leak information through heap write operations (2) it promotes the security level of the computation result of M to at least level ℓ. The first functionality is achieved by switching to PC' from the current PC when reducing the body M (rule *prot-ctx*). Recall Section 5.1.1, the typing of `prot` makes sure that PC' has the correct security that is at least as secure as both PC and ℓ. The second functionality is achieved by stamping the value produced by the body of `prot` (rule *prot-val*). The stamping of values in

$\lambda^c_{\texttt{IFC}}$ (Figure 14 in the Appendix) is analogous to stamping of label expressions and turns to the stamping operation for coercions on labels (Figure 2) in a similar way. Again there are two cases, because the value is either (1) a raw value or (2) a coercion-wrapped value. Suppose the value is a raw value, if its type has `low` security and is stamped with `high`, the value becomes wrapped with a subtype coercion; otherwise the value stays unchanged. Otherwise, if the value is wrapped with an irreducible coercion, we stamp the top-level coercion sequence.

Function Application. The β rule is standard for IFC languages. It generates a `prot` term with the specific security label ℓ that comes from the label on the λ, preventing implicit flow from the function being applied through both the computation result and the heap. The *app-cast* rule applies a function wrapped in a function coercion to a value V. The application is "static", so the security level of `prot` comes from the function type just like β. The function coercion is distributed into its domain coercion $\boldsymbol{c}$, its co-domain coercion $\boldsymbol{d}$, and the coercion on PC $\bar{d}$. The coercion $\bar{c}$ is not used because the function type is fully static, so its security is already indicated by its type. The domain coercion $\boldsymbol{c}$ casts the input of the function V to W and W is substituted into the body of the λ. The substituted body goes through the co-domain cast $\boldsymbol{d}$, and is then protected by ℓ using `prot`. The stamped PC casts to PC' by $\bar{d}$ and PC' is used as the PC for `prot`. The rule *app⋆-cast* is similar to *app-cast* except for two things: (1) the PC is stamped and then injected using *stamp!* to preserve types (2) the security level of the function proxy used in protection is indicated by $|\bar{c}|$ instead, because the top-level security label of the function is statically unknown ($\star$).

If-conditional. The static rule *if-true* is standard; the `if` term reduces a `prot` whose security ℓ comes from the type of the branch condition, guarding against implicit flow. The rationale of *if⋆-true-cast* follows that of *app⋆-cast*: (1) a *stamp!* is generated to stamp and then inject the PC and (2) the security of `prot` is retrieved from the coercion in the branch condition.

NSU and heap operations. Let us consider reference creation first. A "static" reference creation is secure because its typing (rule $\vdash$*-ref*, Figure 12 of the Appendix) already enforces the heap policy. Consequently, the allocation can happen directly (rule *ref*) without any runtime checking. Rule *ref?* does the same reference creation but with NSU checking, by casting the current PC to the security ℓ of the newly created memory cell. The coerce function $(- \Rightarrow^{-} -)$ takes two security labels and a blame label to generate a coercion on labels. In this case, $\star \Rightarrow^{p} \ell$ generates $\mathbf{id}(\star)$; $\ell\,?^{p}$, which performs a projection whose target is ℓ. If the projected PC reduces to a blame, it means that NSU checking fails so we lift the blame to $\lambda^c_{\texttt{IFC}}$. Assignment follows the same pattern: a static assignment can happen directly, while `assign?` requires NSU checking, by stamping the current PC with the security indicated in the coercion and then projecting to $\hat{\ell}$, where $\hat{\ell}$ is the security of the heap cell. The input coercion $\boldsymbol{c}$ is applied before the value is stored into the cell.

The rule for static dereferencing *deref* looks up index n in all memory cells with security level $\hat{\ell}$. The value from the lookup is protected with ℓ, the top-level security label of the reference type. The PC of the `prot` does not matter, because V is already a value and will not reduce. The rule *deref⋆-cast* dereferences a reference proxy. It looks up the value V in the heap, applies the output coercion $\boldsymbol{d}$, and generates a `prot` with the security of the coercion $|\bar{c}|$. The PC of `prot` does not matter in this case either, because applying a coercion is pure and does not produce side effects.

5.2 Meta-theoretical Results of $\lambda^c_{\texttt{IFC}}$

We prove type safety for $\lambda^c_{\texttt{IFC}}$ (§ 5.2.1) and the main simulation lemma for $\lambda^c_{\texttt{IFC}}$ (§ 5.2.2) used in the proof of the gradual guarantee (§ 6.4). We also prove that $\lambda^c_{\texttt{IFC}}$ satisfies noninterference (§ 5.2.3).

5.2.1 Type Safety. We show that $\lambda^c_{\texttt{IFC}}$ is type safe by proving progress and preservation. Progress says that a well-typed $\lambda^c_{\texttt{IFC}}$ term does not get stuck. The term is either a value or a blame, which does not reduce, or the term takes one reduction step. Heap well-typedness is defined point-wise.

Theorem 10 (Progress). *Suppose PC is well-typed:* $\vdash PC \Leftarrow g$, *M is well-typed:* $\emptyset; \Sigma; g; |PC| \vdash M \Leftarrow A$, *and the heap* μ *is also well-typed:* $\Sigma \vdash \mu$. *Then either (1) M is a value or (2) M is a blame:* $M = \texttt{blame}\ p$ *or (3) M can take a reduction step:* $M \mid \mu \mid PC \longrightarrow N \mid \mu'$ *for some N and* μ'.

The operation semantics of $\lambda^c_{\texttt{IFC}}$ preserves types and the well-typedness of heap:

Theorem 11 (Preservation). *Suppose PC is well-typed:* $\vdash PC \Leftarrow g$, *M is well-typed:* $\emptyset; \Sigma; g; |PC| \vdash M \Leftarrow A$ *and the heap* μ *is also well-typed:* $\Sigma \vdash \mu$. *If* $M \mid \mu \mid PC \longrightarrow N \mid \mu'$, *there exists* Σ' *s.t* $\Sigma' \supseteq \Sigma$, $\emptyset; \Sigma'; g; |PC| \vdash N \Leftarrow A$, *and* $\Sigma' \vdash \mu'$.

5.2.2 Simulation Between $\lambda^c_{\texttt{IFC}}$ Terms of Different Precision. The main simulation lemma says that if two terms are related by *precision* and the more precise side takes one step, then the less precise side is able to multi-step and get back in sync. The precision relation is in form $\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'$, where $\Gamma; \Sigma; g; \ell$ corresponds to the typing context, heap context, type of *PC*, and security of *PC* of the less precise term M and $\Gamma'; \Sigma'; g'; \ell'$ is for those of the more precise term M'. The types of the two terms, A and A', are related by precision between types. The intuition between this precision relation is that casts are allowed to appear in different places between the more precise and the less precise $\lambda^c_{\texttt{IFC}}$ terms. Moreover, the casts must be in shapes that preserve the precision of $\lambda^\star_{\texttt{IFC}}$ (more or fewer static type annotations provided by the programmer). According to the gradual guarantee, the more precise side is allowed to signal more blames, so there is a rule ($\sqsubseteq$-*blame*) that relates $\texttt{blame}\ p$ to any term M on the less precise side as long as their types are in sync. We list selected precision rules in Figure 21 of the Appendix.

With the precision relation of $\lambda^c_{\texttt{IFC}}$ defined, we first state the catch-up lemma, which catches up to a more-precise value by multi-stepping on the less-precise side:

Lemma 12 (Catching up to more precise). *If term M and value* V' *are related by precision:*

$$\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

then there exists value V *s.t* $M \mid \mu \mid PC \longrightarrow^* V \mid \mu$ *and* $\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$.

We prove the main simulation lemma using the catch-up lemma. Heap precision is defined pointwise similar to the definition of heap well-typedness.

Lemma 13 (Simulation between more precise and less precise $\lambda^c_{\texttt{IFC}}$ terms). *Suppose PC, PC' are related by precision:* $\vdash PC \sqsubseteq PC' \Leftarrow g \sqsubseteq g'$. *Moreover suppose M, M' are related by precision:*

$$\emptyset; \emptyset; \Sigma_1; \Sigma_1'; g; g'; |PC|; |PC'| \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'$$

heap contexts Σ_1, Σ_1' *are related by precision:* $\Sigma_1 \sqsubseteq \Sigma_1'$, *the initial heaps* μ_1, μ_1' *are also related by precision:* $\Sigma_1; \Sigma_1' \vdash \mu_1 \sqsubseteq \mu_1'$.
If $M' \mid \mu_1' \mid PC' \longrightarrow N' \mid \mu_2'$, *there exists* Σ_2, Σ_2', N, μ_2 *s.t* $\Sigma_2 \supseteq \Sigma_1$, $\Sigma_2' \supseteq \Sigma_1'$, $\Sigma_2 \sqsubseteq \Sigma_2'$,

$$M \mid \mu_1 \mid PC \longrightarrow^* N \mid \mu_2$$

the resulting terms are related by precision: $\emptyset; \emptyset; \Sigma_2; \Sigma_2'; g; g'; |PC|; |PC'| \vdash N \sqsubseteq N' \Leftarrow A \sqsubseteq A'$ *and the resulting heaps are also related by precision:* $\Sigma_2; \Sigma_2' \vdash \mu_2 \sqsubseteq \mu_2'$.

5.2.3 Noninterference. We prove that $\lambda^c_{\texttt{IFC}}$ satisfies termination-insensitive noninterference in Section 11 of the Appendix. The statement of termination-insensitive noninterference says that if we run a program with different high-security inputs in two executions, then their low-security output values should be the related (e.g the same boolean):

Lemma 14 (Noninterference for $\lambda^c_{\texttt{IFC}}$). *If* M *is well-typed:* $(x{:}\texttt{Bool}_{\texttt{high}}); \emptyset; \texttt{low}; \texttt{low} \vdash M : \texttt{Bool}_{\texttt{low}}$ *and* $M[x := \$\, b_1] \mid \emptyset \mid \texttt{low} \longrightarrow^* V_1 \mid \mu_1$ *and* $M[x := \$\, b_2] \mid \emptyset \mid \texttt{low} \longrightarrow^* V_2 \mid \mu_2$, *then* $V_1 = V_2$.

$$\begin{array}{lrcl} \text{terms} & L, M, N & ::= & x \mid (\$\ k)_{\ell} \mid (\lambda^{g} x{:}A.\ N)_{\ell} \mid (L\ M)^{p} \\ & & \mid & (\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N)^{p} \mid \texttt{let}\ x = M\ \texttt{in}\ N \\ & & \mid & (\texttt{ref}\ \ell\ M)^{p} \mid\ !^{p}\ M \mid (L := M)^{p} \mid (M : A)^{p} \end{array}$$

$\boxed{\Gamma; g \vdash M : A}$

$$\vdash lam\ \frac{(\Gamma, x{:}A); g \vdash N : B}{\Gamma; g' \vdash (\lambda^{g} x{:}A.\ N)_{\ell} : (A \xrightarrow{g} B)_{\ell}} \qquad \vdash assign\ \frac{\Gamma; g' \vdash L : (\texttt{Ref}\ T_{\hat{g}})_{g} \quad \Gamma; g' \vdash M : A \quad A \lesssim T_{\hat{g}} \quad g \precsim \hat{g} \quad g' \precsim \hat{g}}{\Gamma; g' \vdash (L := M)^{p} : \texttt{Unit}_{\texttt{low}}}$$

Fig. 7. Syntax and selected typing rules of $\lambda^{\star}_{\texttt{IFC}}$ (highlighted security labels ℓ default to low if omitted)

6 THE $\lambda^{\star}_{\texttt{IFC}}$ LANGUAGE WITH GRADUAL INFORMATION-FLOW CONTROL

In this section, we first define the gradual language $\lambda^{\star}_{\texttt{IFC}}$ in Section 6.1. It is similar to GSL_{Ref} with respect to syntax and typing rules. The main syntactic difference is that in $\lambda^{\star}_{\texttt{IFC}}$, the security labels of literals and newly created memory cells default to a specific label such as low, while in GSL_{Ref} they default to a runtime unknown security level $\star$. We show that $\lambda^{\star}_{\texttt{IFC}}$ can be compiled into our cast calculus $\lambda^{c}_{\texttt{IFC}}$ and the compilation preserves types in Section 6.2. As a result, the semantics of $\lambda^{\star}_{\texttt{IFC}}$ can be defined by the operational semantics of $\lambda^{c}_{\texttt{IFC}}$. In Section 6.3, we prove the noninterference of $\lambda^{\star}_{\texttt{IFC}}$ as a corollary of the noninterference lemma for $\lambda^{c}_{\texttt{IFC}}$ and compilation preserves types. Finally, we prove the gradual guarantee for $\lambda^{\star}_{\texttt{IFC}}$ as a corollary of the main simulation lemma of $\lambda^{c}_{\texttt{IFC}}$ (Lemma 13), thus solving the tension discovered by Toro et al. [2018].

6.1 Syntax and Typing of the Surface Language $\lambda^{\star}_{\texttt{IFC}}$

The syntax and selected typing rules of $\lambda^{\star}_{\texttt{IFC}}$ are shown in Figure 7 (full version in Figure 10 of the Appendix). They are directly adapted from those of GSL_{Ref}, by changing the security labels on values to disallow the $\star$ label.

The rule $\vdash assign$ includes two side conditions $g \precsim \hat{g}$ and $g' \precsim \hat{g}$. If g, g', and $\hat{g}$ are all specific security labels, the heap policy is enforced statically, because the typing tells us that the security of the current PC as well as the memory address itself is lower than or equal to the security of the memory cell, thus no implicit flow through the heap. Indeed, we are going to see in the next section that an assignment where all three labels are statically known generates an static assign. The semantics of assign does not perform NSU because the static check on its typing rule suffices. Relating to the $\lambda^{\star}_{\texttt{IFC}}$ program at the end of Section 2.2, there is no runtime NSU check, because the program generates a static assign that enforces the heap policy statically.

6.2 Compiling $\lambda^{\star}_{\texttt{IFC}}$ to $\lambda^{c}_{\texttt{IFC}}$: Cast Insertion

The compile function takes the form $C\ M = M'$, where M is a $\lambda^{\star}_{\texttt{IFC}}$ program and M' is a $\lambda^{c}_{\texttt{IFC}}$ term. Consider the case for assignment:

$$C\ (L := M)^{p} = \begin{cases} \texttt{assign}\ (C\ L)\ ((C\ M)\ \langle \boldsymbol{c_2} \rangle)\ T\ \hat{g}\ g & \text{, if } g, g' \text{ and } \hat{g} \text{ are all specific} \\ \texttt{assign?}^{p}\ ((C\ L)\ \langle \boldsymbol{c_1} \rangle)\ ((C\ M)\ \langle \boldsymbol{c_2} \rangle)\ T\ \hat{g} & \text{, otherwise} \end{cases}$$

$$\text{where } \boldsymbol{c_1} = (\texttt{Ref}\ T_{\hat{g}})_{g} \Rightarrow^{p} (\texttt{Ref}\ T_{\hat{g}})_{\star}, \boldsymbol{c_2} = A \Rightarrow^{p} T_{\hat{g}}, \Gamma; g' \vdash L : (\texttt{Ref}\ T_{\hat{g}})_{g}, \Gamma; g' \vdash M : A$$

If g, g', and $\hat{g}$ are specific, the check for heap policy can be statically justified. We recursively compile both L and M and generate a static assign. We cast M' using coercion $\boldsymbol{c_2}$, which casts from the type of M' to the type of the memory cell. The coercion is produced by the coerce function, $(- \Rightarrow^{-} -)$, which takes two types and a blame label, returning a coercion on values by calling the

coerce function of labels on each pair of security labels inside those two types. If at least one of the three security labels is $\star$, the typing information is insufficient to justify the assignment. The compilation produces an `assign?` instead, whose semantics performs NSU checking at runtime.

Compilation from $\lambda^{\star}_{\mathtt{IFC}}$ to $\lambda^{c}_{\mathtt{IFC}}$ preserves types:

Theorem 15 (Compilation preserves types). *If* $\Gamma; g \vdash M : A$, *then* $\Gamma; \emptyset; g; \mathtt{low} \vdash C\ M \Leftarrow A$.

6.3 Noninterference for $\lambda^{\star}_{\mathtt{IFC}}$

The noninterference theorem of $\lambda^{\star}_{\mathtt{IFC}}$ is a straightforward corollary of the noninterference lemma for $\lambda^{c}_{\mathtt{IFC}}$ (Section 5.2.3) and compilation preserves types. The proof is in Section 11 of the Appendix.

Theorem 16 (Noninterference for $\lambda^{\star}_{\mathtt{IFC}}$). *Suppose a* $\lambda^{\star}_{\mathtt{IFC}}$ *program* M *is well-typed:* $(x{:}\mathtt{Bool}_{\mathtt{high}}); \mathtt{low} \vdash M : \mathtt{Bool}_{\mathtt{low}}$. *If for any boolean inputs* b_1, b_2

$$(C\ M)[x := \$\, b_1] \mid \emptyset \mid \mathtt{low} \longrightarrow^{*} V_1 \mid \mu_1 \quad \text{and} \quad (C\ M)[x := \$\, b_2] \mid \emptyset \mid \mathtt{low} \longrightarrow^{*} V_2 \mid \mu_2$$

then the resulting values $V_1 = V_2$.

6.4 The Gradual Guarantee

Finally, we state the gradual guarantee of $\lambda^{\star}_{\mathtt{IFC}}$ and prove it as a corollary of Lemma 13. The definition of term precision for $\lambda^{\star}_{\mathtt{IFC}}$ is in Figure 19 of the Appendix.

Theorem 17 (Gradual guarantee). *Suppose* M *and* M' *are well-typed terms in* $\lambda^{\star}_{\mathtt{IFC}}$ *that are related by precision, that is* $\vdash M \sqsubseteq M'$. *So* $\emptyset; \mathtt{low} \vdash M : A$, $\emptyset; \mathtt{low} \vdash M' : A'$, *and* $A \sqsubseteq A'$. *If the compilation of* M' *reduces to a value:* $C\ M' \mid \emptyset \mid \mathtt{low} \longrightarrow^{*} V' \mid \mu'$
there exists a value V *and heap* μ *s.t. the compilation of* M *reduces to* V: $C\ M \mid \emptyset \mid \mathtt{low} \longrightarrow^{*} V \mid \mu$
and the resulting values are related by precision for some Σ, Σ':

$$\emptyset; \emptyset; \Sigma; \Sigma'; \mathtt{low}; \mathtt{low}; \mathtt{low}; \mathtt{low} \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

Proof. Compilation preserves precision, so $\emptyset; \emptyset; \emptyset; \emptyset; \mathtt{low}; \mathtt{low}; \mathtt{low}; \mathtt{low} \vdash C\ M \sqsubseteq C\ M' \Leftarrow A \sqsubseteq A'$. We then proceed by induction on the reduction of $C\ M'$ to a value V', using Lemma 13 to show that $C\ M$ reduces to a corresponding term at each step. So we have $C\ M \longrightarrow^{*} N$ where $N \sqsubseteq V'$ for some N. We then apply Lemma 12 to show that N reduces to a value V where $V \sqsubseteq V'$. □

7 CONCLUSION

We presented the design of a gradual information-flow language $\lambda^{\star}_{\mathtt{IFC}}$ that satisfies both noninterference and the gradual guarantee while maintaining the principle of type-based reasoning. The key to the design of $\lambda^{\star}_{\mathtt{IFC}}$ is to walk back the decision in $\mathrm{GSL}_{\mathsf{Ref}}$ to include the unknown label $\star$ among the runtime security labels. So $\lambda^{\star}_{\mathtt{IFC}}$ takes a more standard approach to gradually-typed IFC: the $\star$ label can be used in type annotations but not as the security level of a runtime value. The $\lambda^{\star}_{\mathtt{IFC}}$ language is defined by translation to a cast calculus $\lambda^{c}_{\mathtt{IFC}}$. This intermediate language employs a coercion calculus to express the implicit conversions between more-or-less precise parts of the program. We proved that $\lambda^{\star}_{\mathtt{IFC}}$ satisfies termination-insensitive noninterference. We proved that $\lambda^{\star}_{\mathtt{IFC}}$ satisfies the gradual guarantee and mechanized the result in Agda.

DATA-AVAILABILITY STATEMENT

The Agda code of this paper is in the supplementary material [Chen 2024].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1763922.

REFERENCES

Aslan Askarov and Andrei Sabelfeld. 2009. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *2009 22nd IEEE Computer Security Foundations Symposium*. 43–59. https://doi.org/10.1109/CSF.2009.22

Thomas H Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. 113–124. https://doi.org/10.1145/1554339.1554353

Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3, Article 10 (may 2017), 56 pages. https://doi.org/10.1145/3024086

Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling noninterference and gradual typing. In *Logic in Computer Science (LICS)*. https://doi.org/10.1145/3373718.3394778

Abhishek Bichhawat, McKenna McCall, and Limin Jia. 2021. Gradual Security Types and Gradual Guarantees. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00015

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/2784731.2784758

Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 463–475. https://doi.org/10.1109/ACSAC.2007.37

Tianyu Chen. 2024. *cty12/pldi2024-ae: PLDI Release 2.* https://doi.org/10.5281/zenodo.10933110

Tianyu Chen and Jeremy G. Siek. 2022. Mechanized Noninterference for Gradual Security. arXiv:2211.15745 [cs.PL]

Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243. https://doi.org/10.1145/360051.360056

Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy*. 109–124. https://doi.org/10.1109/SP.2010.15

Tim Disney and Cormac Flanagan. 2011. Gradual Information Flow Typing. In *Workshop on Script to Program Evolution*.

L. Fennell and P. Thiemann. 2013. Gradual Security Typing with References. In *2013 IEEE 26th Computer Security Foundations Symposium*. 224–239. https://doi.org/10.1109/CSF.2013.22

Luminous Fennell and Peter Thiemann. 2015. LJGS: Gradual Security Types for Object-Oriented Languages. In *Workshop on Foundations of Computer Security (FCS)*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.9

Robert Bruce Findler and Matthias Felleisen. 2002. *Contracts for Higher-Order Functions.* Technical Report NU-CCS-02-05. Northeastern University. https://doi.org/10.1145/2502508.2502521

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL 2016)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Michael Greenberg. 2014. Space-Efficient Manifest Contracts. *CoRR* abs/1410.2813 (2014). http://arxiv.org/abs/1410.2813

Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (June 1994), 197–230. https://doi.org/10.1016/0167-6423(94)00004-2

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Gurvan Le Guernic. 2007. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. IEEE, 218–232. https://doi.org/10.1109/CSF.2007.10

Gurvan Le Guernic and Thomas Jensen. 2005. Monitoring information flow. In *Proc. Workshop on Foundations of Computer Security*. 19–30.

Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical Computer Science* 411, 19 (2010), 1974 – 1994. https://doi.org/10.1016/j.tcs.2010.01.025 Mathematical Foundations of Programming Semantics (MFPS 2006).

Scott Moore and Stephen Chong. 2011. Static analysis for efficient hybrid information-flow control. In *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE, 146–160. https://doi.org/10.1109/CSF.2011.17

Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 228–241. https://doi.org/10.1145/292540.292561

Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. *SIGOPS Oper. Syst. Rev.* 31, 5 (oct 1997), 129–142. https://doi.org/10.1145/269005.266669

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. *Jif 3.0: Java information flow.* http://www.cs.cornell.edu/jif

Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 186–199. https://doi.org/10.1109/CSF.2010.20

Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic Dependency Monitoring to Secure Information Flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. 203–217. https://doi.org/10.1109/CSF.2007.20

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (LCNS, Vol. 4609)*. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

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). https://doi.org/10.1017/S0956796816000241

Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*. 95–106. https://doi.org/10.1145/2034675.2034688

Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. 2012. Flexible dynamic information flow control in the presence of exceptions. *arXiv preprint arXiv:1207.1457* (2012).

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*. https://doi.org/10.1145/1328897.1328486

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4, Article 16 (Dec. 2018), 55 pages. https://doi.org/10.1145/3229061

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187. https://doi.org/10.3233/JCS-1996-42-304

Philip Wadler. 1989. Theorems for free!. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture* (Imperial College, London, United Kingdom). ACM, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: A User Study. In *Dynamic Languages Symposium*. https://doi.org/10.1145/3393673.3276947

Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. IEEE Press, Piscataway, NJ, USA. https://doi.org/10.1109/SP40001.2021.00002