Partial Cast Calculus for Gradual Information Security

Zechi Shen[†], Zhiwu Xu[†*], Shengchao Qin[‡], Zhong Ming[§†]
[†]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[‡]School of Computer Science and Technology, Xidian University, Xi'an, China
[§]School of Artificial Intelligence, Shenzhen Technology University, Shenzhen, China

***Abstract*—Gradual typing effectively combines the advantages of dynamic and static typing and is utilized in Information Flow Control. The gradual guarantee ensures that removing type annotations does not affect runtime behavior. However, Toro et al. identify a tension between the gradual guarantee and noninterference, leading many gradual security-typed languages to reconcile these two properties. Recently, Chen and Siek's language $\lambda^{\star}_{\mathtt{IFC}}$ achieved both properties. Nevertheless, their cast calculus suffers from scalability issues and lacks binary operators, which limits its practical use.**

In this paper, we present a partial cast calculus λ^{p} for gradual information security based on the concept of *partial label*. Partial labels serve as concise representations for security coercions and runtime labels, simplifying the definition of label-related operators and facilitating cast reductions within the semantics of λ^{p}. Consequently, our calculus effectively supports binary operators and naturally supports richer security lattices. We formally prove both noninterference and the gradual guarantee for λ^{p}. Furthermore, we present a gradual surface language λ^{g} along with a compilation from surface terms to cast-calculus terms. We prove that this compilation preserves typing, precision relations, and observational equivalence. Finally, we extend our calculus to accommodate unknown references and provide a semantic interpretation of partial labels.

***Index Terms*—Gradual Security, Gradual Typing, Information Flow Security, Partial Labels, Machine-checked Proofs**

I. Introduction

Information Flow Control (IFC) serves as a fundamental methodology for enforcing security policies within software systems. It regulates the flow of sensitive data to prevent unauthorized access or information leakage across various security levels. Various approaches can be employed to implement IFC, including static, dynamic, and hybrid methods. Static approaches leverage type-based systems [1]–[4] to detect and mitigate potential security violations at compile-time. However, these methods face challenges when applied to legacy systems or rapidly evolving software environments. In contrast, dynamic approaches [5]–[10] monitor information flows at runtime. This flexibility facilitates policy adjustments without necessitating extensive code modifications. However, it may introduce additional runtime overhead. Hybrid approaches [11]–[16] enhance runtime monitoring through the integration of static analysis techniques.

Gradual typing [17]–[20] effectively combines the advantages of dynamic and static typing, surpassing mere integration. It introduces a special annotation, $\star$, to signify a statically unknown type, enabling certain parts of a program to be dynamically typed while other parts remain statically typed. Programmers determine which parts are dynamic or static by either omitting type annotations or incorporating them as needed. The type system ensures the type safety of the statically typed parts, while runtime semantics monitors the interactions between the static and dynamic parts.

Gradual typing has been implemented in IFC, where a specific label, such as $\star$, is utilized to denote a statically unknown label. Early efforts employed dynamic security labels in conjunction with runtime monitoring; however, these approaches faced challenges in reconciling gradual typing with strong security guarantees [21]–[25]. Since the introduction of the gradual guarantee as a criterion for gradually typed languages [17], researchers have explored the feasibility of fulfilling both the gradual guarantee and noninterference properties simultaneously. However, Toro et al. [26] identified an inherent tension between the dynamic gradual guarantee (DGG) and noninterference (NI).

Several solutions [26]–[30] have been proposed to address this tension, but they often necessitate compromising other properties, such as type-guided classification (TGC) [28], [29] (the ability to classify values through static type annotations or casting) or no-sensitive-upgrade (NSU) checking [30] (a runtime mechanism that prevents assignments to low-security references when executing under high-security control flow to block implicit information leaks).

Recently, Chen and Siek [31] proposed a novel gradual security-typed language $\lambda^{\star}_{\mathtt{IFC}}$, which satisfies both NI and DGG without compromising TGC and NSU. To reconcile the tension between NI and DGG, $\lambda^{\star}_{\mathtt{IFC}}$ excludes the unknown label $\star$ from runtime labels: values carry a statically known initial label. Specifically, $\lambda^{\star}_{\mathtt{IFC}}$ introduces a coercion calculus for security labels as its runtime IFC monitor. However, the resulting coercion sequences lack conciseness, posing scalability challenges, particularly when dealing with more complex security label sets. Additionally, the absence of a principled mechanism for merging coercion sequences forces $\lambda^{\star}_{\mathtt{IFC}}$ to exclude binary operators, such as arithmetic, comparison, or logical operations, that are commonly found in other gradual security-typed languages [26], [28], [30]. This restriction

*Corresponding author: Zhiwu Xu (xuzhiwu@szu.edu.cn). This work was supported in part by the National Natural Science Foundation of China (Nos. 62372304, 61972260, 62472339).

significantly limits its practical applicability.

In this paper, we introduce λ^p, a partial cast calculus for gradual information security, based on the concept of *partial label*. A partial label is either a fully known security label or a partial known/unknown label that is at least/most equivalent to an associated security label. Partial labels concisely represent security coercions and runtime labels, and extend naturally to more complex security lattices. Moreover, partial labels facilitate straightforward definitions of label-related operators, such as the join operator for combining two runtime labels. This capability enables λ^p to effectively support binary operators. Finally, partial labels simplify the semantics, type system, and precision relations within λ^p, thereby facilitating mechanized verification. In our proof conducted in Agda, we maintain an abstract representation of the security label lattice, ensuring that all lemmas and theorems presented herein apply to any lattice with the required properties.

We adopt the same strategy as $\lambda^{\star}_{\mathtt{IFC}}$ to reconcile the tension between NI and DGG: all values, including references, carry a statically known initial label. Although this design conflicts with the intuitive goal of gradual typing: weaker-typed or untyped code should interoperate seamlessly with strongly-typed code without modification, such security annotations are semantically necessary to maintain soundness guarantees. A pragmatic solution is to assume a default non-secret label, so that only high-security information demands explicit labeling, but this solution again sacrifices the DGG: untyped code that manipulates secrets is made to fail when it previously didn't. Consequently, like $\lambda^{\star}_{\mathtt{IFC}}$, our calculus excludes references with unknown labels. To accommodate such unknown references while preserving NI, we would need to maintain the initial partial labels of references at allocation time as a guard, which would necessitate compromising DGG.

To summarize, this paper makes the following contributions:

- We introduce the concept of partial labels (Section II), which serves as a concise representation for security coercions and runtime labels.
- We develop a cast calculus for partial labels (Section III-A) and coercion labels (Section III-B), exploring several fundamental properties of security labels.
- We propose a partial cast calculus λ^p for gradual information security (Section IV), which satisfies both NI and DGG. This calculus can be readily extended to accommodate more complex security label sets while effectively facilitating binary operators.
- We present a gradual surface language λ^g as well as the compilation from surface terms to cast-calculus terms (Section V). We prove that this compilation preserves typing, precision relation and observed equivalence.
- We extend our calculus to accommodate unknown references, albeit at the cost of sacrificing DGG (Section V-C), and provide a semantic interpretation of partial labels (Section VI).
- We mechanize all lemmas and theorems in this paper in Agda, available at https://github.com/SZU-SE/Partial-Cast-Calculus-for-Gradual-Information-Security.

II. Background and Motivation

This section establishes the necessary background on security types and gradual security types, then surveys existing approaches to gradual IFC, highlighting their limitations. We conclude with the key idea underlying our partial cast calculus.

A. Security Types

In an Information Flow Control (IFC) system [32], [33], data is classified according to its inherent sensitivity level. For instance, *name* is classified as a low-security value, whereas *salary* represents a high-security value. Similarly, types are also associated with security labels; for example, $\mathtt{String}_{\mathtt{low}}$ is the type for strings with low security, whereas $\mathtt{Int}_{\mathtt{high}}$ is the type for integers with high security.

```
let name: String_low = "alice"_low
let salary: Int_high = 18000_high
```

Formally, these security labels constitute a lattice with an underlying partial order $\preccurlyeq$ to indicate the permissible flows of information within a program. For instance, $\{\mathtt{low}, \mathtt{high}\}$ forms a 2-element lattice where $\mathtt{low} \preccurlyeq \mathtt{high}$.

A property that captures information flow is termed *non-interference* (NI) [32], which asserts that variations in high-security input do not cause variations in low-security output. The following example illustrates two violations. First, passing the high-security variable *salary* to a function *time3* accepting a low-security input creates an explicit flow and is thus prohibited. Second, assigning the low-security variable *pub* within a branch conditioned on the high-security variable *salary* creates an implicit flow via control dependence, which is also prohibited.

```
let time3: Int_low →_low Int_low = λ x. x × 3_low in
   time3(salary) // explicit flow leak
let pub: Bool_low = false_low in
   // implicit flow leak
   if salary > 10000 then pub = true_low
```

Furthermore, *no-sensitive-upgrade* (NSU) [5] prevents implicit flows by tracking the program counter label *pc*, which accumulates the security label of all active branch conditions. A write to a variable with label ℓ is permitted only if $pc \preccurlyeq \ell$. In the example above, the branch condition $salary > 10000$ raises pc to $\mathtt{high}$. The subsequent assignment to *pub* (labeled $\mathtt{low}$) violates $\mathtt{high} \not\preccurlyeq \mathtt{low}$, causing NSU to reject the operation and prevent the information leak.

To characterize implicit flow in functions, a security label ℓ is utilized to represent the write effect of a function, indicating that every assignment within the function targets a variable with a security level meeting or exceeding ℓ.

B. Gradual Security Types

Gradual security typing enables programmers to begin with "unknown" labels and incrementally strengthen them over time, rather than requiring complete static annotations upfront. Specifically, a single placeholder label $\star$ is introduced to annotate both types and values, which signifies an unknown label at compile-time and will be verified at runtime [26], [28],

[30], [31]. For instance, $\texttt{Bool}_\star$ is the type of a boolean whose security label is unknown at compile time. The set of primitive labels along with the unknown label $\star$ are collectively referred to as gradual labels. The placeholder $\star$ serves as the most permissive static approximation: the type system never rejects a program for lack of annotations, yet it still enforces the intended security policy—statically where possible, dynamically where necessary. For example, the following code snippet defines a `square` function that accepts an argument bearing *any* security label at compile time:

```
let square: Int_⋆ →_⋆ Int_⋆ = λ x. x × x
let square_H: Int_high →_high Int_high = λ x. x × x
let square_L: Int_low →_low Int_low = λ x. x × x
```

Gradual security typing further affords significant flexibility. In contrast, under a purely static typing discipline, implementing the `square` function would force a choice between two unsatisfactory alternatives: (i) over-approximating the argument's label as `high`, potentially rejecting otherwise valid programs; or (ii) under-approximating it as `low`, thereby compromising security. One could instead define separate monomorphic versions for each label, but this sacrifices code reuse and introduces combinatorial complexity. Label polymorphism abstracts over security labels, offering a principled alternative. But it requires explicit type annotations or sophisticated type inference, and forces call sites to explicitly instantiate polymorphic functions at concrete labels. Consequently, integrating polymorphic functions with unannotated code often introduces friction at boundaries, whereas gradual typing supports such interactions natively via implicit runtime checks.

A precision order $\sqsubseteq$ is defined for gradual security labels, where $g \sqsubseteq \star$ for any gradual label g and $\ell \sqsubseteq \ell$ for any specific security label ℓ. This precision order extends to types in a natural way, so for example, $\texttt{Bool}_{\texttt{low}} \rightarrow_{\texttt{low}} \texttt{Bool}_{\texttt{low}} \sqsubseteq \texttt{Bool}_\star \rightarrow_\star \texttt{Bool}_\star$. *Dynamic gradual guarantee* (DGG) [17] is introduced as a criterion for gradually typed languages, mandating that the dynamic behavior of a gradual type system should be consistent with the behavior of a static type system in a gradual manner. Specifically, DGG asserts that less precise security type annotations in a program should not cause more errors and is expected to yield a less precise result.

Type-guided classification (TGC) [26], [31] in gradual information flow control requires upgrading the runtime label associated with a value during a runtime type cast. This mechanism ensures that the dynamic label of the value remains consistent with its newly assigned static type.

C. Cast Calculus for Security Labels

In the following, we will focus on cast calculus specifically designed for security labels, wherein basic types are omitted. This cast calculus forms a fundamental aspect of a programming language equipped with gradual IFC. We will discuss diverse design approaches adopted by various programming languages, highlighting how they navigate the tension between noninterference and gradual guarantee.

Assuming a security label lattice denoted as LABEL, let ℓ represent a specific label from LABEL and g represent a gradual label from LABEL $\cup \{\star\}$. Let p denote a runtime label associated with values during execution. The cast with respect to security labels is represented as $p\ \langle g \rangle \Rightarrow p'$, which indicates that a runtime label p is cast by the gradual label g, resulting in another runtime label p'[1].

ML-GS [22] employs specific labels as the initial labels for values, and subsequently upgrades the runtime labels if the target label is superior than the source one (*e.g.*, `low` $\langle$`high`$\rangle \Rightarrow$ `high`), while disregarding any gradual information (*e.g.*, `low` $\langle\star\rangle \Rightarrow$ `low`). However, ML-GS does not satisfy the DGG concerning labels. Considering the above two casts, $\star$ is less precise than `high`, but the resulting label `low` does not exhibit less precise compared to `high`, both of which are fully known.

To ensure DGG, GLIO [28] and $\lambda^\star_{\text{SEC}}$ [29] check the cast operation but never changing their labels when coercing values. Consequently, this effectively addresses the issues associated with the aforementioned counterexamples. But this approach leads to a loss of TGC. For instance, when coercing a `low` value to `high`, the resulting label remains `low`, following the label cast rule `low` $\langle$`high`$\rangle \Rightarrow$ `low`. However, the resulting value is expected to be typed/labeled as `high`. Moreover, if a further cast with `low` is applied afterward, this cast process surprisingly succeeds, which should blame due to an inappropriate cast from `high` back to `low`.

GSL_{Ref} [26] is designed based on the principles of Abstracting Gradual Typing (AGT) [34], [35]. It incorporates the unknown label $\star$ as the initial labels of values and employs the label intervals to represent runtime labels. However, these label intervals primarily serve as an evidence for dynamic checks of consistent subtyping involved in cast operations, making the cast operations on labels somewhat complex[2]. Moreover, GSL_{Ref} identifies an inherent tension between NI and DGG, which fundamentally stems from the dynamic heap updates. Consider the following two programs:

```
1 //Program 1: less precise, more dynamic
2 let x: Ref Bool_high = ref true_high
3 let y: Ref Bool_⋆ = ref true_⋆
4 if !x then y := false_high
5 //Program 2: more precise, more static
6 let x: Ref Bool_high = ref true_high
7 let y: Ref Bool_high = ref true_high
8 if !x then y := false_high
```

The lower program executes without errors. However, an issue arises with the upper program: (i) Allowing the assignment in line 4 implies $\star$ should be classified as `high`, yet y remains usable as a `low` variable, violating NI. (ii) Disallowing it breaks DGG. To preserve NI, GSL_{Ref} employs an extra check on the dynamic heap updates, which compromises DGG.

Similarly, WHILE^{G} [30] is a simple imperative language that integrates gradual IFC based on AGT, it addresses the

[1]The *blame* result has been omitted for the sake of simplicity.

[2]If we focus on the label cast applied to values, the interior or refining operators in GSL_{Ref} or WHILE^{G} are fundamentally equivalent to our straightforward reduction rules presented in Figure 1.

tension between DGG and NI by statically collecting the write effects of the untaken branch while dynamically upgrading relevant references for both branches. For instance, in the example above, y would be upgraded to at least high to ensure NI. However, extending this approach to in a setting with higher-order functions, such as our calculus, would significantly complicate the tracking of such information. Moreover, this strategy comes at the cost of abandoning NSU checking. Consider the following program:

```
// heap: [y ↦ false_low]
if false_high then y := true_high
```

Under the NSU model, the assignment inside the branch is never executed, so the value of y remains false and its label stays low. In contrast, the hybrid model employed by WHILE^{G} upgrades the label of y to high due to the potential assignment within the high-security condition.

$\lambda^{\star}_{\texttt{IFC}}$ [31] resolves the tension by revisiting a design choice in GSL_{Ref}: allowing the unknown label $\star$ as the initial values labels. Traditionally, this unknown label $\star$ indicates a lack of static information rather than a lack of dynamic information. By eliminating $\star$ from the runtime labels and defaulting literals to low, $\lambda^{\star}_{\texttt{IFC}}$ is able to fulfill all relevant properties. In particular, $\lambda^{\star}_{\texttt{IFC}}$ introduces a coercion calculus for security labels that encompasses aspects such as security coercion, coercion sequences, reduction rules, and composition of coercion sequences. Runtime labels are expressed as label expressions consisting of a specific label followed by a sequence of security coercions. However, irreducible coercion sequences (*i.e.*, coercion sequences in normal form) lack conciseness, which poses scalability challenges, especially with complex security label sets. Furthermore, the calculus offers no principled way to merge two coercion sequences, forcing $\lambda^{\star}_{\texttt{IFC}}$ to exclude binary operators, such as arithmetic, comparison, or logical operations, that are commonly found in other gradual security-typed languages [26], [28], [30], thereby restricting its practical applicability. We illustrate these issues in the following section.

D. Limitations of $\lambda^{\star}_{\texttt{IFC}}$

Space and Scalability Issues. Values in $\lambda^{\star}_{\texttt{IFC}}$ are associated with a label expression that consists of an initial concrete label followed by a potentially empty irreducible coercion sequence. While semantically precise, this representation can be verbose. For instance, the values $20000_{\texttt{low}}\langle\overrightarrow{\texttt{low};\texttt{high}}\rangle$ and $12000_{\texttt{low}}\langle\overrightarrow{\texttt{low};\texttt{mid};\star}\rangle$ can indeed be further simplified to $20000_{\texttt{high}}$ and $12000_{\overrightarrow{\texttt{mid};\star}}$, respectively[3].

Moreover, determining whether a term qualifies as a value necessitates identifying the normal form of its coercion sequence. However, since normal forms are defined recursively, this process involves non-trivial checks. For instance, $\overrightarrow{\texttt{low};\texttt{high};\star}$ is recognized as a normal form, whereas $\overrightarrow{\texttt{low};\star;\texttt{high}}$ is not. Furthermore, operations on coercion sequences are defined enumeratively through an explicit listing of all possible normal forms. Consider the *stamp* operator in $\lambda^{\star}_{\texttt{IFC}}$, which promotes the security of a coercion sequence to at least the specified security label and serves to capture the implicit flow. For the two-point lattice $\{\texttt{low},\texttt{high}\}$, this operator already requires six distinct normal forms.

The scalability limitations become apparent in richer security lattices. Take a financial system with two teams and one department whose respective labels are $\texttt{mid}_1$, $\texttt{mid}_2$, and mid with $\texttt{mid} \preccurlyeq \texttt{mid}_1$ and $\texttt{mid} \preccurlyeq \texttt{mid}_2$ (indicating that team data are more private than departmental data). Beyond low and high, this lattice contains at least five distinct labels, yielding exponentially more ascending sequences (*i.e.*, normal forms) than the six cases required for the *stamp* operator. Concise normal forms are therefore essential.

Absence of Binary Operators. Binary operators, such as arithmetic, comparison, or logical operations, are commonly found in other gradual security-typed languages [26], [28], [30]. However, $\lambda^{\star}_{\texttt{IFC}}$ does not support binary operators, which restricts its practical applicability.

This limitation primarily arises from the inherent challenges associated with joining two label expressions. Specifically, a binary expression must compute the join of $\ell_1;\overrightarrow{g_1} \sqcup \ell_2;\overrightarrow{g_2}$, where ℓ_i and $\overrightarrow{g_i}$ denote the initial concrete labels and the irreducible coercion sequences for the respective values involved. A straightforward approach to joining two coercion sequences is to merge them into an ascending sequence, which is relatively simple for a linear lattice (*e.g.*, $\{\texttt{low},\texttt{high}\}$). For instance, the result of $\overrightarrow{\texttt{low}} \sqcup \overrightarrow{\texttt{high}}$ is $\overrightarrow{\texttt{low};\texttt{high}}$. However, this approach fails for non-linear lattices when attempting to combine unrelated labels. For instance, consider the join of $\overrightarrow{\texttt{low};\texttt{high}} \sqcup \overrightarrow{\texttt{low};\texttt{high}'}$, where high and $\texttt{high}'$ represent two distinct secret labels associated with different categories of private data. These labels are unrelated, and thus cannot be effectively merged into a single sequence. An alternative approach involves calculating and combining the security labels of coercion sequences (denoted as $|\_|$). So the result of the above join is $|\overrightarrow{\texttt{low};\texttt{high}'}| \sqcup |\overrightarrow{\texttt{low};\texttt{high}}| = \top$. However, this approach causes coercions to disappear from values involving binary operators, with a critical consequence: the potential loss of unknown labels, which may violate the DGG. To illustrate, consider adding two integers with coercion sequences $\overrightarrow{\texttt{low};\star}$ and $\overrightarrow{\texttt{low};\texttt{mid}}$. The join of these coercion sequences is mid, which no longer incorporates any coercion sequence and omits $\star$ entirely. Moreover, the join operation fails to preserve the precision order. If the first value's coercion sequence becomes $\overrightarrow{\texttt{low};\texttt{high}}$ (more precision than $\overrightarrow{\texttt{low};\star}$), the join yields high, which is not more precise than mid. This violates the expectation that more precise inputs should yield more precise (or equivalent) results. Thus, no principled method exists for merging coercion sequences in $\lambda^{\star}_{\texttt{IFC}}$ while preserving both precision and DGG.

E. Partial Cast Calculus

Our Design Choices. We propose a cast calculus for gradual IFC that supports rich security label lattices and primitive binary operators, making it suitable for practical

[3] This latter sequence suggests that the value is at least mid, implying that mid cannot be omitted.

applications. To enhance expressivity, our calculus includes first-class functions, which are essential for abstraction and compositional programming, as well as heap objects, which enable mutable state and shared data structures. Regarding security and gradual guarantees, our calculus is proved to enforce NI, DGG, TGC, and NSU checking. To reconcile the tension between NI and DGG, we adopt the same strategy as $\lambda^{\star}_{\texttt{IFC}}$: every value, including heap-allocated data, carries a statically known initial label. To sum up, ours is the first calculus to satisfy all these properties.

Our calculus allows programmers to explicitly annotate constants, mutable references, and λ-abstractions with specific security labels. Unannotated values default to `low`. This default is both pragmatic and justified: in realistic programs, most literals (if not all) carry low-security sensitivity [31]. Consequently, only high-security information requires explicitly labeling by programmers, particularly when integrating legacy code.

Similar to WHILE$^{\mathrm{G}}$ [30], we model I/O via two primitive functions: `input(`ℓ`)`, which reads from a channel labeled ℓ, and `print(`ℓ`,` v`)`, which outputs value v to a channel labeled ℓ. For convenience, we provide default behaviors: when invoked without an explicit label, `input` defaults to `high` (modeling sensitive input) and `print` defaults to `low` (modeling a public output channel). While these defaults capture a common sanitization scenario, they do not accommodate inputs of unknown sensitivity.

A limitation of our calculus is its lack of support for unknown references. To accommodate them while preserving NI, we maintain the initial partial labels of unknown references at allocation time as a guard (see Section V-C). This guard, however, can reject programs that would succeed with less precise types, thereby weakening DGG. The trade-off is pragmatic: heap-allocated objects in legacy code lack explicit labels, but such labels can be inferred from programmer-provided annotations or their initial values at creation time.

In addition, declassification [36] is the controlled, explicit relaxation of non-interference policies to permit specific high-to-low flows while maintaining security guarantees for unauthorized flows, which will be explored in future work.

Partial Labels. Our cast calculus for security labels are based on the concept of *partial label*, which provides a concise representation for security coercions and runtime labels. Firstly, inspired by cast calculus on types [37], [38], we observe that it is unnecessary to maintain the (entire) sequence of coercions. Instead, retaining only the source label, target label, greatest lower bound (GLB), and least upper bound (LUB) is sufficient for valid (*i.e.*, ascending) sequences: $\langle source, GLB, LUB, target\rangle$. The GLB represents the maximum label applicable by the sequence, whereas the LUB indicates the minimum label that can be returned from it. For instance, the coercion sequence $\overrightarrow{\star; \texttt{low}; \star; \texttt{mid}; \star; \texttt{high}; \star}$ can be represented as $\langle \star, \texttt{low}, \texttt{high}, \star\rangle$. Secondly, when the target label is known (*i.e.*, not $\star$), an identity exists between the target label and LUB; thus preserving either one is adequate. In contrast, when the target label is unknown (*i.e.*, $\star$), assuming that LUB is ℓ, we represent these two labels as $\ell\,?$, indicating that the target label is partially unknown and at least equivalent to ℓ. Consequently, we define *partial labels* as follows:

$$p ::= \ell \mid \ell\ ?$$

Similarly, with respect to the source label and GLB, they can be represented as either ℓ or $?\ \ell$, where $?\ \ell$ indicates that the source label is partially unknown and at most equivalent to ℓ. These representations can be viewed as the reverse facets of p. So we define the *reverse partial labels* as follows:

$$\hat{p} ::= \ell \mid ?\ \ell$$

Based on these definitions above, coercion sequences are concisely represented as a pair of a reverse partial label and a partial label: $\langle \hat{p} \triangleright p\rangle$. We refer to this representation as a *coercion label*. For instance, the above example $\langle \star, \texttt{low}, \texttt{high}, \star\rangle$ can be represented as $\langle ?\ \texttt{low} \triangleright \texttt{high}\ ?\rangle$. Moreover, to reconcile the tension between NI and DGG, our calculus, like $\lambda^{\star}_{\texttt{IFC}}$, requires that the initial labels of values are static known (*e.g.*, ℓ). This eliminates the need for the GLB, allowing the runtime label $\ell\langle \hat{p} \triangleright p\rangle$ to be simplified to a partial label p.

Label Operators. In addition to their space efficiency, these concise representations facilitate clear and straightforward definitions of operations associated with runtime labels. As illustrated in Section III-A, the label cast is defined as follows: casting a runtime label with a known label is successful only when the cast label is greater than or equal to the current runtime label. Conversely, when casting a runtime label with an unknown label $\star$, the runtime label may be updated to a partial label if necessary. For instance,

$$\texttt{low}\ ?\ \langle \texttt{high}\rangle \Rightarrow \texttt{high} \qquad\qquad \texttt{low}\ \langle\star\rangle \Rightarrow \texttt{low}\ ?$$

The join of two partial labels refers to their least upper bound with respect to the partial order on security labels. To compute it, we join the corresponding underlying security labels while preserving the unknown part as needed. For example,

$$\texttt{low}\ ? \sqcup \texttt{high} = \texttt{high}\ ? \qquad\qquad \texttt{high} \sqcup \texttt{low}\ ? = \texttt{high}\ ?$$

These examples illustrate that joining a label that is at least `low` with `high` yields a label that is at least `high`. This join operator preserves precision monotonicity: less precise inputs yield less precise or equivalent results. The operation that promotes a runtime label to at least a specific security level to capture implicit flows (*i.e.*, the *stamp* operator in $\lambda^{\star}_{\texttt{IFC}}$) can be directly expressed via the join operator. Moreover, these operators extend naturally to more complex security label sets.

Values. In contrast to $\lambda^{\star}_{\texttt{IFC}}$, which relies on security labels and coercion sequences, we associate each raw value with a partial label p in the intermediate language, which serves as a concise representation of runtime labels. For instance, the value $12000_{\texttt{low}}\langle\overrightarrow{\texttt{low}; \texttt{mid}; \star}\rangle$ is represented as $12000_{\texttt{mid}\ ?}$. Furthermore, both functions and heap objects are associated with certain coercion labels, which function as the coercion sequences necessary to facilitate alignment for function calls and operations involving reading or writing heap objects. For instance, consider the following function (referred to as f), which associates a partial label along with three coercion labels:

$$(\lambda^{\langle \texttt{low} \triangleright \texttt{high} \rangle} x.x \times 2_{\texttt{mid}\ ?})^{\langle ?\ \texttt{mid} \triangleright \texttt{mid} \rangle,\ \langle ?\ \texttt{high} \triangleright \texttt{high} \rangle}_{\texttt{low}}$$

These labels represent the runtime label (*i.e.*, $\texttt{low}$) of the function itself, as well as the coercions applied to its argument (*i.e.*, $\langle ?\ \texttt{mid} \triangleright \texttt{mid} \rangle$), effect (*i.e.*, $\langle \texttt{low} \triangleright \texttt{high} \rangle$), and return value (*i.e.*, $\langle ?\ \texttt{high} \triangleright \texttt{high} \rangle$).

Binary Expressions. As previously mentioned, the join of partial labels is permissible within our framework, facilitating effective support for binary operators. For instance, adding two integers $20000_{\texttt{mid}_1} + 12000_{\texttt{mid}_2\ ?}$ yields $32000_{\texttt{high}\ ?}$, where the result label is the join of the operand labels. Moreover, if the second operand's label is concretized to $\overrightarrow{\texttt{mid}_2; \texttt{high}}$ (*i.e.*, $\texttt{high}$), the result becomes $32000_{\texttt{high}}$, which is more precise than $\texttt{high}\ ?$.

Casts. Casts mediate between the static (gradual) types and dynamic (runtime) types, and are inserted as needed during both compilation and evaluation phases. Applying the function f on an integer $3_{\texttt{low}}$ proceeds as follows, where pc denotes the label of program counter (PC) and is assumed to be $\texttt{low}$:

$$\begin{array}{rl}
 & f(3_{\texttt{low}}) \\
\xrightarrow{pc' \sqcup \texttt{low}} & (3_{\texttt{mid}} \times 2_{\texttt{mid}\ ?})\langle ?\ \texttt{high} \triangleright \texttt{high} \rangle \\
\xrightarrow{pc' \sqcup \texttt{low}} & (6_{\texttt{mid}\ ?})\langle ?\ \texttt{high} \triangleright \texttt{high} \rangle \\
\xrightarrow{pc' \sqcup \texttt{low}} & 6_{\texttt{high}} \\
\xrightarrow{pc} & 6_{\texttt{high}} \sqcup \texttt{low} \\
\xrightarrow{pc} & 6_{\texttt{high}}
\end{array}$$

The application succeeds as both the argument cast and the effect cast associated with function f succeed when applied to the argument (*i.e.*, $3_{\texttt{low}}\langle ?\ \texttt{mid} \triangleright \texttt{mid} \rangle \to 3_{\texttt{mid}}$) and PC label (*i.e.*, $pc\langle \texttt{low} \triangleright \texttt{high} \rangle \to pc'$), respectively. Before returning, this result is cast according to the return cast. Furthermore, the reduction of the function body must be safeguarded by the label $\texttt{low}$ associated with the function f, which is achieved through the promotion of the PC label and the promotion of the result.

III. Cast Calculus for Security Labels

As mentioned in Section II-E, values in IFC carry runtime labels (sensitivity levels) and security coercion sequences (casts facilitating alignments). This section first introduces a cast calculus for partial labels (Section III-A), providing a concise representation of runtime labels and clear definitions of related operators. It then presents a cast calculus for coercion labels (Section III-B), extending partial labels to compactly represent security coercion sequences.

A. Cast Calculus for Partial Labels

Let $(\textsc{Label}, \preccurlyeq, \curlyvee, \curlywedge)$ be a security lattice, where $\textsc{Label}$ represents a set of security labels; the relation $\preccurlyeq$ denotes the partial order among these labels, which captures the permissible flows of information within a program; $\curlyvee$ and $\curlywedge$ refer to the join and meet operators of the labels, respectively. A gradual security label g is either a static label ℓ or the unknown label $\star$. The partial order $\preccurlyeq$ can be extended to a consistent partial order $\precsim$ on gradual labels [17], [26], where $\ell \precsim \star$ and $\star \precsim \ell$ for all ℓ. A *partial label* p is either a fully known label ℓ, or a partially known/unknown label $\ell\ ?$, indicating that although partially known/unknown, the label is at least equivalent to ℓ.

$$\begin{array}{rrcl}
\text{Security label} & \ell & \in & \textsc{Label} \\
\text{Gradual label} & g & ::= & \ell \mid \star \\
\text{Partial label} & p & ::= & \ell \mid \ell\ ?
\end{array}$$

The precision relations of gradual labels and partial labels are illustrated in Figure 1, where $p \sqsubseteq p'$ means that the label p is more precise than another label p'[4]. The precision relation of gradual labels follows standard conventions [17], [26]. The first two rules governing the precision relation of partial labels are relatively straightforward. However, further clarification is necessary for the latter two[5]. A partial label can be interpreted as a set of labels, with the precision relation corresponding to the subset inclusion: a superset denotes a less precise label.

The typing relation for partial labels is of the form $\vdash p : g$, which indicates that a partial label p is typed by a gradual label g. The typing rules are straightforward and also presented in Figure 1.

A label cast involves casting a partial label p by a gradual label g, represented as $p\ \langle g \rangle\ \Rightarrow\ p'$, where p' denotes the resulting label. The label cast serves to model explicit flows. Consequently, if the cast label g is known and not higher than the current (known) label p, then such a cast will lead to blame. Otherwise, the partial label p is upgraded with respect to the cast label g. The reduction rules applicable to both scenarios are presented in Figure 1 as well.

The join of two partial labels refers to their least upper bound with respect to the consistent partial order. It computes the join of the corresponding underlying security labels while preserving the unknown part as required:

$$\begin{array}{rclcrcl}
\ell \sqcup \ell' & = & \ell \curlyvee \ell' & \quad & \ell\ ? \sqcup \ell' & = & (\ell \curlyvee \ell')\ ? \\
\ell \sqcup \ell'\ ? & = & (\ell \curlyvee \ell')\ ? & \quad & \ell\ ? \sqcup \ell\ ? & = & (\ell \curlyvee \ell')\ ?
\end{array}$$

For example, joining a label that is at least $\texttt{low}$ with $\texttt{high}$ yields a label that is at least $\texttt{high}$. With this partial label, we can directly employ a join operator to capture implicit flows, as in traditional static security type systems [26]. Consequently, the stamp and stamp! found in $\lambda^{\star}_{\texttt{IFC}}$ [31] become unnecessary. Moreover, the join operator of partial labels naturally accommodates binary operators within expressions.

1) Properties of Partial Labels: In this section, we explore several fundamental properties for security labels. This analysis proves crucial in developing languages with gradual Information Flow Control (IFC) and serves as a prerequisite for establishing noninterference (NI) and gradual guarantees (GG) within such languages.

[4] We follow the precision notation used in [17], [26].

[5] Rule (PPℓ'?) is superfluous, which can be drawn from Rules (PPℓ?) and (PP??). We retain it for the sake of simplicity in mechanized proofs.

$$\boxed{g \sqsubseteq g'} \qquad \boxed{\vdash p : g}$$

$$\textsc{GP}\star\ \frac{}{g \sqsubseteq \star} \qquad \textsc{GP}\ell\ \frac{}{\ell \sqsubseteq \ell} \qquad \textsc{Ty}\ell\ \frac{}{\vdash \ell : \ell} \qquad \textsc{Ty}\star\ \frac{}{\vdash \ell\ \textbf{?} : \star}$$

$$\boxed{p \sqsubseteq p'}$$

$$\textsc{PP}\ell\ell\ \frac{}{\ell \sqsubseteq \ell} \qquad \textsc{PP}\ell?\ \frac{}{\ell \sqsubseteq \ell\ \textbf{?}} \qquad \textsc{PP}\ell'?\ \frac{\ell' \preccurlyeq \ell}{\ell \sqsubseteq \ell'\ \textbf{?}} \qquad \textsc{PP}??\ \frac{\ell' \preccurlyeq \ell}{\ell\ \textbf{?} \sqsubseteq \ell'\ \textbf{?}}$$

$$\boxed{p\ \langle g \rangle \Rightarrow p'} \qquad \boxed{p\ \langle g \rangle \Rightarrow \texttt{blame}}$$

$$\textsc{Cast}\ell\star\ \frac{}{\ell\ \langle \star \rangle \Rightarrow \ell\ \textbf{?}} \qquad \textsc{Cast}\ell?\star\ \frac{}{\ell\ \textbf{?}\ \langle \star \rangle \Rightarrow \ell\ \textbf{?}} \qquad \textsc{Cast}\ell\ell'\ \frac{\ell \preccurlyeq \ell'}{\ell\ \langle \ell' \rangle \Rightarrow \ell'}$$

$$\textsc{Cast}\ell?\ell'\ \frac{\ell \preccurlyeq \ell'}{\ell\ \textbf{?}\ \langle \ell' \rangle \Rightarrow \ell'} \qquad \textsc{CastB}\ell\ell'\ \frac{\ell \not\preccurlyeq \ell'}{\ell\ \langle \ell' \rangle \Rightarrow \texttt{blame}} \qquad \textsc{CastB}\ell?\ell'\ \frac{\ell \not\preccurlyeq \ell'}{\ell\ \textbf{?}\ \langle \ell' \rangle \Rightarrow \texttt{blame}}$$

Fig. 1. Precision, typing and cast of partial labels.

Let $|p|$ denote the security label of a runtime label[6] p. For our partial label, it is defined as follows:

$$|\ell| = \ell \qquad |\ell\ \textbf{?}| = \ell$$

Label cast is employed to model explicit flows. Consequently, values must not be downgraded through cast; in other words, a successful cast should never diminish the security label of the runtime label. Failing to comply with this principle would violate NI. We refer to this property as *flow safety for security labels* (Theorem 3.1). NI is a fundamental property for IFC. Consequently, all related languages prioritize flow safety and adhere to its principles, including ours.

Theorem 3.1 (Flow safety for security labels): Let p, p' be two runtime labels, and g a gradual label. If $p\ \langle g' \rangle \Rightarrow p'$, then $|p| \preccurlyeq |p'|$.

Based on the work of Siek et al. [17], we establish *static* gradual guarantee (SGG) and *dynamic* gradual guarantee (DGG) for security labels. The SGG (Theorem 3.2) states that: decreasing the precision of a runtime label does not affect its well-typedness, and would be typed by a gradual label of less precision. While the DGG (Theorem 3.3) states that: less precise label-cast should not cause more runtime errors and is expected to yield a less precise result. As discussed in Section II, ML-GS does not fulfill the DGG for security labels, thereby failing to comply with the DGG at the language level. In contrast, our calculus of partial labels adheres to both the SGG and DGG.

Theorem 3.2 (SGG for security labels): Let p_1, p_2 be two runtime labels, and g_1 a gradual label. If $p_1 \sqsubseteq p_2$, and $\vdash p_1 : g_1$, then there exists a gradual label g_2 s.t. $\vdash p_2 : g_2$ and $g_1 \sqsubseteq g_2$.

Theorem 3.3 (DGG for security labels): Let p_1, p_2 be two runtime labels, and g_1, g_2 two gradual labels. Suppose that $p_1 \sqsubseteq p_2$ and $g_1 \sqsubseteq g_2$.

(1) If $p_1\ \langle g_1 \rangle \Rightarrow p_1'$, then there exists a runtime label p_2' s.t. $p_2\ \langle g_2 \rangle \Rightarrow p_2'$ and $p_1' \sqsubseteq p_2'$
(2) If $p_2\ \langle g_2 \rangle \Rightarrow \texttt{blame}$, then $p_1\ \langle g_1 \rangle \Rightarrow \texttt{blame}$.
(3) If $p_2\ \langle g_2 \rangle \Rightarrow p_2'$, then either there exists a runtime label p_1' s.t. $p_1\ \langle g_1 \rangle \Rightarrow p_1'$ and $p_1' \sqsubseteq p_2'$, or $p_1\ \langle g_1 \rangle \Rightarrow \texttt{blame}$.

Theorem 3.4 presents a type-based classification (TGC) framework for security labels, which states that: if a typable runtime label is successfully applied through cast with a gradual label, then the type of the runtime label should be a consistent subtype of the cast label; and the resulting runtime label is expected to be typed as the cast label. As discussed in Section II, GLIO and $\lambda^{\star}_{\texttt{SEC}}$ do not fulfill TGC for security labels, thereby failing to comply with this property at the language level. In contrast, our calculus of partial labels adheres to TGC.

Theorem 3.4 (TGC for security labels): Let p, p' be two runtime labels, and g, g' two gradual labels. If $\vdash p : g$ and $p\ \langle g' \rangle \Rightarrow p'$, then $\vdash p' : g'$ and $g \precsim g'$.

B. Cast Calculus for Coercion Labels

Casts can be executed one-by-one as needed. So we assert that the calculus of partial labels is sufficient for languages that execute cast either sequentially or using a lazy evaluation strategy [39]. However, for higher-order and reference casts, they are typically wrapped or attached as a coercion sequence, which is neither concise nor efficient. In this section, we will introduce our coercion labels, which provide a streamlined representation of coercion sequences.

First, we introduce the reverse partial labels, which denote that labels may be either fully known or partially known/unknown and *at most* equivalent to a label ℓ:

$$\hat{p} ::= \ell \mid \textbf{?}\ \ell$$

The precision relation, typing and security label of the reverse partial labels closely resemble those of partial labels.

A coercion label is either the unknown coercion $\star$, a valid coercion, or an invalid coercion, which is defined as follows:

$$c ::= \langle \star \rangle \mid \langle \hat{p}^{\hat{P}} \triangleright p \rangle \mid \langle \hat{p}^{\hat{P}} \triangledown g^{P} \rangle$$

where $P, \hat{P} \in \textsc{BlameLabel}$ are blame positions that may render the coercion invalid. Valid coercions are represented as pairs of a reverse partial label and a partial label, denoted by $\langle \hat{p}^{\hat{P}} \triangleright p \rangle$, where $\hat{p}$ and p denote the source label and the target label of the coercions, respectively; and $\hat{P}$ represents the blame position where the application could fail. Unlike existing work, these two labels may be partially known/unknown and are at most or at least equivalent to a security label ℓ. For convenience, we assume that a valid coercion satisfies the condition that its target label is greater than or equal to its source label (*i.e.*, $|\hat{p}| \preccurlyeq |p|$). This assumption eliminates the

[6]Our partial label serves as a representation of the runtime label. We utilize this notation to ensure that any programming language can instantiate the runtime label as its representation.

necessity for specifying a position for the target label. The unknown coercion $\langle\star\rangle$ could in principle be represented as the coercion label $\langle ?\ \texttt{high} \triangleright \texttt{low}\ ?\rangle$, rendering it redundant. However, this coercion label violates the well-formedness condition stated above, so we retain $\langle\star\rangle$ as a primitive.

Invalid coercions are represented as pairs consisting of a partial label and a gradual label, denoted as $\langle \hat{p}^{\hat{P}} \nabla g^{P}\rangle$. Intuitively, invalid coercions can no longer be applied. However, coercions always proceed from left to right. Consequently, blame can arise between the runtime label and the coercion label regardless of whether the coercion label is valid or invalid. For instance, in the cast of $\texttt{high}\langle \texttt{mid}_1{}^{\hat{P}} \nabla \texttt{low}^{P}\rangle$, it should blame at position $\hat{P}$ rather than position P. To effectively track blame positions, we retain both the source label and its associated position. Furthermore, to facilitate typing and precision relations among coercion labels, it is necessary to maintain the type information (*i.e.*, g) for invalid coercions. Invalid coercions are essential for tracking blame in coercion operations and also arise in programs when security checks fail at runtime. Moreover, invalid coercions enable the expression of special values[7], such as read-only heap objects. For instance, casting a heap object *obj* with label $\texttt{low}$ to $\star$ and then to $\texttt{high}$ succeeds in the read direction ($\texttt{low} \Rightarrow \star \Rightarrow \texttt{high}$) but fails in the write direction ($\texttt{high} \Rightarrow \star \Rightarrow \texttt{low}$), yielding a read-only heap object.

The typing relation and the precision relation of coercion labels are illustrated in Figure 2. Both relations are constructed based on their corresponding relations of partial labels, respectively. Note that, to maintain consistency with the SGG for security labels (Theorem 3.2), invalid coercions exhibit greater precision than any coercions involving types of less precision.

Figure 3 illustrates the reduction rules for the composition $c_1; c_2$ of coercion labels. The first four rules address scenarios in which one of the coercion labels is invalid. Intuitively, when an invalid coercion label is composed with any other coercion label, the result will invariably be an invalid coercion label. These four rules encompass more than just this principle; they also track and update both the source label and the blame position based on various cases. The subsequent three rules consider cases where one of the coercions is unknown (*i.e.*, $\langle\star\rangle$). Composing two unknown coercions is relatively straightforward. In the remaining two cases, where one of them would be a valid coercion, it suffices to update either the corresponding source or target label as partial labels if necessary. Finally, the last two rules aim to compose two valid coercions. Successful composition can occur only if the target label of the first coercion is less than or equal to the source label of the second coercion; otherwise, such a composition results in an invalid coercion.

The reduction rules for applying a coercion label to a runtime label (*i.e.*, coercion cast) are illustrated in Figure 4. Similar to cast with a gradual label, cast with an unknown coercion label $\star$ will update the runtime label as a partial

[7]Alternatively, one could reject these values immediately upon detecting the inconsistent coercion, thereby eliminating the need for invalid coercions.

$\boxed{\vdash c : g \Rightarrow g'}$

$$\text{TC}\star\ \frac{}{\vdash \langle\star\rangle : \star \Rightarrow \star} \qquad \text{TCPAIR}\ \frac{\vdash \hat{p} : g \qquad \vdash p : g'}{\vdash \langle \hat{p}^{\hat{P}} \triangleright p\rangle : g \Rightarrow g'}$$

$$\text{TCBLAME}\ \frac{\vdash \hat{p} : g}{\vdash \langle \hat{p}^{\hat{P}} \nabla g'^{P}\rangle : g \Rightarrow g'}$$

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

$$\text{PRPAIR}\ \frac{\vdash \hat{p} : g_1 \qquad \vdash c' : g_1' \Rightarrow g_2' \qquad g_1 \sqsubseteq g_1' \qquad g_2 \sqsubseteq g_2'}{\vdash \langle \hat{p}^{\hat{P}} \nabla g_2{}^{P}\rangle \sqsubseteq c' : g_1 \Rightarrow g_2 \sqsubseteq g_1' \Rightarrow g_2'}$$

$$\text{PRSTAR}\ \frac{\vdash c : g_1 \Rightarrow g_2}{\vdash c \sqsubseteq \langle\star\rangle : g_1 \Rightarrow g_2 \sqsubseteq \star \Rightarrow \star}$$

$$\text{PRPAIR}\ \frac{\vdash \hat{p} \sqsubseteq \hat{p}' : g_1 \sqsubseteq g_1' \qquad \vdash p \sqsubseteq p' : g_2 \sqsubseteq g_2'}{\vdash \langle \hat{p}^{\hat{P}} \triangleright p\rangle \sqsubseteq \langle \hat{p}'^{P} \triangleright p'\rangle : g_1 \Rightarrow g_2 \sqsubseteq g_1' \Rightarrow g_2'}$$

Fig. 2. Typing and precision of coercion labels.

label if necessary (Rule (CastCStar)). As indicated in Rules (CastCPair) and (CastCPairB), a valid coercion can only be successfully applied if the source label is greater than or equal to the current runtime label; otherwise, it raises a blame at the position associated with the source label. As previously stated, invalid coercions can no longer be applied, but need to track the blame position, as demonstrated in Rules (CastCBlame) and (CastCBlameB).

1) Properties of Coercion Labels: Indeed, cast with coercion labels is fundamentally built upon the principle of cast with gradual labels. It is straightforward to define a function, denoted as $g2c$, that converts a gradual label into a coercion label:

$$g2c(\star) = \langle\star\rangle \qquad\qquad g2c(\ell) = \langle \ell^{P} \triangleright \ell\rangle$$

where P denotes the code position of the label ℓ. It can be easily demonstrated that the coercion labels preserve the semantics of cast (Lemma 3.5).

Lemma 3.5 (Preservation of cast for coercion labels): Let p, p' be two runtime labels, and g a gradual label. If $p\ \langle g\rangle \Rightarrow p'$, then $p\langle g2c(g)\rangle \Rightarrow p'$.

Cast operators can be generalized to sequences of coercion labels by performing these operations on the resultant composition derived from the sequence:

$$p\langle c_1; c_2\rangle = p\langle c_3\rangle \qquad \text{if } c_1; c_2 \Rightarrow c_3$$

We show that coercion composition preserves the semantics of cast as well (Lemma 3.6) and adheres to the associative property (Lemma 3.7).

MgBlameL
$$\frac{\vdash c : g \Rightarrow g'}{\langle \hat{p}^{\hat{P}} \nabla g^{P} \rangle ; c \Rightarrow \langle \hat{p}^{\hat{P}} \nabla g'^{P} \rangle}$$

MgStarBlame
$$\frac{}{\langle \star \rangle ; \langle \hat{p}^{\hat{P}} \nabla g^{P} \rangle \Rightarrow \langle (\texttt{?}\ |\hat{p}|)^{\hat{P}} \nabla g^{P} \rangle}$$

MgPairBlame
$$\frac{|p_1| \preccurlyeq |\hat{p}_2|}{\langle \hat{p}_1^{\hat{P}_1} \triangleright p_1 \rangle ; \langle \hat{p}_2^{\hat{P}_2} \nabla g^{P_2} \rangle \Rightarrow \langle \hat{p}_1^{\hat{P}_1} \nabla g^{P_2} \rangle}$$

MgPairBlameB
$$\frac{|p_1| \not\preccurlyeq |\hat{p}_2|}{\langle \hat{p}_1^{\hat{P}_1} \triangleright p_1 \rangle ; \langle \hat{p}_2^{\hat{P}_2} \nabla g^{P_2} \rangle \Rightarrow \langle \hat{p}_1^{\hat{P}_1} \nabla g^{\hat{P}_2} \rangle}$$

MgStarId
$$\frac{}{\langle \star \rangle ; \langle \star \rangle \Rightarrow \langle \star \rangle}$$

MgPairStar
$$\frac{}{\langle \hat{p}^{\hat{P}} \triangleright p \rangle ; \langle \star \rangle \Rightarrow \langle \hat{p}^{\hat{P}} \triangleright |p|\ \texttt{?} \rangle}$$

MgStarPair
$$\frac{}{\langle \star \rangle ; \langle \hat{p}^{\hat{P}} \triangleright p \rangle \Rightarrow \langle (\texttt{?}\ |\hat{p}|)^{\hat{P}} \triangleright p \rangle}$$

MgPair
$$\frac{|p_1| \preccurlyeq |\hat{p}_2|}{\langle \hat{p}_1^{\hat{P}_1} \triangleright p_1 \rangle ; \langle \hat{p}_2^{\hat{P}_2} \triangleright p_2 \rangle \Rightarrow \langle \hat{p}_1^{\hat{P}_1} \triangleright p_2 \rangle}$$

MgPairB
$$\frac{|p_1| \not\preccurlyeq |\hat{p}_2| \qquad \vdash p_2 : g_2}{\langle \hat{p}_1^{\hat{P}_1} \triangleright p_1 \rangle ; \langle \hat{p}_2^{\hat{P}_2} \triangleright p_2 \rangle \Rightarrow \langle \hat{p}_1^{\hat{P}_1} \nabla g_2^{\hat{P}_2} \rangle}$$

Fig. 3. Composition of coercion labels.

Lemma 3.6 (Cast Preservation for coercion composition): Let p, p' be two runtime labels, and c_1, c_2 two coercion labels. $p\ \langle c_1; c_2 \rangle \Rightarrow p'$ if and only if $(p\langle c_1 \rangle)\ \langle c_2 \rangle \Rightarrow p'$.

Lemma 3.7 (Associative property of coercion composition): Let c_1, c_2, c_3 be three coercion labels. $(c_1; c_2); c_3 = c_1; (c_2; c_3)$.

Next, we demonstrate that coercion composition adheres to a principle consistent with DGG: less precise coercion composition should not cause more blames and is expected to yield a less precise result.

Lemma 3.8 (DGG for coercion composition): Let c_1, c_1', c_2, c_2' be coercion labels. Suppose that $\vdash c_1 \sqsubseteq c_1' : g_1 \Rightarrow g_2 \sqsubseteq g_1' \Rightarrow g_2'$ and $\vdash c_2 \sqsubseteq c_2' : g_2 \Rightarrow g_3 \sqsubseteq g_2' \Rightarrow g_3'$. If $c_1; c_2 \Rightarrow c_3$, then there exists a coercion label c_3' s.t. $c_1'; c_2' \Rightarrow c_3'$ and $\vdash c_3 \sqsubseteq c_3' : g_1 \Rightarrow g_3 \sqsubseteq g_1' \Rightarrow g_3'$.

Furthermore, we investigate several fundamental properties presented in Section III-A1 for coercion labels. We demonstrate that the cast calculus of coercion labels also adheres to the flow safety, DGG, and TGC. Note that, SGG is likewise

CastCStar
$$\frac{}{p\ \langle \star \rangle \Rightarrow |p|\ \texttt{?}}$$

CastCPair
$$\frac{|p| \preccurlyeq |\hat{p}|}{p\ \langle \hat{p}^{P} \triangleright p' \rangle \Rightarrow p'}$$

CastCPairB
$$\frac{|p| \not\preccurlyeq |\hat{p}|}{p\ \langle \hat{p}^{P} \triangleright p' \rangle \Rightarrow \texttt{blame}\ P}$$

CastCBlame
$$\frac{|p| \preccurlyeq |\hat{p}|}{p\ \langle \hat{p}^{\hat{P}} \nabla g^{P} \rangle \Rightarrow \texttt{blame}\ P}$$

CastCBlameB
$$\frac{|p| \not\preccurlyeq |\hat{p}|}{p\ \langle \hat{p}^{\hat{P}} \nabla g^{P} \rangle \Rightarrow \texttt{blame}\ \hat{P}}$$

Fig. 4. Cast with coercion labels.

$$
\begin{array}{rcl}
R & ::= & \texttt{Bool} \mid \texttt{Unit} \mid \texttt{Ref}\ T \mid T \xrightarrow{g} S \\
T, S & ::= & R_g \\
C & ::= & \texttt{CUnit} \mid \texttt{CBool} \mid \texttt{CRef}\ \hat{\mathcal{C}}_{in}\ \hat{\mathcal{C}}_{out} \mid \hat{\mathcal{C}}_{arg} \xrightarrow{c} \hat{\mathcal{C}}_{ret} \\
\hat{\mathcal{C}} & ::= & C_c \\
b & ::= & \texttt{true} \mid \texttt{false} \\
r & ::= & b \mid \texttt{unit} \mid o^{\ell;\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid (\lambda^c x : T.t)^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \\
v & ::= & r_p \\
\oplus & ::= & \wedge \mid \vee \\
err & ::= & \texttt{nsu-error}\ P \mid \texttt{blame}\ P \\
t, s & ::= & x \mid v \mid t\ s \mid t \oplus s \mid \texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & & \mid \texttt{ref}^{P}_{R_\ell}\ t \mid !t \mid t :=^{P} s \mid t\ \langle \hat{\mathcal{C}} \rangle \mid \texttt{let}\ x = t\ \texttt{in}\ s
\end{array}
$$

Fig. 5. Syntax of partial cast calculus λ^p

satisfied for coercion labels as illustrated in the definition of precision relation.

IV. Partial Cast Calculus λ^p

Building on our partial labels and coercion labels, we have designed a language for gradual information-flow control that satisfies both noninterference (NI) and the gradual guarantee (GG). Following the structure of many other gradual languages, our framework consists of a gradual surface language λ^g (see Section V) and a partial cast calculus λ^p. This section presents our partial cast calculus λ^p.

A. Syntax and Typing

Figure 5 presents the syntax of partial cast calculus λ^p, which serves as the core calculus addressing structures related to casts. A type T consists of a raw type R and a gradual label g. The raw type R can be Bool, Unit, a reference type, or a function type. Note that, the function type $T \xrightarrow{g} S$ exhibits an inherent security effect denoted by g. Similarly, a coercion $\hat{\mathcal{C}}$ is comprised of a raw coercion C and a coercion label c, which casts the raw types and gradual labels, respectively.

A raw value is either a boolean value, a unit, a heap object, or a function. Heap objects are instantiated only at runtime and are absent from the surface language (see Section V). In contrast to previous works that rely on either security labels or gradual labels [26], [31], we associate each raw value with a partial label p, which serves as a concise form of runtime labels introduced in Section III-A. Specifically, a heap object is represented as $o^{\ell;\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}}$, where ℓ denotes the security label of the referenced memory cell, $\hat{\mathcal{C}}_{in}$ represents the coercion applied when assigning a value to it, and $\hat{\mathcal{C}}_{out}$ represents the coercion used when reading a value from it during dereferencing. Note that, to reconcile the tension between DGG and NI, following $\lambda^{\star}_{\mathtt{SEC}}$ [29] and $\lambda^{\star}_{\mathtt{IFC}}$ [31], heap labels must be statically known, precluding unknown references (discussed in Section V-C). Similarly, a function is represented as $(\lambda^c x : T.t)^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}}$, where $\hat{\mathcal{C}}_{arg}, \hat{\mathcal{C}}_{ret}$ respectively correspond to the coercions for the function argument and return value, and c represents the coercion for the function effect.

λ^p is a variant of lambda calculus that incorporates boolean expressions and references. Most terms are conventional. The reference term $\mathtt{ref}^{P}_{R_\ell} t$ denotes the creation of a new memory cell associated with ℓ in the heap. With our partial labels, λ^p can effectively accommodate boolean expressions $t \oplus s$, where $\oplus$ represents boolean operators like And ($\wedge$) and Or ($\vee$). There are two types of errors: non-sensitive updates (NSU) errors $\mathtt{nsu\text{-}error}\ P$ and cast errors $\mathtt{blame}\ P$, where the blame label $P \in$ BlameLabel specifies the positions of these errors.

The typing rules of λ^p are expressed in the form $\Gamma; \Sigma; g \vdash t : T$, where Γ and Σ respectively represent typing contexts for variables and heap objects, and g denotes the type (*i.e.*, gradual label) of the PC label. This formulation is consistent with that presented in GSL_{Ref}, yet it is simpler than the one found in $\lambda^{\star}_{\mathtt{IFC}}$. The difference is that $\lambda^{\star}_{\mathtt{IFC}}$ enforces a stricter separation between static and dynamic checking for NSU. Consequently, it requires multiple universal quantifiers ($\forall$) for PC labels, resulting in more complex typing rules for conditionals, λ-terms, and let-terms.

Figure 6 presents the typing rules for λ^p. Most rules are standard. Rule (TPtr) requires $\hat{\mathcal{C}}_{in}$ and $\hat{\mathcal{C}}_{out}$ to be symmetric coercions. Rule (TLam) examines three relevant coercions to facilitate the cast from $T \xrightarrow{gc} S$ to $T' \xrightarrow{gc'} S'$. Rule (TRef) confirms that the PC type is a consistent subtype relative to that of the reference.

B. Big-Step Operational Semantics

In this section, we present a big-step semantics for λ^p. In order to clarify the properties of λ^p, we divide the semantics into two distinct parts: normal reduction $\Downarrow$ and error reduction $\Downarrow_\epsilon$. The normal reduction relation takes the form $t \mid \mu \mid pc \Downarrow v \mid \mu'$, which indicates that term t and heap μ are evaluated under the PC label pc, resulting value v and heap μ'. A heap μ is a mapping that associates heap objects with their corresponding values. Figure 7 presents the reduction rules for the big-step semantics of λ^p.

Binary Operators. The evaluation of binary operators is straightforward (Rule (BBinary)), due to the join operator associated with partial labels.

Promotions. Promotion of the PC label blocks implicit flows and is realized by the join operator. Rules (BIfTrue) and (BIfFalse) evaluate the chosen branch under a PC label obtained by joining the current PC with the partial label of the branch guard. Rule (BApp) similarly promotes the PC with the function guard and the effect coercion label before the body is executed.

Coercions. The coercions in our calculus are represented as a single coercion label rather than a sequence of coercions, allowing us to cast any value in a single step. Numerous reduction rules employ casts: Rule (BCast) performs cast for values along with their respective coercions; Rule (BApp) applies the coercion $\hat{\mathcal{C}}_{arg}$ to cast the argument v, and generates both the PC cast and result cast for evaluating the function body; Rule (BDeref) retrieves the value from a pointer in the heap and subsequently casts it using the coercion $\hat{\mathcal{C}}_{out}$. In Rule (BAssign), the value is cast using the coercion $\hat{\mathcal{C}}_{out}$, followed by an NSU checking before storing it into the heap. Should any of these casting operations fail, a blame will be raised. Note that in our calculus, coercions are applied as needed, and two successive coercions will be composed when the associated term is reduced to either a function or a heap object.

NSU Checking. NSU checking is specifically designed to prevent unauthorized implicit flows that could compromise the integrity of the heap. When confidential information is improperly directed towards an incorrect heap, an $\mathtt{nsu\text{-}error}\ P$ will be generated instead of the typical blame (refer to Rules (ERefNSU) and (EAssignNSU)). Rules (BRef) and (BAssign) utilize this mechanism, which dynamically verify the safety of any implicit flow (as highlighted). Once NSU Checking is successfully passed, Rule (BRef) will allocate a new memory cell within the heap, where the function $\langle\langle T \precsim^{P} S \rangle\rangle$ facilitates a coercion from T to S.

C. Meta Theoretic Results of λ^p

We show that λ^p satisfies NI and GG. In our proof conducted in Agda, we maintain an abstract representation of the security label lattice to ensure that any lattice possessing the requisite properties can be accommodated by these theorems.

1) Address Equivalence and Noninterference: We prove NI through an observed equivalence relation $\approx^{\ell}$, which defines behavioral in-distinguishability from a specific observation label ℓ: two program configurations must be semantically equivalent across all observable components (including heap, terms, control flow, etc.) at or below ℓ. Differences in information above ℓ are unobservable.

Addresses allocated by an observed reference across multiple executions may diverge due to certain unobservable references (*i.e.*, those labeled with high-security). To address this issue, we introduce a predicate *address equivalence* to characterize the corresponding addresses allocated by an observed reference over two executions. Specifically, we define an address context A to record address equivalence relations within the heaps of two distinct executions:

$$A ::= \emptyset \mid o ::^{L} A \mid o ::^{R} A \mid (o_1, o_2) ::^{E} A$$

Under the address context A, two addresses o_1 and o_2 are equivalent if and only if the pair (o_1, o_2) is included in A, even

$$\frac{x : T \in \Gamma}{\Gamma; \Sigma; gc \vdash x : T}\ \text{TVAR} \qquad \frac{\vdash p : g}{\Gamma; \Sigma; gc \vdash b_p : \texttt{Bool}_g}\ \text{TBOOL} \qquad \frac{\vdash p : g}{\Gamma; \Sigma; gc \vdash \texttt{unit}_p : \texttt{Unit}_g}\ \text{TUNIT}$$

$$\frac{o : R_\ell \in \Sigma \qquad \vdash \hat{\mathcal{C}}_{in} : T \Rightarrow R_\ell \qquad \vdash \hat{\mathcal{C}}_{out} : R_\ell \Rightarrow T \qquad \vdash p : g}{\Gamma; \Sigma; gc \vdash o_p^{\ell, \hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} : (\texttt{Ref}\ T)_g}\ \text{TPTR}$$

$$\frac{\Gamma, x : T; \Sigma; gc \vdash t : S \qquad \vdash \hat{\mathcal{C}}_{arg} : T' \Rightarrow T \qquad \vdash c : gc' \Rightarrow gc \qquad \vdash \hat{\mathcal{C}}_{ret} : S \Rightarrow S' \qquad \vdash p : g}{\Gamma, x : T; \Sigma; gc'' \vdash (\lambda^c x : T.t)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} : (T' \xrightarrow{gc'} S')_g}\ \text{TLAM}$$

$$\frac{\Gamma; \Sigma; gc \vdash t : \texttt{Bool}_{g_1} \qquad \Gamma; \Sigma; gc \vdash s : \texttt{Bool}_{g_2}}{\Gamma; \Sigma; gc \vdash t \oplus s : \texttt{Bool}_{g_1\ \widetilde{\curlyvee}\ g_2}}\ \text{TBINARY} \qquad \frac{\Gamma; \Sigma; gc \vdash t : (S \xrightarrow{gc\ \widetilde{\curlyvee}\ g} T)_g \qquad \Gamma; \Sigma; gc \vdash s : S}{\Gamma; \Sigma; gc \vdash t\ s : T\ \widetilde{\curlyvee}\ g}\ \text{TAPP}$$

$$\frac{\Gamma; \Sigma; gc \vdash t : \texttt{Bool}_g \qquad \Gamma; \Sigma; gc\ \widetilde{\curlyvee}\ g \vdash s_1 : S \qquad \Gamma; \Sigma; gc\ \widetilde{\curlyvee}\ g \vdash s_2 : S}{\Gamma; \Sigma; gc \vdash \texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : S\ \widetilde{\curlyvee}\ g}\ \text{TIF}$$

$$\frac{\Gamma; \Sigma; gc \vdash t : (\texttt{Ref}\ R_{g'})_g \quad \Gamma; \Sigma; gc \vdash s : R_{g'} \quad gc \precsim g' \quad g \precsim g'}{\Gamma; \Sigma; gc \vdash t :=^P s : \texttt{Unit}_\bot}\ \text{TASSIGN} \qquad \frac{\Gamma; \Sigma; gc \vdash t : R_\ell \qquad gc \precsim \ell}{\Gamma; \Sigma; gc \vdash \texttt{ref}_{R_\ell}^P\ t : (\texttt{Ref}\ R_\ell)_\bot}\ \text{TREF}$$

$$\frac{\Gamma; \Sigma; gc \vdash t : (\texttt{Ref}\ T)_g}{\Gamma; \Sigma; gc \vdash !t : T\ \widetilde{\curlyvee}\ g}\ \text{TDEREF} \qquad \frac{\Gamma; \Sigma; gc \vdash t : T \qquad \vdash \hat{\mathcal{C}} : T \Rightarrow S}{\Gamma; \Sigma; gc \vdash t\ \langle \hat{\mathcal{C}} \rangle : S}\ \text{TCAST} \qquad \frac{\Gamma; \Sigma; gc \vdash t : T \qquad \Gamma, x : T; \Sigma; gc \vdash s : S}{\Gamma; \Sigma; gc \vdash \texttt{let}\ x = t\ \texttt{in}\ s : S}\ \text{TLET}$$

Fig. 6. Typing rules for partial cast calculus λ^p

though they may not numerically equal $o_1 \neq o_2$. In addition to equivalent addresses, the context A also accommodates non-equivalent addresses. If a reference o is unobservable to the observer, its address is recorded in the context either as $o ::^L A$ for the (left) heap of the first execution, or as $o ::^R A$ for the (right) heap of the second execution.

Theorem 4.1 establishes the NI property: given two equivalent programs running under equivalent heaps and PC labels, their resulting values and heaps preserve equivalence.

Theorem 4.1 (NI): Suppose t_1, t_2 are equivalent terms $A; \emptyset; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell_o} t_2 : T$, μ_1, μ_2 are equivalent heaps $A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell_o} \mu_2$, and pc_1, pc_2 are equivalent PC labels $pc_1 \approx^{\ell_o} pc_2 : gc$. If $t_i \mid \mu_i \mid pc_i \Downarrow v_i \mid \mu_i'$, then there exist A', Σ_1', Σ_2' s.t. the values and heaps preserve equivalence $A'; \emptyset; \Sigma_1'; \Sigma_2' \vdash v_1 \approx^{\ell_o} v_2 : T$ and $A'; \Sigma_1'; \Sigma_2' \vdash \mu_1' \approx^{\ell_o} \mu_2'$.

2) Gradual Guarantee: Following the work of Siek et al. [17], we establish the static gradual guarantee (SGG) and the dynamic gradual guarantee (DGG). Similar to $\lambda^\star_{\texttt{IFC}}$ [31], we define the precision relations for heaps, terms, and values as follows:

$$\begin{aligned} \Sigma; \Sigma' \vdash \mu \sqsubseteq \mu' & \qquad \text{(Heap)} \\ \Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : T \sqsubseteq T' & \qquad \text{(Term)} \\ \Gamma; \Gamma'; \Sigma; \Sigma' \vdash v \sqsubseteq v' : T \sqsubseteq T' & \qquad \text{(Value)} \end{aligned}$$

Each relation is built point-wise from the precision relations of partial labels and coercion labels defined in Section III. Note that SGG is satisfied as illustrated in the precision relation of terms.

Theorem 4.2 establishes the DGG property through three distinct cases: first, if a more static program executes successfully, then a more dynamic program must necessarily succeed; second, if a more dynamic program fails, then a more static program must also fail; and third, if a more dynamic program executes successfully, a more static program may either succeed or fail, due to inherent nondeterminism.

Theorem 4.2 (DGG): Suppose that two terms t, t' are related by precision: $\emptyset; \emptyset; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : T \sqsubseteq T'$, two partial labels pc, pc' are related by precision: $\vdash pc \sqsubseteq pc' : gc \sqsubseteq gc'$, two heap contexts Σ, Σ' are related by precision: $\Sigma \sqsubseteq \Sigma'$, two initial heaps μ, μ' are also related by precision: $\Sigma; \Sigma' \vdash \mu \sqsubseteq \mu'$.

(1) If $t \mid \mu \mid pc \Downarrow v \mid \mu_1$, there exist Σ_1, Σ_1', v' and μ_1' s.t. $\Sigma_1 \sqsubseteq \Sigma_1'$, $t' \mid \mu' \mid pc' \Downarrow v' \mid \mu_1'$, the resulting values are related by precision: $\emptyset; \emptyset; \Sigma_1; \Sigma_1' \vdash v \sqsubseteq v' : T \sqsubseteq T'$, and the resulting heaps are also related by precision: $\Sigma_1; \Sigma_1' \vdash \mu_1 \sqsubseteq \mu_1'$.
(2) If $t' \mid \mu' \mid pc' \Downarrow_\epsilon err'$, there exists an error err s.t.

BBinary

$$\frac{t \mid \mu \mid pc \Downarrow (b_1)_{p_1} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow (b_2)_{p_2} \mid \mu_2}{t \oplus s \mid \mu \mid pc \Downarrow (b_1 [\![\oplus]\!] b_2)_{(p_1 \sqcup p_2)} \mid \mu_2}$$

BApp

$$\frac{\begin{array}{c} t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \\ s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad v \ \langle \hat{\mathcal{C}}_{arg} \rangle \longrightarrow v_1 \qquad (pc \sqcup p) \ \langle c \rangle \Rightarrow pc_1 \qquad t_1[x := v_1] \mid \mu_2 \mid pc_1 \Downarrow v_2 \mid \mu_3 \qquad v_2 \ \langle \hat{\mathcal{C}}_{ret} \rangle \longrightarrow v_3 \end{array}}{t \ s \mid \mu \mid pc \longrightarrow v_3 \sqcup p \mid \mu_3}$$

BIfTrue

$$\frac{t \mid \mu \mid pc \Downarrow \texttt{true}_p \mid \mu_1 \qquad s_1 \mid \mu_1 \mid pc \sqcup p \Downarrow v \mid \mu_2}{\texttt{if } t \texttt{ then } s_1 \texttt{ else } s_2 \mid \mu \mid pc \Downarrow v \sqcup p \mid \mu_2}$$

BIfFalse

$$\frac{t \mid \mu \mid pc \Downarrow \texttt{false}_p \mid \mu_1 \qquad s_2 \mid \mu_1 \mid pc \sqcup p \Downarrow v \mid \mu_2}{\texttt{if } t \texttt{ then } s_1 \texttt{ else } s_2 \mid \mu \mid pc \Downarrow v \sqcup p \mid \mu_2}$$

BRef

$$\frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad |pc| \preccurlyeq \ell \qquad \hat{\mathcal{C}} = \langle\!\langle R_\ell \precsim^{P} R_\ell \rangle\!\rangle \qquad o = \text{fresh}(\mu_1)}{\texttt{ref}^{P}_{R_\ell} \ t \mid \mu \mid pc \Downarrow o_\perp^{\ell, \hat{\mathcal{C}}\hat{\mathcal{C}}} \mid \mu_1[o \mapsto v]}$$

BDeref

$$\frac{t \mid \mu \mid pc \Downarrow o_p^{\ell, \hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad o \mapsto v \in \mu_1 \qquad v \ \langle \hat{\mathcal{C}}_{out} \rangle \longrightarrow v_1}{!t \mid \mu \mid pc \Downarrow v_1 \sqcup p \mid \mu_1}$$

BAssign

$$\frac{t \mid \mu \mid pc \Downarrow o_p^{\ell, \hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad o \mapsto v' \in \mu_2 \qquad v \ \langle \hat{\mathcal{C}}_{in} \rangle \longrightarrow v_1 \qquad |p| \curlyvee |pc| \preccurlyeq \ell}{t :=^{P} s \mid \mu \mid pc \Downarrow \texttt{unit}_\perp \mid \mu_2[o \mapsto v_1]}$$

BCast

$$\frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad v \ \langle \hat{\mathcal{C}} \rangle \longrightarrow v_1}{t \ \langle \hat{\mathcal{C}} \rangle \mid \mu \mid pc \Downarrow v_1 \mid \mu_1}$$

BLet

$$\frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad s[x := v] \mid \mu_1 \mid pc \Downarrow v_1 \mid \mu_2}{\texttt{let } x = t \texttt{ in } s \mid \mu \mid pc \Downarrow v_1 \mid \mu_2}$$

ERefNSU

$$\frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad |pc| \not\preccurlyeq \ell}{\texttt{ref}^{P}_{R_\ell} \ t \mid \mu \mid pc \Downarrow_\epsilon \texttt{nsu-error} \ P}$$

ECastBlame

$$\frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad v \ \langle \hat{\mathcal{C}} \rangle \longrightarrow \texttt{blame} \ P}{t \ \langle \hat{\mathcal{C}} \rangle \mid \mu \mid pc \Downarrow_\epsilon \texttt{blame} \ P}$$

EAssignNSU

$$\frac{t \mid \mu \mid pc \Downarrow o_p^{\ell, \hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad o \mapsto v' \in \mu_2 \qquad v \ \langle \hat{\mathcal{C}}_{in} \rangle \longrightarrow v_1 \qquad |p| \curlyvee |pc| \not\preccurlyeq \ell}{t :=^{P} s \mid \mu \mid pc \Downarrow_\epsilon \texttt{nsu-error} \ P}$$

Fig. 7. Reduction rules for big-step semantics of λ^p

$t \mid \mu \mid pc \Downarrow_\epsilon err$.

(3) If $t' \mid \mu' \mid pc' \Downarrow v' \mid \mu'_1$, then either (a) there exist Σ_1, Σ'_1, v and μ_1 s.t. $\Sigma_1 \sqsubseteq \Sigma'_1$, $t \mid \mu \mid pc \Downarrow v \mid \mu_1$, the resulting values are related by precision: $\emptyset; \emptyset; \Sigma_1; \Sigma'_1 \vdash v \sqsubseteq v' : T \sqsubseteq T'$, and the resulting heaps are also related by precision: $\Sigma_1; \Sigma'_1 \vdash \mu_1 \sqsubseteq \mu'_1$; or (b) there exists an error err s.t. $t \mid \mu \mid pc \Downarrow_\epsilon err$.

V. Gradual Surface Language λ^g

This section presents our gradual surface language λ^g.

A. Syntax and Compilation

Our gradual surface language λ^g conservatively extends $\lambda^\star_{\texttt{IFC}}$ with binary operators, such as arithmetic operations, relational comparisons, and Boolean connectives, while preserving the original lightweight security annotations of the source program. Programmers specify security labels using only lattice constants or the unknown label $\star$; partial labels and coercion labels from λ^p are completely hidden. The syntax of λ^g is given in Figure 8.

Surface-language terms of λ^g are compiled into cast-calculus terms of λ^p by decorating them with appropriate coercions and enriched types. Only well-typed terms are admitted to compilation. Formally, we define the type-and-compile judgement $\Gamma; gc \vdash t : T \rightsquigarrow t'$, which states that, under typing environment Γ and PC label gc, surface term t has type T and compiles to cast-calculus term t'. The core idea is that, each consistent-subtyping relation required by

$$
\begin{array}{rcl}
R & ::= & \texttt{Bool} \mid \texttt{Unit} \mid T \xrightarrow{g} S \mid \texttt{Ref}\ T \\
T, S & ::= & R_g \\
r & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{unit} \mid (\lambda^g x : T.t)^P \\
v & ::= & r_\ell \qquad \oplus \ ::= \ \wedge \mid \vee \\
t, s & ::= & x \mid v \mid (t\ s)^P \mid t \oplus s \mid (\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2)^P \\
& & \mid (\texttt{ref}_\ell\ t)^P \mid !t \mid (t := s)^P \mid (t :: T)^P \\
& & \mid \texttt{let}\ x = t\ \texttt{in}\ s
\end{array}
$$

Fig. 8. Syntax of gradual surface language λ^g

the typing derivation is realized as a runtime cast, annotated with a blame label for precise error attribution. To illustrate compilation, consider the term that casts function f from $\texttt{low} \xrightarrow{\texttt{low}} \texttt{low}$ to $\star \xrightarrow{\star} \star$ and applies it to a $\texttt{high}$ value:

$$let\ f = (\lambda^{\texttt{low}}(x : \texttt{low}).x)_{\texttt{low}} :: (\star \xrightarrow{\star} \star)\ in\ f\ true_{\texttt{high}}$$

This term is well-typed, as every consistent-subtyping obligation involves $\star$ and is therefore trivially satisfiable. During compilation, the necessary casts are realized as coercions with blame labels. Specifically, two casts arise: (i) the explicit cast of function f from $\texttt{low} \xrightarrow{\texttt{low}} \texttt{low}$ to $\star \xrightarrow{\star} \star$; and (ii) the implicit cast of the argument from $\texttt{high}$ to $\star$. The first cast decomposes into three coercions that correspond to argument, effect and return: $\star \Rightarrow \texttt{low}$ (*i.e.*, $\langle ?\ \texttt{low} \triangleright \texttt{low}\rangle$), $\star \Rightarrow \texttt{low}$ (*i.e.*, $\langle ?\ \texttt{low} \triangleright \texttt{low}\rangle$), and $\texttt{low} \Rightarrow \star$ (*i.e.*, $\langle \texttt{low} \triangleright \texttt{low}\ ?\rangle$). For the second cast, the coercion label is $\langle \texttt{high} \triangleright \texttt{high}\ ?\rangle$. Consequently, the original term compiles to the following term

$$
\begin{array}{l}
let\ f = (\lambda^{\langle ?\ \texttt{low}\triangleright\texttt{low}\rangle}(x : \texttt{low}).x)_{\texttt{low}}^{\langle ?\ \texttt{low}\triangleright\texttt{low}\rangle,\langle\texttt{low}\triangleright\texttt{low}\ ?\rangle} \\
in\ f\ (true_{\texttt{high}}\langle \texttt{high} \triangleright \texttt{high}\ ?\rangle)
\end{array}
$$

B. Meta Theoretic Results of λ^g

Similar to their cast-calculus counterparts, surface terms are related by an observation-indexed equivalence judgement $\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : T$, where ℓ_o is the observation label. Theorem 5.1 guarantees that compilation maps equivalent surface terms to equivalent cast-calculus terms.

Theorem 5.1 (Compilation Preserves Equivalence): Let t_1, t_2 be two surface terms. If they are equivalent $\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : T$, then there exist two cast-calculus terms s_1, s_2 s.t. $\emptyset; gc \vdash t_i : T \rightsquigarrow s_i$, and $\emptyset; \Gamma; \emptyset; \emptyset; gc \vdash s_1 \approx^{\ell_o} s_2 : T$.

The noninterference theorem of λ^g is a straightforward corollary of the noninterference theorem for λ^p (Theorem 4.1).

Theorem 5.2 (NI): Suppose t_1, t_2 are equivalent surface terms $\emptyset; gc \vdash t_1 \approx^{\ell_o} t_2 : T$, pc_1, pc_2 are equivalent $pc_1 \approx^{\ell_o} pc_2 : gc$, and t_1, t_2 are compilable $\emptyset; gc \vdash t_i : T \rightsquigarrow s_i$. If $s_1 \mid \emptyset \mid pc \Downarrow v_1 \mid \mu_1$ and $s_2 \mid \emptyset \mid pc \Downarrow v_2 \mid \mu_2$, then there exist A, Σ_1, Σ_2 s.t. values and heaps preserve equivalence $A; \emptyset; \Sigma_1; \Sigma_2 \vdash v_1 \approx^{\ell_o} v_2 : T$ and $A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell_o} \mu_2$.

Similar to their cast-calculus counterparts, surface-term precision is defined structurally: every corresponding sub-term must stand in the precision relations on partial and coercion labels.

Theorem 5.3 (Compilation Preserves Precision): Let t, t' be two surface terms. If $\Gamma \sqsubseteq \Gamma'$, $gc \sqsubseteq gc'$, $t \sqsubseteq t'$, $\Gamma; gc \vdash t : T \rightsquigarrow s$, $\Gamma'; gc' \vdash t' : T' \rightsquigarrow s'$ then $\Gamma; \Gamma'; \emptyset; \emptyset; gc; gc' \vdash s \sqsubseteq s' : T \sqsubseteq T'$.

We first establish the SGG for λ^g: decreasing the precision of a surface term preserves well-typedness and yields a type that is equally or less precise.

Theorem 5.4 (SGG): Suppose that two surface terms t, t' are related by precision: $t \sqsubseteq t'$, two gradual labels gc, gc' are related by precision: $gc \sqsubseteq gc'$, two type contexts Γ, Γ' are also related by precision: $\Gamma \sqsubseteq \Gamma'$. If $\Gamma; gc \vdash t : T$, then there exists a type T' s.t. $\Gamma'; gc' \vdash t' : T'$ and $T \sqsubseteq T'$.

Finally, we address the DGG for λ^g. Similarly to NI, the DGG of λ^g can be proved as a corollary of the DGG of λ^p (Theorem 4.2) and compilation preserves precision.

Theorem 5.5 (DGG): Suppose that two surface terms t, t' are related by precision: $t \sqsubseteq t'$, two partial labels pc, pc' are related by precision: $\vdash pc \sqsubseteq pc' : gc \sqsubseteq gc'$, both terms t, t' are compilable: $\emptyset; gc \vdash t : T \rightsquigarrow t_1$ and $\emptyset; gc' \vdash t' : T' \rightsquigarrow t_1'$.

(1) If $t_1 \mid \mu \mid pc \Downarrow v \mid \mu_1$, there exist Σ_1, Σ_1', v' and μ_1' s.t. $\Sigma_1 \sqsubseteq \Sigma_1'$, $t_1' \mid \mu' \mid pc' \Downarrow v' \mid \mu_1'$, the resulting terms are related by precision: $\emptyset; \emptyset; \Sigma_1; \Sigma_1' \vdash v \sqsubseteq v' : T \sqsubseteq T'$, and the resulting heaps are also related by precision: $\Sigma_1; \Sigma_1' \vdash \mu_1 \sqsubseteq \mu_1'$.

(2) If $t_1' \mid \mu' \mid pc' \Downarrow_\epsilon err'$, there exists an error err s.t. $t_1 \mid \mu \mid pc \Downarrow_\epsilon err$.

(3) If $t_1' \mid \mu' \mid pc' \Downarrow v' \mid \mu_1'$, then either (a) there exist Σ_1, Σ_1', v and μ_1 s.t. $\Sigma_1 \sqsubseteq \Sigma_1'$, $t_1 \mid \mu \mid pc \Downarrow v \mid \mu_1$, the resulting values are related by precision: $\emptyset; \emptyset; \Sigma_1; \Sigma_1' \vdash v \sqsubseteq v' : T \sqsubseteq T'$, and the resulting heaps are also related by precision: $\Sigma_1; \Sigma_1' \vdash \mu_1 \sqsubseteq \mu_1'$; or (b) there exists an error err s.t. $t_1 \mid \mu \mid pc \Downarrow_\epsilon err$.

C. Unknown References

A limitation of our calculus is its lack of support for unknown references, due to the tension between DGG and NI [26]. In this section, we discuss how our calculus accommodates unknown references, while sacrificing DGG.

Unknown references become desirable when we wish to import legacy code whose heap locations carry no security labels. To accommodate them, we simply revisit the syntax concerning references by substituting a static label ℓ with a gradual label g:

$$r \ ::= \ \cdots \mid o^{g, \hat{c}_{in} \hat{c}_{out}} \mid \cdots \qquad t \ ::= \ \cdots \mid \texttt{ref}_{R_g}^{P}\ t \mid \cdots$$

The typing rules for heap objects (Rule (TREF)) and reference terms (Rule (TPTR)) are derived by substituting the static label ℓ with a gradual label g. The operational semantics for heap updates (Rule (BASSIGN)) can be correspondingly adapted, incorporating the side condition: $|p| \curlyvee |pc| \preccurlyeq g$. When g is $\star$, this side condition is trivially satisfied. Consequently, a high PC may silently influence an unknown reference and subsequently leak sensitive information through a low channel, breaking NI. To illustrate this issue, consider the following example with two distinct execution runs where x is typed as $\texttt{Ref Bool}_{\texttt{high}}$, where T and F respectively denote $\texttt{true}$ and $\texttt{false}$. In the first run, only the assignment in line 4 is executed, while only the assignment in line 5 occurs during the second run. As indicated by the results of low variable z, these two runs violate NI.

```
1  //R1: [x ↦ T_high]            R2: [x ↦ F_high]
2  let y: Ref Bool_⋆ = ref true_low
3  let z: Ref Bool_low = ref true_low
4  if !x then y := false_low
5  if !y then z := false_low
6  //R1: [·; y ↦ F_low; z ↦ T_low]  R2: [·; y ↦ T_low; z ↦ F_low]
```

To enforce NI, we maintain the initial partial label at allocation time and require that the PC label is lower than this label and the reference content is upgraded with this label. Intuitively, this initial label acts as a safeguard (as highlighted), preventing the reference from being modified under any PC label that is not lower than it.

BREF

$$\frac{|pc| \preccurlyeq |p| \qquad \begin{array}{c} t \mid \mu \mid pc \Downarrow r_p \mid \mu_1 \\ \hat{\mathcal{C}} = \langle\!\langle R_g \precsim^P R_g \rangle\!\rangle \end{array} \qquad o = \mathrm{fresh}(\mu_1)}{\texttt{ref}^P_{R_g}\ t \mid \mu \mid pc \Downarrow o_\perp^{g,\hat{\mathcal{C}}\hat{\mathcal{C}}} \mid \mu_1[o \mapsto^p r_p]}$$

BASSIGN

$$\frac{\begin{array}{cc} t \mid \mu \mid pc \Downarrow o_{p'}^{g',\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 & s \mid \mu_1 \mid pc \Downarrow v' \mid \mu_2 \end{array} \\ o \mapsto^p v \in \mu_2 \qquad v' \ \langle \hat{\mathcal{C}}_{in} \rangle \longrightarrow v_1 \qquad |p'| \curlyvee |pc| \preccurlyeq |p|}{t :=^P s \mid \mu \mid pc \Downarrow \texttt{unit}_\perp \mid \mu_2[o \mapsto^p v_1 \sqcup p]}$$

With this guard in place, line 4 of the first run must satisfy high $\preccurlyeq$ |low ?|, which fails. If we instead initialise y as ref $\texttt{true}_{\texttt{high}}$, the first run proceeds, but line 5 of the second run now requires |high ?| $\preccurlyeq$ low, which also fails. Thus neither execution can transmit the secret to the low variable z, and NI is preserved.

While GSL_{Ref} [26] requires the label of the original reference content to be higher than the PC label, our approach trades some flexibility for practicality: the initial label can be derived directly from the programmer's annotation or the initial value assignment and need only be annotated once, rather than at every assignment. Moreover, our solution is more amenable to mechanization, particularly for establishing heap-observed equivalence in Agda.

Like GSL_{Ref}, our solution sacrifices the DGG. Consider these two programs: the gradual program creates an unknown reference y and subsequent updates it, while the static program creates a static reference y and subsequent updates it.

```
//Program 1: gradual
let x: Ref Bool_high = ref true_high
let y: Ref Bool_⋆ = ref true_low
if !x then y := false_low // fail
//Program 2: static
let x: Ref Bool_high = ref true_high
let y: Ref Bool_high = ref true_low
if !x then y := false_low
```

The guard |high| $\not\preccurlyeq$ |low ?| causes the gradual program to abort, while the static program executes successfully. Thus, erasing precision (replacing high by $\star$) introduces a new run-time failure, violating DGG.

VI. SEMANTIC INTERPRETATION

Abstract Gradual Typing (AGT) [34] is a framework that applies abstract interpretation at the type level to systematically derive gradually typed languages from existing statically typed languages. In this section, we present a semantic interpretation of our partial labels, which facilitates the application of AGT.

Syntactically, $\top \sqsubseteq \top\ ?$, but the converse fails, since the term $\texttt{true}_\top :: \top$ is strictly more precise than $\texttt{true}_\top :: \star$. Semantically, however, $\top\ ?$ denotes *at least* $\top$, and since $\top$ is the greatest element in the security lattice, $\top\ ?$ must coincide with $\top$, that is, $\top \sqsubseteq \top\ ?$ and $\top\ ? \sqsubseteq \top$. This creates a mismatch between syntactic and semantic precision[8]. Introducing a syntactic rule $\top\ ? \sqsubseteq \top$ would resolve it, but at the cost of exposing an unintuitive rule to programmers. Instead, we introduce a semantic-only label $\blacktriangledown$ invisible in surface syntax, preserving both intuitive syntactic relations and sound semantics.

Given a security lattice $(\textsc{Label}, \preccurlyeq, \curlyvee, \curlywedge)$, we introduce a special label denoted by $\blacktriangledown$ with $\top \preccurlyeq \blacktriangledown$. A label interval I is defined as follows:

$$I \ ::= \ [\ell, \ell] \mid [\ell, \blacktriangledown] \qquad \text{where } \ell \in \textsc{Label}$$

Partial labels can be interpreted as intervals of security labels. Specifically, a concretization function γ is defined:

$$\gamma(\ell) \ = \ [\ell, \ell] \qquad \gamma(\ell\ ?) \ = \ [\ell, \blacktriangledown]$$

Based on this interpretation, the precision relation $\sqsubseteq$ between partial labels can be defined semantically:

$$p_1 \sqsubseteq p_2 \text{ if and only if } \gamma(p_1) \subseteq \gamma(p_2).$$

Similarly, an abstraction function α mapping intervals into partial labels is defined:

$$\alpha([\ell, \ell]) \ = \ \ell \qquad \alpha([\ell, \blacktriangledown]) \ = \ \ell\ ?$$

We can prove γ and α forms a Galois connection: (1) $\forall p.\ \alpha(\gamma(p)) \sqsubseteq p$; (2) $\forall I.\ I \subseteq \gamma(\alpha(I))$.

The join of two partial labels p_1, p_2 can be defined:

$$p_1 \sqcup p_2 = \alpha(\{\ell_1 \curlyvee \ell_2 \ \mid\ \ell_i \in \gamma(p_i)\}).$$

To semantically define the cast of a partial label p with a gradual label g, we extend the concretization function γ to accommodate gradual labels: $\gamma(\star) = \gamma(\perp\ ?)$. The cast can be defined as follows:

$$p\ \langle g \rangle \Rightarrow \alpha([\ell_1^g \curlyvee \ell_1^p, \ell_2^g]) \text{ if } \exists \ell \in \gamma(p), \ell' \in \gamma(g) \ s.t.\ \ell \preccurlyeq \ell'$$

where $\gamma(p) = [\ell_1^p, \ell_2^p]$ and $\gamma(g) = [\ell_1^g, \ell_2^g]$.

Likewise, reverse partial labels can be interpreted as intervals of security labels as well. A concretization function $\hat{\gamma}$ and an abstraction function $\hat{\alpha}$ can be defined accordingly:

$$\hat{\gamma}(?\ \ell) \ = \ [\blacktriangle, \ell] \qquad \hat{\alpha}([\blacktriangle, \ell]) \ = \ ?\ \ell$$

where $\blacktriangle$ is a specific label with $\blacktriangle \preccurlyeq \perp$. We can prove $\hat{\gamma}$ and $\hat{\alpha}$ forms a Galois connection as well.

Based on the interpretation of partial labels, we define an interpretation for coercion labels. For simplicity, only valid coercion labels are considered. Specifically, an interpretation

[8] We assume that $\top$ is visible to programmers; otherwise, this issue does not arise, and $\top$ is used in place of $\blacktriangledown$.

of a coercion label is represented as a pair of two label intervals:

$$\overline{\gamma}(\langle \hat{p} \triangleright p \rangle) = (\hat{\gamma}(\hat{p}), \gamma(p))$$

The first component denotes the range applicable under the coercion label, while the second component indicates the range that may be returned from it. With this interpretation, the precision relation $\sqsubseteq$ between coercion labels can be defined:

$$\langle \hat{p}_1 \triangleright p_1 \rangle \sqsubseteq \langle \hat{p}_2 \triangleright p_2 \rangle \text{ iff } \hat{\gamma}(\hat{p}_1) \subseteq \hat{\gamma}(\hat{p}_2) \text{ and } \gamma(p_1) \subseteq \gamma(p_2)$$

The cast of a partial label p' with a coercion label $\langle \hat{p} \triangleright p \rangle$ is defined:

$$p' \langle \hat{p} \triangleright p \rangle \Rightarrow \alpha([\ell_1^p \curlyvee \ell_1^{p'}, \ell_2^p]) \text{ if } \exists \ell \in \gamma(p'), \ell' \in \hat{\gamma}(\hat{p}) \; s.t. \; \ell \preccurlyeq \ell'$$

where $\gamma(p') = [\ell_1^{p'}, \ell_2^{p'}]$, $\hat{\gamma}(\hat{p}) = [\ell_1^{\hat{p}}, \ell_2^{\hat{p}}]$, and $\gamma(p) = [\ell_1^p, \ell_2^p]$. The composition of coercion labels can be defined similarly.

Compared to GSL_{Ref} [26], our approach to semantic interpretation exhibits three main differences: First, our approach uses specific labels ▼ and ▲ to clarify some subtle partial types, thereby preserving an intuitive syntactic relation. Second, our approach provides interpretations not only for runtime labels (*i.e.*, partial labels) but also for cast sequences (*i.e.*, coercion labels). Third, our approach requires only two types of intervals: $[\ell, ▼]$ (indicating at least ℓ) and $[▲, \ell]$ (indicating at most ℓ), resulting in a more streamlined system. This simplification arises from the concise representation of valid cast sequences. Specifically, there are three types of valid cast sequences in IFC: ascending sequences for covariant scenarios, descending sequences for contravariant scenarios, or equivalent sequences for invariant scenarios. An equivalent sequence comprises a pair of an ascending sequence and a descending sequence (as shown in references). A descending sequence is essentially the reserve of an ascending sequence (as demonstrated in function arguments). Therefore, only ascending sequences suffice and can be concisely represented as $(source, GLB, LUB, target)$. Moreover, for a valid ascending sequence, two specific cases need to be addressed: casting the unknown source $\star$ to the GLB ℓ or casting the LUB ℓ to the unknown target $\star$. Other cases involving an unknown label $\star$ can be verified immediately. For instance, $\star$ in $\overrightarrow{\texttt{low}; \star; \texttt{high}}$ can be verified immediately.

This streamlined interpretation could facilitates the application of AGT, which will be explored in future work.

VII. Related Work

Disney and Flanagan [21] studied gradual security types for a pure lambda calculus, which is more akin to a cast calculi than to a gradual language. Fennell and Thiemann proposed ML-GS [22], a monomorphic ML core language with references and higher-order functions that implements gradual typing for IFC, and then extended ML-GS to the object-oriented setting in the language LJGS [24]. Nevertheless, none of the aforementioned studies have addressed the gradual guarantees [17].

GSL_{Ref} [26] is a gradually typed higher-order language with references, based on abstract gradual typing (AGT) [34]. Due to the tension between them, GSL_{Ref} sacrifices dynamic gradual guarantee (DGG) in favor of achieving noninterference (NI). Our language λ^g resembles GSL_{Ref}. However, instead of using evidences (*i.e.*, interval pairs), we employ a more concise representation *partial label* as runtime monitors. This adjustment simplifies operators related to security labels within λ^g. Moreover, we adopt the design principle from $\lambda^\star_{\texttt{IFC}}$, which restricts references to associate with static labels. This restriction enables λ^g to satisfy both DGG and NI simultaneously. Additionally, our partial and coercion labels can be interpreted as intervals as well. This may enhance the application of AGT in security types, facilitating the development of a language that meets both DGG and NI.

GLIO [28] is a gradual IFC language that supports higher-order functions, general references, and coarse-grained information flow control with first-class labels. While it is expressive and upholds both NI and DGG, it compromises on type-guided classification. In contrast, our language λ^g, akin to GSL_{Ref} and $\lambda^\star_{\texttt{IFC}}$, adopts a fine-grained security type system. Prior research [40], [41] has established logical relations between coarse-grained and fine-grained security type systems within both static and dynamic contexts. Concerning gradual scenarios, there may exist potential logical relationships between GLIO and λ^g, which remains an area for future exploration.

WHILE^{G} [30] is a simple imperative language that supports gradual security typing. To resolve the tension between the DGG and NI when dealing with mutable reference, WHILE^{G} effectively utilizes the static phase of its gradual type system to identify the set of variables being written to in different branches and loops, and refines their possible security labels to implement a dynamic monitoring strategy. However, this approach compromises NSU checking and may not be applicable to functional languages due to the presence of high order functions. In contrast, our language λ^g adopts the design principle from $\lambda^\star_{\texttt{IFC}}$, which restricts references to associate with static labels. This enables λ^g to fulfill both NI and DGG while maintaining effective NSU checking.

$\lambda^\star_{\texttt{SEC}}$ [29] provides the first mechanized proof of noninterference in a language featuring both static and gradual control. Built on GLIO, $\lambda^\star_{\texttt{SEC}}$ also does not change the runtime label for casting, resulting in a loss of type-guided classification.

$\lambda^\star_{\texttt{IFC}}$ [31] is a novel gradual security-typed language that satisfies both NI and DGG without making any compromises on type-guided classification and NSU. However, $\lambda^\star_{\texttt{IFC}}$ suffers from scalability issues and lacks binary operators. In contrast, our *partial label* offers concise representations for runtime monitors, allowing our calculus to be easily extended to accommodate more complex security label sets while effectively facilitating binary operators. Furthermore, all theorem proofs are mechanized in Agda.

Finally, label polymorphsim [42] relieves programmers from writing explicit security annotations by abstracting over labels, whereas gradual IFC abstracts *from* labels, allowing unknown labels to be refined later. Faceted semantics [43] also handles uncertainty about security labels. It encapsulates multiple labels within a single execution; gradual IFC instead allows values to carry the unknown label $\star$ and enforces security through runtime casts.

VIII. Conclusion

We have introduced *partial label*, a concise representation for security coercions and runtime labels, facilitating the development of a coercion calculus for security labels within gradual security-typed languages. We have presented λ^g, a gradual security-typed language, together with its associated partial cast calculus λ^p; both satisfy the gradual guarantee and noninterference. Our calculus naturally supports richer security lattices and binary operators. We have further explored extensions that accommodate unknown references and provided a semantic interpretation of partial labels. All proofs of the lemmas and theorems have been fully mechanized in Agda.

References

[1] N. Heintze and J. G. Riecke, "The slam calculus: Programming with secrecy and integrity," in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, pp. 365–377.

[2] A. C. Myers, "Jflow: Practical mostly-static information flow control," in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pp. 228–241.

[3] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," J. Comput. Secur., vol. 4, no. 2/3, pp. 167–188, 1996.

[4] Z. Xu, H. Chen, A. Tiu, Y. Liu, and K. Sareen, "A permission-dependent type system for secure information flow analysis," J. Comput. Secur., vol. 29, no. 2, pp. 161–228, 2021.

[5] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, 2009, pp. 113–124.

[6] ——, "Multiple facets for dynamic information flow," in Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2012, pp. 165–178.

[7] T. H. Austin, T. Schmitz, and C. Flanagan, "Multiple facets for dynamic information flow with exceptions," ACM Trans. Program. Lang. Syst., vol. 39, no. 3, pp. 10:1–10:56, 2017.

[8] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," J. Funct. Program., vol. 27, p. e5, 2017.

[9] J. Xiang and S. Chong, "Co-inflow: Coarse-grained information flow control for java-like languages," in 42nd IEEE Symposium on Security and Privacy, 2021, pp. 18–35.

[10] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From fine- to coarse-grained dynamic information flow control and back," Found. Trends Program. Lang., vol. 8, no. 1, pp. 1–117, 2023.

[11] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a java virtual machine," in 23rd Annual Computer Security Applications Conference, 2007, pp. 463–475.

[12] G. L. Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in 20th IEEE Computer Security Foundations Symposium, 2007, pp. 218–232.

[13] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in Proceedings of the 24th IEEE Computer Security Foundations Symposium, 2011, pp. 146–160.

[14] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in Proceedings of the 23rd IEEE Computer Security Foundations Symposium, 2010, pp. 186–199.

[15] P. Shroff, S. F. Smith, and M. Thober, "Dynamic dependency monitoring to secure information flow," in 20th IEEE Computer Security Foundations Symposium, 2007, pp. 203–217.

[16] P. Li and S. Zdancewic, "Arrows for secure information flow," Theor. Comput. Sci., vol. 411, no. 19, pp. 1974–1994, 2010.

[17] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland, "Refined criteria for gradual typing," in 1st Summit on Advances in Programming Languages, vol. 32, 2015, pp. 274–293.

[18] G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek, "Gradual typing: a new perspective," Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 16:1–16:32, 2019.

[19] M. S. New, D. R. Licata, and A. Ahmed, "Gradual type theory," Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 15:1–15:31, 2019.

[20] J. P. C. III, M. W. Khan, and S. Chen, "Type-based gradual typing performance optimization," Proc. ACM Program. Lang., vol. 8, no. POPL, pp. 2667–2699, 2024.

[21] T. Disney and C. Flanagan, "Gradual information flow typing," in International workshop on scripts to programs, 2011.

[22] L. Fennell and P. Thiemann, "Gradual security typing with references," in Proceedings of the 26th IEEE Computer Security Foundations Symposium, 2013, pp. 224–239.

[23] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: mixing static and dynamic typing for information-flow control in haskell," in Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, 2015, pp. 289–301.

[24] L. Fennell and P. Thiemann, "LJGS: gradual security types for object-oriented languages," in 30th European Conference on Object-Oriented Programming, vol. 56, 2016, pp. 9:1–9:26.

[25] T. Chen and J. G. Siek, "Mechanized type safety for gradual information flow," in IEEE Security and Privacy Workshops, 2021, pp. 194–206.

[26] M. Toro, R. Garcia, and É. Tanter, "Type-driven gradual security with references," ACM Trans. Program. Lang. Syst., vol. 40, no. 4, pp. 16:1–16:55, 2018.

[27] A. Bichhawat, M. McCall, and L. Jia, "First-order gradual information flow types with gradual guarantees," CoRR, vol. abs/2003.12819, 2020.

[28] A. A. de Amorim, M. Fredrikson, and L. Jia, "Reconciling noninterference and gradual typing," in Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, 2020, pp. 116–129.

[29] T. Chen and J. G. Siek, "Mechanized noninterference for gradual security," CoRR, vol. abs/2211.15745, 2022.

[30] A. Bichhawat, M. McCall, and L. Jia, "Gradual security types and gradual guarantees," in Proceedings of the 34th IEEE Computer Security Foundations Symposium, 2021, pp. 1–16.

[31] T. Chen and J. G. Siek, "Quest complete: The holy grail of gradual security," Proc. ACM Program. Lang., vol. 8, no. PLDI, pp. 1609–1632, 2024.

[32] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," IEEE J. Sel. Areas Commun., vol. 21, no. 1, pp. 5–19, 2003.

[33] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in Software Safety and Security, 2012, vol. 33, pp. 319–347.

[34] R. Garcia, A. M. Clark, and É. Tanter, "Abstracting gradual typing," in Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp. 429–442.

[35] F. B. Schwerter, A. M. Clark, K. A. Jafery, and R. Garcia, "Abstracting gradual typing moving forward: precise and space-efficient," Proc. ACM Program. Lang., vol. 5, no. POPL, pp. 1–28, 2021.

[36] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," J. Comput. Secur., vol. 17, no. 5, pp. 517–548, 2009.

[37] J. G. Siek and W. Taha, "Gradual typing for objects," in Proceedings of the 21st European Conference on Object-Oriented Programming, vol. 4609, 2007, pp. 2–27.

[38] J. G. Siek and P. Wadler, "Threesomes, with and without blame," in Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2010, pp. 365–376.

[39] J. G. Siek, R. Garcia, and W. Taha, "Exploring the design space of higher-order casts," in 18th European Symposium on Programming, vol. 5502, 2009, pp. 17–31.

[40] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in 31st IEEE Computer Security Foundations Symposium, 2018, pp. 233–246.

[41] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From fine- to coarse-grained dynamic information flow control and back," Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 76:1–76:31, 2019.

[42] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," International Journal of Information Security, vol. 6, no. 2, pp. 67–84, 2007.

[43] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2012, pp. 165–178.

APPENDIX

The Appendix is organized as follows: Section A presents some properties for coercion labels and partial cast calculus. Section B presents definitions and rules for typing, reduction, compilation.

A. Meta Theoretic Results

1) Properties for Coercion Labels: We demonstrate that the cast calculus of coercion labels also adheres to the flow safety (Theorem A.1), dynamic gradual guarantee (Theorem A.2), and type-based classification (Theorem A.3).

Theorem A.1 (Flow safety for coercion labels): Let p, p' be two runtime labels, and c a coercion label. If $p\ \langle c\rangle \Rightarrow p'$, then $|p| \preccurlyeq |p'|$.

Theorem A.2 (Dynamic gradual guarantee for coercion labels): Let p_1, p_1' be two runtime labels, and c, c' be two coercion labels. Suppose that $\vdash p_1 \sqsubseteq p_1' : g_1 \sqsubseteq g_1'$ and $\vdash c \sqsubseteq c' : g_1 \Rightarrow g_2 \sqsubseteq g_1' \Rightarrow g_2'$.

(1) If $p_1\ \langle c\rangle \Rightarrow p_2$, then there exists a runtime label p_2' s.t. $p_1'\ \langle c'\rangle \Rightarrow p_2'$ and $\vdash p_2 \sqsubseteq p_2' : g_2 \sqsubseteq g_2'$
(2) If $p_1'\ \langle c'\rangle \Rightarrow \texttt{blame}$, then $p_1\ \langle c\rangle \Rightarrow \texttt{blame}$.
(3) If $p_1'\ \langle c'\rangle \Rightarrow p_2'$, then either there exists a runtime label p_2 s.t. $p_1\ \langle c\rangle \Rightarrow p_2$ and $\vdash p_2 \sqsubseteq p_2' : g_2 \sqsubseteq g_2'$, or $p_1\ \langle c\rangle \Rightarrow \texttt{blame}$.

Theorem A.3 (Type-based classification for coercion labels): Let p, p' be two runtime labels, c a coercion label, and g, g', g'' three gradual labels. If $\vdash p : g$, $\vdash c : g'' \Rightarrow g'$ and $p\ \langle c\rangle \Rightarrow p'$, then $\vdash p' : g'$ and $g \precsim g''$.

2) Properties for Partial Cast Calculus: We demonstrate the type preservation property (Theorem A.4) of the partial cast calculus.

Theorem A.4 (Preservation): Suppose $\emptyset; \Sigma; gc \vdash t : T$, $\vdash pc : gc$, and $\Sigma \vdash \mu$. If $t \mid \mu \mid pc \Downarrow v \mid \mu'$, then there exists Σ' such that $\Sigma \subseteq \Sigma'$, $\Sigma' \vdash \mu'$, and $\emptyset; \Sigma'; gc \vdash v : T$.

Lemma A.5 establishes the properties of the heap equivalence, allowing the value equivalence to align with the address equivalence.

Lemma A.5 (Properties of Heap Observed Equivalence): Suppose two heaps μ_1, μ_2 are observed equivalent $A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell_o} \mu_2$,

(1) If two addresses o_1, o_2 are in the context $(o_1, o_2) \in^E A$, then there exist two well-typed values v_1, v_2 in the heap and they are also equivalence $A; \emptyset; \Sigma_1; \Sigma_2 \vdash v_1 \approx^{\ell_o} v_2 : R_\ell$, and $\ell \preccurlyeq \ell_o$.
(2) If an address o_1 is in the context $o_1 \in^L A$, then there exists a well-typed value v_1 in the heap $\emptyset; \Sigma_1; \perp \vdash v_1 : R_\ell$, and $\ell \not\preccurlyeq \ell_o$.
(3) If an address o_2 is in the context $o_2 \in^R A$, then there exists a well-typed value v_2 in the heap $\emptyset; \Sigma_2; \perp \vdash v_2 : R_\ell$, and $\ell \not\preccurlyeq \ell_o$.

Programs operating under high PC labels may exhibit control flow variations. Lemma A.6 proves that heap equivalence is preserved across different control flows when PC labels are unobservable. This behavior occurs specifically when IF statements contain unobservable conditions, allowing equivalent programs to follow completely different execution branches under high PC labels.

Lemma A.6 (Simulation under High PC Labels): Given an observer label ℓ_o and two observed equivalent heaps $A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell} \mu_2$, suppose that two well-typed terms $\emptyset; \Sigma_i; gc_i \vdash t_i : T_i$ are evaluated $t_i \mid \mu_i \mid pc_i \Downarrow v_i \mid \mu_i'$ in two unobservable PC labels $|pc_i| \not\preccurlyeq \ell_o$, then there exist A', Σ_1', Σ_2' s.t. heaps preserve equivalence $A'; \Sigma_1'; \Sigma_2' \vdash \mu_1' \approx^{\ell} \mu_2'$.

The following lemmas are to preserve the precision relation among types or values within their corresponding contexts or heaps. This facilitates a formal proof of DGG.

Lemma A.7 (Typing Context Precision): Let Γ and Γ' be two typing contexts related by precision relation $\Gamma \sqsubseteq \Gamma'$. For any variable x, if $x : T \in \Gamma$, then there exists T' s.t. $x : T' \in \Gamma'$ and $T \sqsubseteq T'$.

Lemma A.8 (Heap Context Precision): Let Σ and Σ' be two heap contexts related by precision relation $\Sigma \sqsubseteq \Sigma'$. (1) For any address o, if $o : T \in \Sigma$, then there exists T' s.t. $o : T' \in \Sigma'$ and $T \sqsubseteq T'$. (2) For any address o', if $o' : T' \in \Sigma'$, then there exists T s.t. $o' : T \in \Sigma$ and $T \sqsubseteq T'$.

Lemma A.9 (Heap Precision): Let μ and μ' be two heap contexts related by precision relation $\Sigma; \Sigma' \vdash \mu \sqsubseteq \mu'$. For any address o, if $o : T \in \Sigma$ and $o : T' \in \Sigma'$, then there exist v and v' s.t. $o \mapsto v \in \mu$, $o \mapsto v' \in \mu'$, and $\emptyset; \emptyset; \Sigma; \Sigma' \vdash v \sqsubseteq v' : T \sqsubseteq T'$.

B. Definitions and Rules for Typing, Reduction, Compilation

Figure 9 presents common definitions for calculus. Figure 10 presents the simplified formalization of several existing cast calculus for security labels. Figure 11 presents precision, typing and security label of reverse partial labels. Figure 12 presents precision relation for types. Figure 13 presents definitions related to type coercion. Figure 14 and 15 present blame reduction rules for big-step semantics. Figure 16 presents precision relation for surface terms. Figure 17 presents rules for typing and compilation of surface language. Figure 18 presents the observed equivalent definition of heaps. Figures 19 and 20 include the observed equivalent definitions of PLabel, terms, values, and raw values. Figure 21 presents observed Equivalence for surface language. Figure 22 presents precision relation for terms of partial cast calculus. Figure 23 presents precision relation for values and coercions of partial cast calculus.

$\boxed{g \sim g'}$ $\boxed{T \sim T}$ $\boxed{R \sim R}$

$$\frac{}{\star \sim \ell} \qquad \frac{}{\ell \sim \star} \qquad \frac{}{\ell \sim \ell} \qquad \frac{R \sim R' \qquad g \sim g'}{R_g \sim R'_{g'}} \qquad \frac{}{\texttt{Bool} \sim \texttt{Bool}} \qquad \frac{}{\texttt{Unit} \sim \texttt{Unit}} \qquad \frac{T \sim T'}{\texttt{Ref}\ T \sim \texttt{Ref}\ T'}$$

$$\frac{T \sim T' \qquad g \sim g' \qquad S \sim S'}{T \xrightarrow{g} S \sim T' \xrightarrow{g'} S'}$$

$\boxed{g \precsim g'}$ $\boxed{R \precsim R'}$ $\boxed{T \precsim T'}$

$$\frac{}{\star \precsim g} \qquad \frac{}{g \precsim \star} \qquad \frac{\ell \preccurlyeq \ell'}{\ell \precsim \ell'} \qquad \frac{}{\texttt{Bool} \precsim \texttt{Bool}} \qquad \frac{}{\texttt{Unit} \precsim \texttt{Unit}} \qquad \frac{T \precsim T' \qquad T' \precsim T}{\texttt{Ref}\ T \precsim \texttt{Ref}\ T'}$$

$$\frac{T' \precsim T \qquad g' \precsim g \qquad S \precsim S'}{T \xrightarrow{g} S \precsim T' \xrightarrow{g'} S'} \qquad \frac{R \precsim R' \qquad g \precsim g'}{R_g \precsim R'_{g'}}$$

$\boxed{g \mathbin{\widetilde{\curlyvee}} g' = g''}$ $\boxed{g \mathbin{\widetilde{\curlywedge}} g' = g''}$

$$\star \mathbin{\widetilde{\curlyvee}} g = \star \qquad g \mathbin{\widetilde{\curlyvee}} \star = \star \qquad \ell \mathbin{\widetilde{\curlyvee}} \ell' = \ell \curlyvee \ell' \qquad \star \mathbin{\widetilde{\curlywedge}} g = \star \qquad g \mathbin{\widetilde{\curlywedge}} \star = \star \qquad \ell \mathbin{\widetilde{\curlywedge}} \ell' = \ell \curlywedge \ell'$$

$\boxed{g \wedge g' = g''}$

$$\star \wedge g = g \qquad g \wedge \star = g \qquad \ell \wedge \ell = \ell$$

$\boxed{T \mathbin{\widetilde{\curlyvee}} T' = T''}$ $\boxed{T \mathbin{\widetilde{\curlywedge}} T' = T''}$ $\boxed{R \mathbin{\widetilde{\curlyvee}} R' = R''}$ $\boxed{R \mathbin{\widetilde{\curlywedge}} R' = R''}$

$$R_g \mathbin{\widetilde{\curlyvee}} R'_{g'} = (R \mathbin{\widetilde{\curlyvee}} R')_{g \mathbin{\widetilde{\curlyvee}} g'} \qquad R_g \mathbin{\widetilde{\curlywedge}} R'_{g'} = (R \mathbin{\widetilde{\curlywedge}} R')_{g \mathbin{\widetilde{\curlywedge}} g'} \qquad \texttt{Bool} \mathbin{\widetilde{\curlyvee}} \texttt{Bool} = \texttt{Bool} \qquad \texttt{Unit} \mathbin{\widetilde{\curlyvee}} \texttt{Unit} = \texttt{Unit}$$

$$\texttt{Ref}\ T \mathbin{\widetilde{\curlyvee}} \texttt{Ref}\ T' = \texttt{Ref}\ (T \wedge T') \qquad T \xrightarrow{g} S \mathbin{\widetilde{\curlyvee}} T' \xrightarrow{g'} S' = T \mathbin{\widetilde{\curlywedge}} T' \xrightarrow{g \mathbin{\widetilde{\curlywedge}} g'} S \mathbin{\widetilde{\curlyvee}} S' \qquad \texttt{Bool} \mathbin{\widetilde{\curlywedge}} \texttt{Bool} = \texttt{Bool}$$

$$\texttt{Unit} \mathbin{\widetilde{\curlywedge}} \texttt{Unit} = \texttt{Unit} \qquad \texttt{Ref}\ T \mathbin{\widetilde{\curlywedge}} \texttt{Ref}\ T' = \texttt{Ref}\ (T \wedge T') \qquad T \xrightarrow{g} S \mathbin{\widetilde{\curlywedge}} T' \xrightarrow{g'} S' = T \mathbin{\widetilde{\curlyvee}} T' \xrightarrow{g \mathbin{\widetilde{\curlyvee}} g'} S \mathbin{\widetilde{\curlywedge}} S'$$

$\boxed{T \wedge T' = S}$ $\boxed{R \wedge R' = R''}$

$$R_g \wedge R'_{g'} = (R \wedge R')_{g \wedge g'} \qquad \texttt{Bool} \wedge \texttt{Bool} = \texttt{Bool} \qquad \texttt{Unit} \wedge \texttt{Unit} = \texttt{Unit} \qquad \texttt{Ref}\ T \wedge \texttt{Ref}\ T' = \texttt{Ref}\ (T \wedge T')$$

$$T \xrightarrow{g} S \wedge T' \xrightarrow{g'} S' = T \wedge T' \xrightarrow{g \wedge g'} S \wedge S'$$

$\boxed{T \mathbin{\widetilde{\curlyvee}} g = T'}$ $\boxed{v \sqcup p = v'}$

$$R_g \mathbin{\widetilde{\curlyvee}} g' = R_{g \mathbin{\widetilde{\curlyvee}} g'} \qquad r_p \sqcup p' = r_{p \sqcup p'}$$

Fig. 9. Common definitions for calculus

$$\boxed{\text{ML-GS}}$$

$$p ::= \ell \qquad \frac{}{\ell\ \langle\star\rangle \Rightarrow \ell} \qquad \frac{\ell \preccurlyeq \ell'}{\ell\ \langle\ell'\rangle \Rightarrow \ell'} \qquad \frac{\ell \not\preccurlyeq \ell'}{\ell\ \langle\ell'\rangle \Rightarrow \texttt{blame}}$$

$$\boxed{\text{GLIO and } \lambda^{\star}_{\texttt{SEC}}}$$

$$(GLIO)\ p ::= g \qquad (\lambda^{\star}_{\texttt{SEC}})\ p ::= \ell \qquad \frac{}{\ell\ \langle\star\rangle \Rightarrow \ell} \qquad \frac{\ell \preccurlyeq \ell'}{\ell\ \langle\ell'\rangle \Rightarrow \ell} \qquad \frac{\ell \not\preccurlyeq \ell'}{\ell\ \langle\ell'\rangle \Rightarrow \texttt{blame}}$$

$$\boxed{\text{GSL}_{\text{Ref}} \text{ and WHILE}^{\text{G}}}$$

$$p ::= (g, [\ell, \ell']) \qquad \gamma(\star) = [\bot, \top] \qquad \gamma(\ell) = [\ell, \ell] \qquad \frac{refine([\ell, \ell'], \gamma(g')) = ([\ell_1, \ell'_1], [\ell_2, \ell'_2])}{(g, [\ell, \ell'])\ \langle g'\rangle \Rightarrow (g', [\ell_2, \ell'_2])}$$

$$\frac{refine([\ell, \ell'], \gamma(g')) = undef}{(g, [\ell, \ell'])\ \langle g'\rangle \Rightarrow \texttt{blame}} \qquad \frac{\ell_{1l} \preccurlyeq \ell_{1r} \curlywedge \ell_{2r} \qquad \ell_{1l} \curlyvee \ell_{2l} \preccurlyeq \ell_{2r}}{refine([\ell_{1l}, \ell_{1r}], [\ell_{2l}, \ell_{2r}]) = ([\ell_{1l}, \ell_{1r} \curlywedge \ell_{2r}], [\ell_{1l} \curlyvee \ell_{2l}, \ell_{2r}])}$$

$$\boxed{\lambda^{\star}_{\texttt{IFC}}}$$

$$p ::= \ell \mid p; g \qquad \frac{}{p; \ell\ \langle\star\rangle \Rightarrow p; \ell; \star} \qquad \frac{}{p; \ell; \star\ \langle\star\rangle \Rightarrow p; \ell; \star} \qquad \frac{\ell \preccurlyeq \ell'}{p; \ell\ \langle\ell'\rangle \Rightarrow p; \ell; \ell'} \qquad \frac{\ell \preccurlyeq \ell'}{p; \ell; \star\ \langle\ell'\rangle \Rightarrow p; \ell; \ell'}$$

$$\frac{\ell \not\preccurlyeq \ell'}{p; \ell\ \langle\ell'\rangle \Rightarrow \texttt{blame}} \qquad \frac{\ell \not\preccurlyeq \ell'}{p; \ell; \star\ \langle\ell'\rangle \Rightarrow \texttt{blame}}$$

Fig. 10. Simplified formalization of several existing cast calculus for security labels

$$\boxed{\hat{p} \sqsubseteq \hat{p}'} \qquad \boxed{\vdash \hat{p} : g} \qquad \boxed{|\hat{p}| = \ell}$$

$$\frac{}{\ell \sqsubseteq \ell} \qquad \frac{}{\ell \sqsubseteq \mathbf{?}\ \ell} \qquad \frac{\ell \preccurlyeq \ell'}{\ell \sqsubseteq \mathbf{?}\ \ell'} \qquad \frac{\ell \preccurlyeq \ell'}{\mathbf{?}\ \ell \sqsubseteq \mathbf{?}\ \ell'} \qquad \vdash \ell : \ell \qquad \vdash \mathbf{?}\ \ell : \star \qquad |\ell| = \ell \qquad |\mathbf{?}\ \ell| = \ell$$

Fig. 11. Precision, typing and security label of reverse partial labels

$$\boxed{R \sqsubseteq R'}$$

$$\frac{}{\texttt{Unit} \sqsubseteq \texttt{Unit}} \qquad \frac{}{\texttt{Bool} \sqsubseteq \texttt{Bool}} \qquad \frac{T \sqsubseteq T'}{\texttt{Ref}\ T \sqsubseteq \texttt{Ref}\ T'} \qquad \frac{g \sqsubseteq g' \qquad T \sqsubseteq T' \qquad S \sqsubseteq S'}{T \xrightarrow{g} S \sqsubseteq T' \xrightarrow{g'} S'}$$

$$\boxed{T \sqsubseteq T'}$$

$$\frac{g \sqsubseteq g' \qquad R \sqsubseteq R'}{R_g \sqsubseteq R'_{g'}}$$

Fig. 12. Precision relation for types

$\boxed{\vdash C : R \Rightarrow R'}$ $\boxed{\vdash \hat{\mathcal{C}} : T \Rightarrow S}$

$$\frac{}{\vdash \texttt{CUnit} : \texttt{Unit} \Rightarrow \texttt{Unit}} \qquad \frac{}{\vdash \texttt{CBool} : \texttt{Bool} \Rightarrow \texttt{Bool}} \qquad \frac{\vdash \hat{\mathcal{C}}_{in} : S \Rightarrow T \qquad \vdash \hat{\mathcal{C}}_{out} : T \Rightarrow S}{\vdash \texttt{Ref}\ \hat{\mathcal{C}}_{in}\ \hat{\mathcal{C}}_{out} : \texttt{Ref}\ T \Rightarrow \texttt{Ref}\ S}$$

$$\frac{\vdash \hat{\mathcal{C}}_{arg} : T' \Rightarrow T \qquad \vdash c : gc' \Rightarrow gc \qquad \vdash \hat{\mathcal{C}}_{ret} : S \Rightarrow S'}{\vdash \hat{\mathcal{C}}_{arg} \xrightarrow{c} \hat{\mathcal{C}}_{ret} : T \xrightarrow{gc} S \Rightarrow T' \xrightarrow{gc'} S'} \qquad \frac{\vdash C : R \Rightarrow R' \qquad \vdash c : g \Rightarrow g'}{\vdash C_c : R_g \Rightarrow R'_{g'}}$$

$\boxed{C_1 ; C_2 \Rightarrow C_3}$ $\boxed{\hat{\mathcal{C}}_1 ; \hat{\mathcal{C}}_2 \Rightarrow \hat{\mathcal{C}}_3}$

$$\frac{}{\texttt{CUnit};\texttt{CUnit} \Rightarrow \texttt{CUnit}} \qquad \frac{}{\texttt{CBool};\texttt{CBool} \Rightarrow \texttt{CBool}} \qquad \frac{\hat{\mathcal{C}}_{in_2};\hat{\mathcal{C}}_{in_1} \Rightarrow \hat{\mathcal{C}}_{in_3} \qquad \hat{\mathcal{C}}_{out_1};\hat{\mathcal{C}}_{out_2} \Rightarrow \hat{\mathcal{C}}_{out_3}}{\texttt{Ref}\ \hat{\mathcal{C}}_{in_1}\ \hat{\mathcal{C}}_{out_1};\texttt{Ref}\ \hat{\mathcal{C}}_{in_2}\ \hat{\mathcal{C}}_{out_2} \Rightarrow \texttt{Ref}\ \hat{\mathcal{C}}_{in_3}\ \hat{\mathcal{C}}_{out_3}}$$

$$\frac{\hat{\mathcal{C}}_{arg_2};\hat{\mathcal{C}}_{arg_1} \Rightarrow \hat{\mathcal{C}}_{arg_3} \qquad \hat{\mathcal{C}}_{ret_1};\hat{\mathcal{C}}_{ret_2} \Rightarrow \hat{\mathcal{C}}_{ret_3} \qquad c_2;c_1 \Rightarrow c_3}{\hat{\mathcal{C}}_{arg_1} \xrightarrow{c_1} \hat{\mathcal{C}}_{ret_1};\hat{\mathcal{C}}_{arg_1} \xrightarrow{c_2} \hat{\mathcal{C}}_{ret_2} \Rightarrow \hat{\mathcal{C}}_{arg_3} \xrightarrow{c_3} \hat{\mathcal{C}}_{ret_3}} \qquad \frac{C_1;C_2 \Rightarrow C_3 \qquad c_1;c_2 \Rightarrow c_3}{(C_1)_{c_1};(C_2)_{c_2} \Rightarrow (C_3)_{c_3}}$$

$\boxed{r\ \langle C \rangle \longrightarrow r'}$ $\boxed{v\ \langle \hat{\mathcal{C}} \rangle \longrightarrow v'}$ $\boxed{v\ \langle \hat{\mathcal{C}} \rangle \longrightarrow \texttt{blame}\ P}$

RCBool
$$\frac{}{b\ \langle \texttt{CBool} \rangle \longrightarrow b}$$

RCUnit
$$\frac{}{\texttt{unit}\ \langle \texttt{CUnit} \rangle \longrightarrow \texttt{unit}}$$

RCRef
$$\frac{\hat{\mathcal{C}}_{in_2};\hat{\mathcal{C}}_{in_1} \Rightarrow \hat{\mathcal{C}}_{in_3} \qquad \hat{\mathcal{C}}_{out_1};\hat{\mathcal{C}}_{out_2} \Rightarrow \hat{\mathcal{C}}_{out_3}}{o^{\ell,\hat{\mathcal{C}}_{in_1}\hat{\mathcal{C}}_{out_1}}\ \langle \texttt{CRef}\ \hat{\mathcal{C}}_{in_2}\ \hat{\mathcal{C}}_{out_2} \rangle \longrightarrow o^{\ell,\hat{\mathcal{C}}_{in_3}\hat{\mathcal{C}}_{out_3}}}$$

RCLam
$$\frac{\hat{\mathcal{C}}_{arg_2};\hat{\mathcal{C}}_{arg_1} \Rightarrow \hat{\mathcal{C}}_{arg_3} \qquad c_2;c_1 \Rightarrow c_3 \qquad \hat{\mathcal{C}}_{ret_1};\hat{\mathcal{C}}_{ret_2} \Rightarrow \hat{\mathcal{C}}_{ret_3}}{(\lambda^{c_1} x : T.t)^{\hat{\mathcal{C}}_{arg_1}\hat{\mathcal{C}}_{ret_1}}\ \langle \hat{\mathcal{C}}_{arg_2} \xrightarrow{c_2} \hat{\mathcal{C}}_{ret_2} \rangle \longrightarrow (\lambda^{c_3} x : T.t)^{\hat{\mathcal{C}}_{arg_3}\ \hat{\mathcal{C}}_{ret_3}}}$$

RCValue
$$\frac{r\ \langle C \rangle \longrightarrow r' \qquad p\ \langle c \rangle \Rightarrow p'}{r_p\ \langle C_c \rangle \longrightarrow r'_{p'}}$$

RCBlame
$$\frac{p\ \langle c \rangle \Rightarrow \texttt{blame}\ P}{r_p\ \langle C_c \rangle \longrightarrow \texttt{blame}\ P}$$

$\boxed{\langle\!\langle T \precsim^P S \rangle\!\rangle = \hat{\mathcal{C}}}$

$$\begin{array}{rcl}
\langle\!\langle \texttt{Bool}_g \precsim^P \texttt{Bool}_{g'} \rangle\!\rangle & = & \texttt{CBool}_{\langle\!\langle g \precsim^P g' \rangle\!\rangle} \\
\langle\!\langle \texttt{Unit}_g \precsim^P \texttt{Unit}_{g'} \rangle\!\rangle & = & \texttt{CUnit}_{\langle\!\langle g \precsim^P g' \rangle\!\rangle} \\
\langle\!\langle (\texttt{Ref}\ T)_g \precsim^P (\texttt{Ref}\ S)_{g'} \rangle\!\rangle & = & (\texttt{CRef}\ \langle\!\langle S \precsim^P T \rangle\!\rangle\ \langle\!\langle T \precsim^P S \rangle\!\rangle)_{\langle\!\langle g \precsim^P g' \rangle\!\rangle} \\
\langle\!\langle (T \xrightarrow{gc} S)_g \precsim^P (T' \xrightarrow{gc'} S')_{g'} \rangle\!\rangle & = & (\langle\!\langle T' \precsim^P T \rangle\!\rangle \xrightarrow{\langle\!\langle gc' \precsim^P gc \rangle\!\rangle} \langle\!\langle S \precsim^P S' \rangle\!\rangle)_{\langle\!\langle g \precsim^P g' \rangle\!\rangle}
\end{array}$$

$\boxed{\langle\!\langle g \precsim^P g' \rangle\!\rangle = c}$

$$\begin{array}{rcl}
\langle\!\langle \star \precsim^P \star \rangle\!\rangle & = & \langle \star \rangle \\
\langle\!\langle \ell \precsim^P \star \rangle\!\rangle & = & \langle \ell^P \triangleright \ell\ \mathbf{?} \rangle \\
\langle\!\langle \star \precsim^P \ell \rangle\!\rangle & = & \langle \mathbf{?}\ \ell^P \triangleright \ell \rangle \\
\langle\!\langle \ell \precsim^P \ell' \rangle\!\rangle & = & \langle \ell^P \triangleright \ell' \rangle
\end{array}$$

Fig. 13. Definitions related to type coercion

$$\boxed{t \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EBinaryL}\quad \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{t \oplus s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EBinaryR}\quad \frac{t \mid \mu \mid pc \Downarrow (b_1)_{p_1} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow_\epsilon err}{t \oplus s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EAppL}\quad \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{t\ s \mid \mu \mid pc \longrightarrow err \mid \mu_1}$$

$$\textsc{EAppR}\quad \frac{t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow_\epsilon err}{t\ s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EAppBlameArg}\quad \frac{t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad v\ \langle \hat{\mathcal{C}}_{arg} \rangle \longrightarrow \texttt{blame}\ P}{t\ s \mid \mu \mid pc \Downarrow_\epsilon \texttt{blame}\ P}$$

$$\textsc{EAppBlamePC}\quad \frac{t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad v\ \langle \hat{\mathcal{C}}_{arg} \rangle \longrightarrow v_1 \qquad (pc \sqcup p)\ \langle c \rangle \Rightarrow \texttt{blame}\ P}{t\ s \mid \mu \mid pc \Downarrow_\epsilon \texttt{blame}\ P}$$

$$\textsc{EAppEvalLam}\quad \frac{\begin{array}{c} t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \\ s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad v\ \langle \hat{\mathcal{C}}_{arg} \rangle \longrightarrow v_1 \qquad (pc \sqcup p)\ \langle c \rangle \Rightarrow pc_1 \qquad t_1[x := v_1] \mid \mu_2 \mid pc_1 \Downarrow_\epsilon err \end{array}}{t\ s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EAppBlameRet}\quad \frac{\begin{array}{c} t \mid \mu \mid pc \Downarrow (\lambda^c x : T.t_1)_p^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \\ v\ \langle \hat{\mathcal{C}}_{arg} \rangle \longrightarrow v_1 \qquad (pc \sqcup p)\ \langle c \rangle \Rightarrow pc_1 \qquad t_1[x := v_1] \mid \mu_2 \mid pc_1 \Downarrow v_2 \mid \mu_3 \qquad v_2\ \langle \hat{\mathcal{C}}_{ret} \rangle \longrightarrow \texttt{blame}\ P \end{array}}{t\ s \mid \mu \mid pc \Downarrow_\epsilon \texttt{blame}\ P}$$

$$\textsc{EIf}\quad \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EIfTrue}\quad \frac{t \mid \mu \mid pc \Downarrow \texttt{true}_p \mid \mu_1 \qquad s_1 \mid \mu_1 \mid pc \sqcup p \Downarrow_\epsilon err}{\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EIfFalse}\quad \frac{t \mid \mu \mid pc \Downarrow \texttt{false}_p \mid \mu_1 \qquad s_2 \mid \mu_1 \mid pc \sqcup p \Downarrow_\epsilon err}{\texttt{if}\ t\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \mu \mid pc \Downarrow_\epsilon err}$$

Fig. 14. Blame reduction rules for big-step semantics (1)

$\boxed{t \mid \mu \mid pc \Downarrow_\epsilon err}$

$$\textsc{ERef}\ \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{\mathtt{ref}^{P}_{R_\ell}\ t \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{ERefNSU}\ \frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad |pc| \not\preccurlyeq \ell}{\mathtt{ref}^{P}_{R_\ell}\ t \mid \mu \mid pc \Downarrow_\epsilon \mathtt{nsu\text{-}error}\ P}$$

$$\textsc{EDeref}\ \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{!t \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EDerefBlame}\ \frac{t \mid \mu \mid pc \Downarrow o_p^{\ell,\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad o^\ell \mapsto v \in \mu \qquad v\ \langle \hat{\mathcal{C}}_{out} \rangle \longrightarrow \mathtt{blame}\ P}{!t \mid \mu \mid pc \Downarrow_\epsilon \mathtt{blame}\ P}$$

$$\textsc{EAssignL}\ \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{t :=^{P} s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EAssignR}\ \frac{t \mid \mu \mid pc \Downarrow o_p^{\ell,\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow_\epsilon err}{t :=^{P} s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{EAssignBlame}\ \frac{t \mid \mu \mid pc \Downarrow o_p^{\ell,\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad o \mapsto v' \in \mu_2 \qquad v\ \langle \hat{\mathcal{C}}_{in} \rangle \longrightarrow \mathtt{blame}\ P' \qquad |p| \curlyvee |pc| \preccurlyeq \ell}{t :=^{P} s \mid \mu \mid pc \Downarrow_\epsilon \mathtt{blame}\ P'}$$

$$\textsc{EAssignNSU}\ \frac{t \mid \mu \mid pc \Downarrow o_p^{\ell,\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \mid \mu_1 \qquad s \mid \mu_1 \mid pc \Downarrow v \mid \mu_2 \qquad o \mapsto v' \in \mu_2 \qquad v\ \langle \hat{\mathcal{C}}_{in} \rangle \longrightarrow v_1 \qquad |p| \curlyvee |pc| \not\preccurlyeq \ell}{t :=^{P} s \mid \mu \mid pc \Downarrow_\epsilon \mathtt{nsu\text{-}error}\ P}$$

$$\textsc{ECast}\ \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{t\ \langle \hat{\mathcal{C}} \rangle \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{ECastBlame}\ \frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad v\ \langle \hat{\mathcal{C}} \rangle \longrightarrow \mathtt{blame}\ P}{t\ \langle \hat{\mathcal{C}} \rangle \mid \mu \mid pc \Downarrow_\epsilon \mathtt{blame}\ P}$$

$$\textsc{ELetL}\ \frac{t \mid \mu \mid pc \Downarrow_\epsilon err}{\mathtt{let}\ x = t\ \mathtt{in}\ s \mid \mu \mid pc \Downarrow_\epsilon err}$$

$$\textsc{ELetR}\ \frac{t \mid \mu \mid pc \Downarrow v \mid \mu_1 \qquad s[x := v] \mid \mu_1 \mid pc \Downarrow_\epsilon err}{\mathtt{let}\ x = t\ \mathtt{in}\ s \mid \mu \mid pc \Downarrow_\epsilon err}$$

Fig. 15. Blame reduction rules for big-step semantics (2)

$\boxed{t \sqsubseteq t'}$

$$\frac{}{x \sqsubseteq x} \qquad \frac{}{\mathtt{unit}_\ell \sqsubseteq \mathtt{unit}_\ell} \qquad \frac{}{b_\ell \sqsubseteq b_\ell} \qquad \frac{gc \sqsubseteq gc' \qquad T \sqsubseteq T' \qquad t \sqsubseteq t'}{(\lambda^{gc} x : T.t)^{P}_{\ell} \sqsubseteq (\lambda^{gc'} x : T'.t')^{P'}_{\ell}} \qquad \frac{t \sqsubseteq t' \qquad s \sqsubseteq s'}{t\ s \sqsubseteq t'\ s'} \qquad \frac{t \sqsubseteq t' \qquad s \sqsubseteq s'}{t \oplus s \sqsubseteq t' \oplus s'}$$

$$\frac{t \sqsubseteq t' \qquad s_1 \sqsubseteq s_1' \qquad s_2 \sqsubseteq s_2'}{(\mathtt{if}\ t\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2)^{P} \sqsubseteq (\mathtt{if}\ t'\ \mathtt{then}\ s_1'\ \mathtt{else}\ s_2')^{P'}} \qquad \frac{t \sqsubseteq t'}{\mathtt{ref}^{P}_{\ell}\ t \sqsubseteq \mathtt{ref}^{P'}_{\ell}\ t'} \qquad \frac{t \sqsubseteq t'}{!t \sqsubseteq !t'}$$

$$\frac{t \sqsubseteq t' \qquad s \sqsubseteq s'}{(t := s)^{P} \sqsubseteq (t' := s')^{P'}} \qquad \frac{t \sqsubseteq t' \qquad T \sqsubseteq T'}{(t :: T)^{P} \sqsubseteq (t' :: T')^{P'}} \qquad \frac{t \sqsubseteq t' \qquad s \sqsubseteq s'}{\mathtt{let}\ x = t\ \mathtt{in}\ s \sqsubseteq \mathtt{let}\ x = t'\ \mathtt{in}\ s'}$$

Fig. 16. Precision relation for surface terms

$$\boxed{\Gamma; gc \vdash t : T \;\rightsquigarrow s}$$

CVar
$$\frac{x : T \in \Gamma}{\Gamma; gc \vdash x : T \;\rightsquigarrow x}$$

CBool
$$\frac{}{\Gamma; gc \vdash b_\ell : \texttt{Bool}_\ell \;\rightsquigarrow b_\ell}$$

CUnit
$$\frac{}{\Gamma; gc \vdash \texttt{unit}_\ell : \texttt{Unit}_\ell \;\rightsquigarrow \texttt{unit}_\ell}$$

CLam
$$\frac{\Gamma, x : T; gc' \vdash s : S \;\rightsquigarrow s'}{\Gamma; gc \vdash (\lambda^{gc'} x : T.s)^{P}_\ell : (T \xrightarrow{gc'} S)_\ell \;\rightsquigarrow (\lambda^{\langle\!\langle gc' \precsim^{P} gc' \rangle\!\rangle} x : T.s')_\ell^{\langle\!\langle T \precsim^{P} T \rangle\!\rangle \langle\!\langle S \precsim^{P} S \rangle\!\rangle}}$$

CBinary
$$\frac{\Gamma; gc \vdash t : \texttt{Bool}_{g_1} \;\rightsquigarrow t' \qquad \Gamma; gc \vdash s : \texttt{Bool}_{g_2} \;\rightsquigarrow s'}{\Gamma; gc \vdash t \oplus s : \texttt{Bool}_{g_1 \,\widetilde{\curlyvee}\, g_2} \;\rightsquigarrow t' \oplus s'}$$

CApp
$$\frac{\Gamma; gc \vdash t : (S' \xrightarrow{gc'} T)_g \;\rightsquigarrow t' \qquad \Gamma; gc \vdash s : S \;\rightsquigarrow s' \qquad S \precsim S' \qquad g \precsim gc' \qquad gc \precsim gc'}{\Gamma; gc \vdash (t\ s)^{P} : T \,\widetilde{\curlyvee}\, g \;\rightsquigarrow (t'\ \langle\!\langle (S' \xrightarrow{gc'} T)_g \precsim^{P} (S' \xrightarrow{gc \,\widetilde{\curlyvee}\, g} T)_g \rangle\!\rangle)\ (s'\ \langle\!\langle S \precsim^{P} S' \rangle\!\rangle)}$$

CIf
$$\frac{\Gamma; gc \vdash t : \texttt{Bool}_g \;\rightsquigarrow t' \qquad \Gamma; gc \,\widetilde{\curlyvee}\, g \vdash s_1 : S_1 \;\rightsquigarrow s_1' \qquad \Gamma; gc \,\widetilde{\curlyvee}\, g \vdash s_2 : S_2 \;\rightsquigarrow s_2' \qquad S = S_1 \,\widetilde{\curlyvee}\, S_2}{\Gamma; gc \vdash (\texttt{if } t \texttt{ then } s_1 \texttt{ else } s_2)^{P} : S \,\widetilde{\curlyvee}\, g \;\rightsquigarrow \texttt{if } t' \texttt{ then } s_1'\ \langle\!\langle S_1 \precsim^{P} S \rangle\!\rangle \texttt{ else } s_2'\ \langle\!\langle S_2 \precsim^{P} S \rangle\!\rangle}$$

CAssign
$$\frac{\Gamma; gc \vdash t : (\texttt{Ref } R_{g'})_g \;\rightsquigarrow t' \quad \Gamma; gc \vdash s : S \;\rightsquigarrow s' \qquad S \precsim R_{g'} \qquad g \precsim g' \qquad gc \precsim g'}{\Gamma; \Sigma; gc \vdash (t := s)^{P} : \texttt{Unit}_\perp \;\rightsquigarrow t' :=^{P} s' \langle\!\langle S \precsim^{P} R_{g'} \rangle\!\rangle}$$

CRef
$$\frac{\Gamma; gc \vdash t : R_g \;\rightsquigarrow t' \qquad gc \precsim \ell \qquad g \precsim \ell}{\Gamma; gc \vdash (\texttt{ref}_\ell\ t)^{P} : (\texttt{Ref } R_g)_\perp \;\rightsquigarrow \texttt{ref}^{P}_{R_\ell}(t'\ \langle\!\langle R_g \precsim^{P} R_\ell \rangle\!\rangle)}$$

CDeref
$$\frac{\Gamma; gc \vdash t : (\texttt{Ref } T)_g \;\rightsquigarrow t'}{\Gamma; gc \vdash !t : T \,\widetilde{\curlyvee}\, g \;\rightsquigarrow !t'}$$

CAnno
$$\frac{\Gamma; gc \vdash s : S \;\rightsquigarrow s' \qquad S \precsim T}{\Gamma; gc \vdash (s :: T)^{P} : T \;\rightsquigarrow s'\ \langle\!\langle S \precsim^{P} T \rangle\!\rangle}$$

CLet
$$\frac{\Gamma; gc \vdash t : T \;\rightsquigarrow t' \qquad \Gamma, x : T; gc \vdash s : S \;\rightsquigarrow s'}{\Gamma; gc \vdash \texttt{let } x = t \texttt{ in } s : S \;\rightsquigarrow \texttt{let } x = t' \texttt{ in } s'}$$

Fig. 17. Rules for typing and compilation of surface language

$\boxed{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell} \mu_2 : T}$

$$\textsc{Empty}\quad \frac{}{\emptyset; \emptyset; \emptyset \vdash \emptyset \approx^{\ell'} \emptyset}$$

$$\textsc{New-EQ}\quad \frac{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \qquad \ell \preccurlyeq \ell' \qquad A; \emptyset; \Sigma_1; \Sigma_2 \vdash (r_1)_\ell \approx^{\ell'} (r_2)_\ell : R_\ell \qquad o_1 = \text{fresh}(\mu_1) \qquad o_2 = \text{fresh}(\mu_2)}{(o_1, o_2) ::^E A; \Sigma_1[o_1 : R_\ell]; \Sigma_2[o_2 : R_\ell] \vdash \mu_1[o_1 \mapsto (r_1)_\ell] \approx^{\ell'} \mu_2[o_2 \mapsto (r_2)_\ell]}$$

$$\textsc{New-L}\quad \frac{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \qquad \ell \not\preccurlyeq \ell' \qquad \emptyset; \Sigma_1; \bot \vdash (r_1)_\ell : R_\ell \qquad o_1 = \text{fresh}(\mu_1)}{o_1 ::^L A; \Sigma_1[o_1 : R_\ell]; \Sigma_2 \vdash \mu_1[o_1 \mapsto (r_1)_\ell] \approx^{\ell'} \mu_2}$$

$$\textsc{New-R}\quad \frac{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \qquad \ell \not\preccurlyeq \ell' \qquad \emptyset; \Sigma_2; \bot \vdash (r_2)_\ell : R_\ell \qquad o_2 = \text{fresh}(\mu_2)}{o_2 ::^R A; \Sigma_1; \Sigma_2[o_2 : R_\ell] \vdash \mu_1 \approx^{\ell'} \mu_2[o_2 \mapsto (r_2)_\ell]}$$

$$\textsc{Update-EQ}\quad \frac{\begin{array}{c} A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \\ (o_1, o_2) \in^E A \qquad o_1 : T \in \Sigma_1 \qquad o_2 : T \in \Sigma_2 \qquad o_1 \mapsto v_1 \in \mu_1 \qquad o_2 \mapsto v_2 \in \mu_2 \qquad A; \emptyset; \Sigma_1; \Sigma_2 \vdash v_1' \approx^{\ell'} v_2' : T \end{array}}{A; \Sigma_1; \Sigma_2 \vdash \mu_1[o_1 \mapsto v_1'] \approx^{\ell'} \mu_2[o_2 \mapsto v_2']}$$

$$\textsc{Update-L}\quad \frac{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \qquad o_1 \in^L A \qquad o_1 : T \in \Sigma_1 \qquad o_1 \mapsto v_1 \in \mu_1 \qquad \emptyset; \Sigma_1; \bot \vdash v_1' : T}{A; \Sigma_1; \Sigma_2 \vdash \mu_1[o_1 \mapsto v_1'] \approx^{\ell'} \mu_2}$$

$$\textsc{Update-R}\quad \frac{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2 \qquad o_2 \in^L A \qquad o_2 : T \in \Sigma_2 \qquad o_2 \mapsto v_2 \in \mu_2 \qquad \emptyset; \Sigma_2; \bot \vdash v_2' : T}{A; \Sigma_1; \Sigma_2 \vdash \mu_1 \approx^{\ell'} \mu_2[o_2 \mapsto v_2']}$$

Fig. 18. Observed Equivalence for heap

$$\boxed{p_1 \approx^{\ell} p_2 : g}$$

$$\textsc{$\approx$Low}\quad \frac{|p| \preccurlyeq \ell \qquad \vdash p : g}{p \approx^{\ell} p : g} \qquad\qquad \textsc{$\approx$High}\quad \frac{|p_1| \not\preccurlyeq \ell \qquad |p_2| \not\preccurlyeq \ell \qquad \vdash p_1 : g \qquad \vdash p_2 : g}{p_1 \approx^{\ell} p_2 : g}$$

$$\boxed{A; \Gamma; \Sigma_1; \Sigma_2 \vdash r_1 \approx^{\ell} r_2 : R} \quad \boxed{A; \Gamma; \Sigma_1; \Sigma_2 \vdash v_1 \approx^{\ell} v_2 : T}$$

$$\textsc{$\approx$Unit}\quad \frac{}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash \texttt{unit} \approx^{\ell'} \texttt{unit} : \texttt{Unit}} \qquad\qquad \textsc{$\approx$Bool}\quad \frac{}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash b \approx^{\ell'} b : \texttt{Bool}}$$

$$\textsc{$\approx$Ptr-Low}\quad \frac{(o_1, o_2) \in^{E} A \qquad o_1 : R_\ell \in \Sigma_1 \qquad o_2 : R_\ell \in \Sigma_2 \qquad \vdash \hat{\mathcal{C}}_{in} : T \Rightarrow R_\ell \qquad \vdash \hat{\mathcal{C}}_{out} : R_\ell \Rightarrow T}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash o_1^{\ell, \hat{\mathcal{C}}_{in} \hat{\mathcal{C}}_{out}} \approx^{\ell'} o_2^{\ell, \hat{\mathcal{C}}_{in} \hat{\mathcal{C}}_{out}} : \texttt{Ref}\ T}$$

$$\textsc{$\approx$Ptr-High}\quad \frac{o_1 \in^{L} A \qquad o_2 \in^{R} A \qquad o_1 : R_\ell \in \Sigma_1 \qquad o_2 : R_\ell \in \Sigma_2 \qquad \vdash \hat{\mathcal{C}}_{in} : T \Rightarrow R_\ell \qquad \vdash \hat{\mathcal{C}}_{out} : R_\ell \Rightarrow T}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash o_1^{\ell, \hat{\mathcal{C}}_{in} \hat{\mathcal{C}}_{out}} \approx^{\ell'} o_2^{\ell, \hat{\mathcal{C}}_{in} \hat{\mathcal{C}}_{out}} : \texttt{Ref}\ T}$$

$$\textsc{$\approx$Lam}\quad \frac{A; \Gamma, x : S; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : T \qquad \vdash \hat{\mathcal{C}}_{arg} : S' \Rightarrow S \qquad \vdash c : gc' \Rightarrow gc \qquad \vdash \hat{\mathcal{C}}_{ret} : T \Rightarrow T'}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash (\lambda^c x : S.t_1)^{\hat{\mathcal{C}}_{arg} \hat{\mathcal{C}}_{ret}} \approx^{\ell'} (\lambda^c x : S.t_2)^{\hat{\mathcal{C}}_{arg} \hat{\mathcal{C}}_{ret}} : S' \xrightarrow{gc'} T'}$$

$$\textsc{$\approx$Val-Low}\quad \frac{|p| \preccurlyeq \ell' \qquad A; \Gamma; \Sigma_1; \Sigma_2 \vdash r_1 \approx^{\ell'} r_2 : R \qquad \vdash p : g}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash (r_1)_p \approx^{\ell'} (r_2)_p : R_g}$$

$$\textsc{$\approx$Val-High}\quad \frac{|p_1| \not\preccurlyeq \ell' \qquad |p_2| \not\preccurlyeq \ell' \qquad \Gamma; \Sigma_1; \bot \vdash (r_1)_{p_1} : R_g \qquad \Gamma; \Sigma_2; \bot \vdash (r_2)_{p_2} : R_g}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash (r_1)_{p_1} \approx^{\ell'} (r_2)_{p_2} : R_g}$$

Fig. 19. Observed Equivalence for cast calculus (1)

$$\boxed{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell} t_2 : T}$$

$$\approx\text{VAR}\quad \frac{x : T \in \Gamma}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash x \approx^{\ell'} x : T}$$

$$\approx\text{VAL}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2 \vdash v_1 \approx^{\ell'} v_2 : T}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash v_1 \approx^{\ell'} v_2 : T}$$

$$\approx\text{APP}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : (S \xrightarrow{gc \,\widetilde{\curlyvee}\, g} T)_g \qquad A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash s_1 \approx^{\ell'} s_2 : S}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1\ s_1 \approx^{\ell'} t_2\ s_2 : T \sqcup g}$$

$$\approx\text{BINARY}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : \texttt{Bool}_g \qquad A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash s_1 \approx^{\ell'} s_2 : \texttt{Bool}_{g'}}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \oplus s_1 \approx^{\ell'} t_2 \oplus s_2 : \texttt{Bool}_{g \,\widetilde{\curlyvee}\, g'}}$$

$$\approx\text{IF}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : \texttt{Bool}_g \qquad A; \Gamma; \Sigma_1; \Sigma_2; gc \,\widetilde{\curlyvee}\, g \vdash s_1 \approx^{\ell'} s_2 : S \qquad A; \Gamma; \Sigma_1; \Sigma_2; gc \,\widetilde{\curlyvee}\, g \vdash s_1' \approx^{\ell'} s_2' : S}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash \texttt{if } t_1 \texttt{ then } s_1 \texttt{ else } s_1' \approx^{\ell'} \texttt{if } t_2 \texttt{ then } s_2 \texttt{ else } s_2' : S \,\widetilde{\curlyvee}\, g}$$

$$\approx\text{REF}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : R_\ell \qquad gc \precsim \ell}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash \texttt{ref}^{P}_{R_\ell}\ t_1 \approx^{\ell'} \texttt{ref}^{P}_{R_\ell}\ t_2 : \texttt{Ref}\ R_\ell}$$

$$\approx\text{DEREF}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : (\texttt{Ref}T)_g \qquad gc \precsim \ell}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash !t_1 \approx^{\ell'} !t_2 : T \,\widetilde{\curlyvee}\, g}$$

$$\approx\text{ASSIGN}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell''} t_2 : (\texttt{Ref}\ R_{g'})_g \qquad A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash s_1 \approx^{\ell''} s_2 : R_{g'} \qquad gc \precsim g' \qquad g \precsim g'}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 :=^{P} s_1 \approx^{\ell''} t_2 :=^{P} s_2 : \texttt{Unit}_\perp}$$

$$\approx\text{CAST}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : T \qquad \vdash \hat{\mathcal{C}} : T \Rightarrow S}{A; \Gamma; \Sigma_1; \Sigma_2 \vdash t_1\langle\hat{\mathcal{C}}\rangle \approx^{\ell'} t_2\langle\hat{\mathcal{C}}\rangle : S}$$

$$\approx\text{LET}\quad \frac{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash t_1 \approx^{\ell'} t_2 : T \qquad A; \Gamma, x : T; \Sigma_1; \Sigma_2; gc \vdash s_1 \approx^{\ell'} s_2 : S}{A; \Gamma; \Sigma_1; \Sigma_2; gc \vdash \texttt{let } x = t_1 \texttt{ in } s_1 \approx^{\ell'} \texttt{let } x = t_2 \texttt{ in } s_2 : S}$$

Fig. 20. Observed Equivalence for cast calculus (2)

$$\boxed{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : T}$$

$$\frac{\ell \preccurlyeq \ell_o}{\Gamma; gc \vdash \texttt{unit}_\ell \approx^{\ell_o} \texttt{unit}_\ell : \texttt{Unit}_\ell} \qquad \frac{\ell \preccurlyeq \ell_o}{\Gamma; gc \vdash b_\ell \approx^{\ell_o} b_\ell : \texttt{Bool}_\ell}$$

$$\frac{\Gamma, x : S; gc' \vdash t_1 \approx^{\ell_o} t_2 : T \qquad \ell \preccurlyeq \ell_o}{\Gamma; gc \vdash (\lambda^{gc'} x : S.t_1)_\ell \approx^{\ell_o} (\lambda^{gc'} x : S.t_2)_\ell : (S \xrightarrow{gc'} T)_\ell} \qquad \frac{\ell \not\preccurlyeq \ell_o}{\Gamma; gc \vdash \texttt{unit}_\ell \approx^{\ell_o} \texttt{unit}_\ell : \texttt{Unit}_\ell}$$

$$\frac{\ell \not\preccurlyeq \ell_o}{\Gamma; gc \vdash (b_1)_\ell \approx^{\ell_o} (b_2)_\ell : \texttt{Bool}_\ell} \qquad \frac{\Gamma, x : S; gc' \vdash t_1 : T \qquad \Gamma, x : S; gc' \vdash t_2 : T \qquad \ell \not\preccurlyeq \ell_o}{\Gamma; gc \vdash (\lambda^{gc'} x : S.t_1)_\ell \approx^{\ell_o} (\lambda^{gc'} x : S.t_2)_\ell : (S \xrightarrow{gc'} T)_\ell} \qquad \frac{x : T \in \Gamma}{\Gamma; gc \vdash x \approx^{\ell_o} x : T}$$

$$\frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : \texttt{Bool}_{g_1} \qquad \Gamma; gc \vdash s_1 \approx^{\ell_o} s_2 : \texttt{Bool}_{g_2}}{\Gamma; gc \vdash t_1 \oplus s_1 \approx^{\ell_o} t_2 \oplus s_2 : \texttt{Bool}_{g_1 \,\widetilde{\curlyvee}\, g_2}}$$

$$\frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : (S' \xrightarrow{gc'} T)_g \qquad \Gamma; gc \vdash s_1 \approx^{\ell_o} s_2 : S \qquad S \precsim S' \qquad g \precsim gc' \qquad gc \precsim gc'}{\Gamma; gc \vdash (t_1\ s_1)^P \approx^{\ell_o} (t_2\ s_2)^P : T \,\widetilde{\curlyvee}\, g}$$

$$\frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : \texttt{Bool}_g \qquad \Gamma; gc \,\widetilde{\curlyvee}\, g \vdash s_1 \approx^{\ell_o} s_2 : S_1 \qquad \Gamma; gc \,\widetilde{\curlyvee}\, g \vdash s_1' \approx^{\ell_o} s_2' : S_2 \qquad S = S_1 \,\widetilde{\curlyvee}\, S_2}{\Gamma; gc \vdash (\texttt{if } t_1 \texttt{ then } s_1 \texttt{ else } s_1')^P \approx^{\ell_o} (\texttt{if } t_2 \texttt{ then } s_2 \texttt{ else } s_2')^P : S \,\widetilde{\curlyvee}\, g}$$

$$\frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : (\texttt{Ref } R_{g'})_g \qquad \Gamma; gc \vdash s_1 \approx^{\ell_o} s_2 : S \qquad S \precsim R_{g'} \qquad g \precsim g' \qquad gc \precsim g'}{\Gamma; \Sigma; gc \vdash (t_1 := s_1)^P \approx^{\ell_o} (t_2 := s_2)^P : \texttt{Unit}_\bot}$$

$$\frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : R_g \qquad gc \precsim \ell \qquad g \precsim \ell}{\Gamma; gc \vdash (\texttt{ref}_\ell\ t_1)^P \approx^{\ell_o} (\texttt{ref}_\ell\ t_2)^P : (\texttt{Ref } R_g)_\bot} \qquad \frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : (\texttt{Ref } T)_g}{\Gamma; gc \vdash !t_1 \approx^{\ell_o} !t_2 : T \,\widetilde{\curlyvee}\, g}$$

$$\frac{\Gamma; gc \vdash s_1 \approx^{\ell_o} s_2 : S \qquad S \precsim T}{\Gamma; gc \vdash (s_1 :: T)^P \approx^{\ell_o} (s_2 :: T)^P : T} \qquad \frac{\Gamma; gc \vdash t_1 \approx^{\ell_o} t_2 : T \qquad \Gamma, x : T; gc \vdash s_1 \approx^{\ell_o} s_2 : S}{\Gamma; gc \vdash \texttt{let } x = t_1 \texttt{ in } s_1 \approx^{\ell_o} \texttt{let } x = t_2 \texttt{ in } s_2 : S}$$

Fig. 21. Observed Equivalence for surface language

$\boxed{\vdash p \sqsubseteq p' : g \sqsubseteq g'}$

$$\frac{\vdash p : g \qquad \vdash p' : g' \qquad p \sqsubseteq p'}{\vdash p \sqsubseteq p' : g \sqsubseteq g'}$$

$\boxed{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : T \sqsubseteq T'}$

$$\frac{x : T \in \Gamma \qquad x : T' \in \Gamma'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash x \sqsubseteq x : T \sqsubseteq T'} \qquad \frac{\Gamma; \Gamma'; \Sigma; \Sigma' \vdash v \sqsubseteq v' : T \sqsubseteq T'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash v \sqsubseteq v' : T \sqsubseteq T'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : \mathtt{Bool}_{g_1} \sqsubseteq \mathtt{Bool}_{g_1'} \qquad \Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash s \sqsubseteq s' : \mathtt{Bool}_{g_2} \sqsubseteq \mathtt{Bool}_{g_2'}}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \oplus s \sqsubseteq t' \oplus s' : \mathtt{Bool}_{g_1 \,\widetilde{\curlyvee}\, g_2} \sqsubseteq \mathtt{Bool}_{g_1' \,\widetilde{\curlyvee}\, g_2'}}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : (S \xrightarrow{gc \,\widetilde{\curlyvee}\, g} T)_g \sqsubseteq (S' \xrightarrow{gc' \,\widetilde{\curlyvee}\, g'} T')_{g'} \qquad \Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash s \sqsubseteq s' : S \sqsubseteq S'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t\ s \sqsubseteq t'\ s' : T \,\widetilde{\curlyvee}\, g \sqsubseteq T' \,\widetilde{\curlyvee}\, g'}$$

$$\frac{\begin{array}{c}\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : \mathtt{Bool}_g \sqsubseteq \mathtt{Bool}_{g'} \\ \Gamma; \Gamma'; \Sigma; \Sigma'; gc \,\widetilde{\curlyvee}\, g; gc' \,\widetilde{\curlyvee}\, g' \vdash s_1 \sqsubseteq s_1' : S \sqsubseteq S' \qquad \Gamma; \Gamma'; \Sigma; \Sigma'; gc \,\widetilde{\curlyvee}\, g; gc' \,\widetilde{\curlyvee}\, g' \vdash s_2 \sqsubseteq s_2' : S \sqsubseteq S'\end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash \mathtt{if}\ t\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \sqsubseteq \mathtt{if}\ t'\ \mathtt{then}\ s_1'\ \mathtt{else}\ s_2' : S \,\widetilde{\curlyvee}\, g \sqsubseteq S' \,\widetilde{\curlyvee}\, g'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : (\mathtt{Ref}\ S)_g \sqsubseteq (\mathtt{Ref}\ S')_{g'} \qquad \Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash s \sqsubseteq s' : S \sqsubseteq S'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t :=^{P} s \sqsubseteq t' :=^{P'} s' : \mathtt{Unit}_\perp \sqsubseteq \mathtt{Unit}_\perp}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : R_\ell \sqsubseteq R_\ell' \qquad gc \precsim \ell \qquad gc' \precsim \ell}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash \mathtt{ref}^{P}_{R_\ell}\ t \sqsubseteq \mathtt{ref}^{P'}_{R_\ell'}\ t' : (\mathtt{Ref}\ R_\ell)_\perp \sqsubseteq (\mathtt{Ref}\ R_\ell')_\perp}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : (\mathtt{Ref}\ T)_g \sqsubseteq (\mathtt{Ref}\ T')_{g'}}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash !t \sqsubseteq !t' : T \,\widetilde{\curlyvee}\, g \sqsubseteq T' \,\widetilde{\curlyvee}\, g'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash s \sqsubseteq s' : S \sqsubseteq S' \qquad \vdash \hat{\mathcal{C}} \sqsubseteq \hat{\mathcal{C}}' : S \Rightarrow T \sqsubseteq S' \Rightarrow T'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash s\ \langle \hat{\mathcal{C}} \rangle \sqsubseteq s'\ \langle \hat{\mathcal{C}}' \rangle : T \sqsubseteq T'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g' \vdash t \sqsubseteq t' : T \sqsubseteq T' \qquad \vdash c \sqsubseteq c' : gc \Rightarrow g \sqsubseteq gc' \Rightarrow g'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash \langle c \rangle\ t \sqsubseteq \langle c' \rangle\ t' : T \sqsubseteq T'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash t \sqsubseteq t' : T \sqsubseteq T' \qquad \Gamma, x : T; \Gamma', x : T'; \Sigma; \Sigma'; gc; gc' \vdash s \sqsubseteq s' : S \sqsubseteq S'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash \mathtt{let}\ x = t\ \mathtt{in}\ s \sqsubseteq \mathtt{let}\ x = t'\ \mathtt{in}\ s' : S \sqsubseteq S'}$$

$$\frac{\Gamma; \Gamma'; \Sigma; \Sigma'; gc \,\widetilde{\curlyvee}\, g; gc' \,\widetilde{\curlyvee}\, g' \vdash t \sqsubseteq t' : T \sqsubseteq T' \qquad \vdash p \sqsubseteq p' : g \sqsubseteq g'}{\Gamma; \Gamma'; \Sigma; \Sigma'; gc; gc' \vdash \mathtt{prot}_p(t) \sqsubseteq \mathtt{prot}_{p'}(t') : T \,\widetilde{\curlyvee}\, g \sqsubseteq T' \,\widetilde{\curlyvee}\, g'}$$

Fig. 22. Precision relation for terms of cast calculus

$\boxed{\Gamma;\Gamma';\Sigma;\Sigma' \vdash r \sqsubseteq r' : R \sqsubseteq R'}$ $\boxed{\Gamma;\Gamma';\Sigma;\Sigma' \vdash v \sqsubseteq v' : T \sqsubseteq T'}$

$$\frac{}{\Gamma;\Gamma';\Sigma;\Sigma' \vdash \texttt{unit} \sqsubseteq \texttt{unit} : \texttt{Unit} \sqsubseteq \texttt{Unit}} \qquad \frac{}{\Gamma;\Gamma';\Sigma;\Sigma' \vdash b \sqsubseteq b : \texttt{Bool} \sqsubseteq \texttt{Bool}}$$

$$\frac{\begin{array}{c} o^\ell : R \in \Sigma \qquad o^\ell : R' \in \Sigma' \\ \vdash \hat{\mathcal{C}}_{in} \sqsubseteq \hat{\mathcal{C}}'_{in} : T \Rightarrow R_\ell \sqsubseteq T' \Rightarrow R'_\ell \qquad \vdash \hat{\mathcal{C}}_{out} \sqsubseteq \hat{\mathcal{C}}'_{out} : R_\ell \Rightarrow T \sqsubseteq R'_\ell \Rightarrow T' \end{array}}{\Gamma;\Gamma';\Sigma;\Sigma' \vdash o^{\ell;\hat{\mathcal{C}}_{in}\hat{\mathcal{C}}_{out}} \sqsubseteq o^{\ell;\hat{\mathcal{C}}'_{in}\hat{\mathcal{C}}'_{out}} : \texttt{Ref}\ T \sqsubseteq \texttt{Ref}\ T'}$$

$$\frac{\begin{array}{c} \vdash c \sqsubseteq c' : g_2 \Rightarrow g_1 \sqsubseteq g'_2 \Rightarrow g'_1 \qquad \Gamma, x : S_1; \Gamma', x : S'_1; \Sigma; \Sigma'; g_1; g'_1 \vdash t \sqsubseteq t' : T_1 \sqsubseteq T'_1 \\ \vdash \hat{\mathcal{C}}_{arg} \sqsubseteq \hat{\mathcal{C}}'_{arg} : S_2 \Rightarrow S_1 \sqsubseteq S'_2 \Rightarrow S'_1 \qquad \vdash \hat{\mathcal{C}}_{ret} \sqsubseteq \hat{\mathcal{C}}'_{ret} : T_1 \Rightarrow T_2 \sqsubseteq T'_1 \Rightarrow T'_2 \end{array}}{\Gamma;\Gamma';\Sigma;\Sigma' \vdash (\lambda^c x : S_1.t)^{\hat{\mathcal{C}}_{arg}\hat{\mathcal{C}}_{ret}} \sqsubseteq (\lambda^{c'} x : S'_1.t')^{\hat{\mathcal{C}}'_{arg}\hat{\mathcal{C}}'_{ret}} : S_2 \xrightarrow{g_2} T_2 \sqsubseteq S'_2 \xrightarrow{g'_2} T'_2}$$

$$\frac{\Gamma;\Gamma';\Sigma;\Sigma' \vdash r \sqsubseteq r' : R \sqsubseteq R' \qquad \vdash p \sqsubseteq p' : g \sqsubseteq g'}{\Gamma;\Gamma';\Sigma;\Sigma' \vdash r_p \sqsubseteq r'_{p'} : R_g \sqsubseteq R'_{g'}}$$

$\boxed{\vdash C \sqsubseteq C' : R_1 \Rightarrow R_2 \sqsubseteq R'_1 \Rightarrow R'_2}$ $\boxed{\vdash \hat{\mathcal{C}} \sqsubseteq \hat{\mathcal{C}}' : T \Rightarrow S \sqsubseteq T' \Rightarrow S'}$

$$\frac{}{\vdash \texttt{CUnit} \sqsubseteq \texttt{CUnit} : \texttt{Unit} \Rightarrow \texttt{Unit} \sqsubseteq \texttt{Unit} \Rightarrow \texttt{Unit}} \qquad \frac{}{\vdash \texttt{CBool} \sqsubseteq \texttt{CBool} : \texttt{Bool} \Rightarrow \texttt{Bool} \sqsubseteq \texttt{Bool} \Rightarrow \texttt{Bool}}$$

$$\frac{\vdash \hat{\mathcal{C}}_{in} \sqsubseteq \hat{\mathcal{C}}'_{in} : T \Rightarrow S \sqsubseteq T' \Rightarrow S' \qquad \vdash \hat{\mathcal{C}}_{out} \sqsubseteq \hat{\mathcal{C}}'_{out} : S \Rightarrow T \sqsubseteq S' \Rightarrow T'}{\vdash \texttt{CRef}\ \hat{\mathcal{C}}_{in}\ \hat{\mathcal{C}}_{out} \sqsubseteq \texttt{CRef}\ \hat{\mathcal{C}}'_{in}\ \hat{\mathcal{C}}'_{out} : \texttt{Ref}\ S \Rightarrow \texttt{Ref}\ T \sqsubseteq \texttt{Ref}\ S' \Rightarrow \texttt{Ref}\ T'}$$

$$\frac{\vdash c \sqsubseteq c' : g_2 \Rightarrow g_1 \sqsubseteq g'_2 \Rightarrow g'_1 \qquad \vdash \hat{\mathcal{C}}_{arg} \sqsubseteq \hat{\mathcal{C}}'_{arg} : T_2 \Rightarrow T_1 \sqsubseteq T'_2 \Rightarrow T'_1 \qquad \vdash \hat{\mathcal{C}}_{ret} \sqsubseteq \hat{\mathcal{C}}'_{ret} : S_1 \Rightarrow S_2 \sqsubseteq S'_1 \Rightarrow S'_2}{\vdash \hat{\mathcal{C}}_{arg} \xrightarrow{c} \hat{\mathcal{C}}_{ret} \sqsubseteq \hat{\mathcal{C}}'_{arg} \xrightarrow{c'} \hat{\mathcal{C}}'_{ret} : T_1 \xrightarrow{g_1} S_1 \Rightarrow T_2 \xrightarrow{g_2} S_2 \sqsubseteq T'_1 \xrightarrow{g'_1} S'_1 \Rightarrow T'_2 \xrightarrow{g'_2} S'_2}$$

$$\frac{\vdash C \sqsubseteq C' : R_1 \Rightarrow R_2 \sqsubseteq R'_1 \Rightarrow R'_2 \qquad \vdash c \sqsubseteq c' : g_1 \Rightarrow g_2 \sqsubseteq g'_1 \Rightarrow g'_2}{\vdash C_c \sqsubseteq C'_{c'} : (R_1)_{g_1} \Rightarrow (R_2)_{g_2} \sqsubseteq (R'_1)_{g'_1} \Rightarrow (R'_2)_{g'_2}}$$

$\boxed{\Sigma;\Sigma' \vdash \mu \sqsubseteq \mu'}$

$$\frac{}{\emptyset;\emptyset \vdash \emptyset \sqsubseteq \emptyset} \qquad \frac{\Sigma;\Sigma' \vdash \mu \sqsubseteq \mu' \qquad \emptyset;\emptyset;\Sigma;\Sigma' \vdash v \sqsubseteq v' : T \sqsubseteq T' \qquad o = \text{fresh}(\mu) \qquad o' = \text{fresh}(\mu')}{\Sigma[o : T];\Sigma'[o' : T'] \vdash \mu[o \mapsto v] \sqsubseteq \mu'[o' \mapsto v']}$$

$$\frac{\Sigma;\Sigma' \vdash \mu \sqsubseteq \mu' \qquad \emptyset;\emptyset;\Sigma;\Sigma' \vdash v_1 \sqsubseteq v'_1 : T \sqsubseteq T' \qquad o : T \in \Sigma \qquad o' : T' \in \Sigma' \qquad o \mapsto v \in \mu \qquad o' \mapsto v' \in \mu'}{\Sigma;\Sigma' \vdash \mu[o \mapsto v_1] \sqsubseteq \mu'[o' \mapsto v'_1]}$$

Fig. 23. Precision relation for values, coercions, and heaps of cast calculus