GENERALIZING PARAMETRICITY USING INFORMATION FLOW

GEOFFREY WASHBURN AND STEPHANIE WEIRICH

École Polytechnique Fédérale de Lausanne
e-mail address: geoffrey.washburn@epfl.ch

University of Pennsylvania
e-mail address: sweirich@cis.upenn.edu

Abstract. Run-time type analysis allows programmers to easily and concisely define operations based upon type structure, such as serialization, iterators, and structural equality. However, when types can be inspected at run time, nothing is secret. A module writer cannot use type abstraction to hide implementation details from clients: clients can determine the structure of these supposedly "abstract" data types. Furthermore, access control mechanisms do not help isolate the implementation of abstract datatypes from their clients. Buggy or malicious authorized modules may leak type information to unauthorized clients, so module implementors cannot reliably tell which parts of a program rely on their type definitions.

Currently, module implementors rely on parametric polymorphism to provide integrity and confidentiality guarantees about their abstract datatypes. However, standard parametricity does not hold for languages with run-time type analysis; this paper shows how to generalize parametricity so that it does. The key is to augment the type system with annotations about information-flow. Implementors can then easily see which parts of a program depend on the chosen implementation by tracking the flow of dynamic type information.

1. Introduction

Type analysis is an important programming idiom. Traditional applications for type analysis include serialization, structural equality, cloning and iteration. Many systems use type analysis for more sophisticated purposes such as generating user interfaces, testing code, implementing debuggers and XML support. For this reason, it is important to support type analysis in modern programming languages.

A canonical example of run-time type analysis is the generic structural equality function.

2000 ACM Subject Classification: abstract data types, polymorphism, control structures, type structure, program and recursion schemes, functional constructs, lambda calculus and related systems.

Key words and phrases: parametricity, information-flow, runtime type analysis, ad-hoc polymorphism.

This article is an expanded and corrected version of the abstract that appeared in *The Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)* [WW05].

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

```
fun eq['a] =
  typecase 'a of
    bool    =>
      fn (x:bool, y:bool) =>
        if x then y else false
  | 'b * 'c =>
      fn (x:'b*'c, y:'b*'c) =>
        eq ['b] (fst x, fst y) &&
        eq ['c] (snd x, snd y)
  | ...
```

The `eq` function analyzes its type argument `'a` and returns an equality function for that type. More complex examples of type analysis include generic serialization and type-safe casts [Wei00]. Type-safe casts are especially important in systems with dynamic loading, as they are used to verify that reconstituted values have the expected type [HWC01].

Authors of abstract datatypes can use generic operations to quickly build implementations for their datatypes. For example, because equality for the following `Employee.t` datatype is structural, one may implement it via generic equality.

```
module Employee = struct
  (* name, SSN, address and salary *)
  type t = string * int * string * int
  (* An equality for this type. *)
  fun empEq (x : t) (y : t) =
    Generic.eq [t] (x,y)
end :> sig
  type t
  val empEq : t -> t -> bool
end
```

Although type analysis is very useful, it can also be dangerous. When types are analyzable, software developers cannot be sure that abstraction boundaries will be respected and that code will operate in a compositional fashion. As a consequence, type analysis may destroy properties of *integrity* and *confidentiality* that the author of the `Employee` module expects. Using a type-safe cast, anyone may create a value of type `Employee.t`. Although the type will be correct, other invariants not captured by the type system may be broken. For example, the following malicious code creates an employee with an invalid (negative) salary

```
val (forged : Employee.t) =
 case (Generic.cast
        [string * int * string * int]
        [Employee.t]) of
   SOME f =>
     f ("R U Kidding", 0, "none", -10)
 | NONE   => error "oops!"
```

Furthermore, even if the author of the `Employee` module tries to keep aspects of the employee data type hidden, another module can simply use generic operations to discover them. For

example, if no accessor was provided to the salary component of an `Employee.t`, the following malicious code can extract it

```
val spy (x : Employee.t) : int =
 case (Generic.cast [Employee.t]
        [string * int * string * int])
 of SOME f => let (_, _, _, salary) = f x
              in salary end
  | NONE   => error "oops!"
```

One answer to these problems is to simply prohibit run-time type analysis. However, we believe the benefits of type analysis are too compelling to abandon altogether. Therefore, we propose a basis for a language that permits type analysis, yet allows module writers to define integrity and confidentiality policies for abstract datatypes. In particular, we want authors to know how changing their abstract datatype affects the rest of a program and how their code depends on other abstract types they use.

In languages without type analysis, these questions are easy to answer. Authors rely on *parametric polymorphism* to provide guarantees. The author knows the rest of the program must treat her abstract datatypes as black boxes that may only be "pushed around," not inspected, modified or created. Dually authors are restricted in the same fashion when using other abstract datatypes. In the presence of type analysis, the programmer cannot know what code may depend on the definition of an abstract datatype. Any part of the program can dynamically discover the underlying type and introduce dependencies on its definition.

In the past it has been suggested that type analysis could be tamed by distinguishing between analyzable and unanalyzable types [HM95]. Unfortunately, just controlling which parts of the program may analyze a type does not allow programmers to answer our questions. Imagine an extension, not unlike "friends" in C++, where an author can specify which modules may analyze a type. In the following code, modules `A` and `B` may analyze the type `A.t`, and modules `B` and `C` may analyze the type `B.u`.

```
module A = struct
 type t = int
 val x = 3
end :> sig
 type t permit A, B
 val x : t
end
```

```
module B = struct
 type u = A.t
 val y = A.x
end :> sig
 type u permit B, C
 val y : u
end
```

```
module C = struct
 val z = case (cast [B.u] [int]) of
           SOME f => "It is an int"
         | NONE   => "It is not an int"
end :> sig
 val z : string
end
```

Module `C` is not parametric with respect to `A.t`, even though module `C` is not allowed to analyze `A.t`: If the implementation of `A.t` changes, so does the value of `C.z`. Despite restricting analysis of `A.t` to `A` and `B`, the implementation of the type has been leaked to a third-party. Furthermore, because the type `B.u` is abstract, the author of `A` cannot know

of the dependency. Access control places undue trust in a client not to provide others with the capabilities and information it has been granted. Consequently, we must look beyond access-control for a method of answering the desired questions.

We propose that tracking the flow of type information through a program with *information-flow labels* allows a programmer to easily determine how their type definitions influence the rest of the program. Information-flow extends a standard type system with elements of a lattice that describes the information content for each computation. For example, we could use a simple lattice containing two points `L` (low-security) and `H` (high-security). A type `bool @ H` then means the expression it describes could use "high-security" information to produce the resulting boolean, while an expression of type `bool @ L` only requires "low-security" information to produce its result. The novelty of our approach compared to previous information-flow type systems is that we also label kinds to track the information content of type constructors.

To regain parametric reasoning about abstract types in the presence of type analysis, we can label types with an information content that can be tracked. Consequently, computations depending on those types must also have that label.

```
module A = struct
 type t = int
 val x = 3
end :> sig
 type t @ H
 val x : t @ L
end
```

```
module B = struct
 type u = A.t
 val y = A.x
end :> sig
 type u @ H
 val y : u @ L
end
```

```
module C = struct
 val z = case (cast [B.u] [int]) of
           SOME f => "It is an int"
         | NONE   => "It is not an int"
end :> sig
 val z : string @ H
end
```

In the revised example, module `A` is sealed with a signature that indicates that the type definition `t` depends upon high-security information and the value `x` only on low-security information. The type `B.u` and value `C.z` must both be labeled as high security because they depend upon the high-security information in `A.t`. The presence of a label `H` alerts the author of `A` to a dependency.

Furthermore, only the module `A` can create values of type `A.t` that are labeled with `L`. Using type analysis to create values of type `A.t` would taint the result with `H`. Therefore, if module `A` requires its inputs be of type `A.t @ L`, then it is impossible to use its functions with forged values. The author now has a guarantee that module invariants will be maintained and the integrity of her abstraction will not be violated.

Information flow avoids the problems of access control because labels are propagated even when no analysis occurs. For example, the identity function can be assigned both the type `A.t @ L -(L)-> A.t @ L` and the type `A.t @ H -(L)-> A.t @ H` witnessing that it propagates the information content of the argument. Here the function type constructor `-(·)->` is itself labeled to indicate the information content of creating the function – creating the identity function does not require any information.

kinds	$\kappa ::= \star^{\ell}$	*types*
	$\mid \kappa_1 \xrightarrow{\ell} \kappa_2$	*operators*
type constructors	$\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	λ-*calculus*
	$\mid \mathsf{bool}$	*booleans*
	$\mid \tau_1 \to \tau_2$	*functions*
	$\mid \tau_1 \times \tau_2$	*products*
	$\mid \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times}$	*analysis*
types	$\sigma ::= (\tau)\ @\ \ell$	*injection*
	$\mid \sigma_1 \xrightarrow{\ell} \sigma_2$	*functions*
	$\mid \sigma_1 \times^{\ell} \sigma_2$	*products*
	$\mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$	*constructor polymorphism*
terms	$e ::= \mathbf{true} \mid \mathbf{false}$	*booleans*
	$\mid x \mid \lambda x{:}\sigma.e \mid e_1e_2$	λ-*calculus*
	$\mid \langle e_1,e_2\rangle \mid \mathbf{fst}\, e \mid \mathbf{snd}\, e$	*tuples*
	$\mid \Lambda\alpha{:}\star^{\ell}.e \mid e[\tau]$	*constructor polymorphism*
	$\mid \mathbf{fix}\ x{:}\sigma.e$	*fix-point*
	$\mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	*conditional*
	$\mid \mathbf{typecase}[\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\to}\ e_{\times}$	*analysis*
values	$v ::= \mathbf{true} \mid \mathbf{false} \mid \lambda x{:}\sigma.e \mid \langle v_1,v_2\rangle \mid \Lambda\alpha{:}\star^{\ell}.e$	
term substitutions	$\gamma ::= \cdot \mid \gamma,[e/x]$	
type substitutions	$\delta ::= \cdot \mid \delta,[\tau/\alpha]$	
term variable contexts	$\Gamma ::= \cdot \mid \Gamma,x{:}\sigma$	
type variable contexts	$\Delta ::= \cdot \mid \Delta,\alpha{:}\kappa$	

Figure 1: The grammar of the $\lambda_{\mathrm{SEC}i}$ language.

In the next section, we describe a core calculus for combining information-flow and run-time type analysis. We then follow with our key contribution: By tracking the flow of type information, it is possible to generalize the standard parametricity theorem to handle languages with run-time type analysis. This generalized theorem can be used in the same manner as parametricity to establish integrity and confidentiality properties.

2. The $\lambda_{\mathrm{SEC}i}$ language

$\lambda_{\mathrm{SEC}i}$ is a core calculus combining information flow and type analysis. The design of $\lambda_{\mathrm{SEC}i}$ is intended to be as simple as possible while still capturing the essential interactions between data abstraction and type-directed programming. It is derived from the type-analyzing language λ_i^{ML} developed by Harper and Morrisett [HM95] and the information-flow security language λ_{SEC} of Zdancewic [Zda02]. We chose to base $\lambda_{\mathrm{SEC}i}$ on λ_i^{ML} because it provides a

simple yet expressive model of run-time type analysis. The language λ_i^{ML} was developed as an intermediate language for efficiently compiling parametric polymorphism. Similarly, λ_{SEC} was developed to study information flow in the context of the simply-typed λ-calculus.

The grammar for $\lambda_{\text{SEC}i}$ appears in Figure 1. It is a predicative, call-by-value polymorphic λ-calculus with booleans, functions and general recursion. Fixed points are separate from functions to make nontermination aspects of proofs modular. We have chosen to make $\lambda_{\text{SEC}i}$ predicative because it is closer in design to λ_i^{ML}, and avoids the complexities introduced by higher-order type analysis. We conjecture that our results extend to languages with impredicative polymorphism. Also for simplicity, we do not allow higher-kinded polymorphism, but conjecture that our results extend to that feature as well.

In $\lambda_{\text{SEC}i}$ type constructors, τ, which can be analyzed at run-time, are separated from types, σ, which describe terms. The language of type constructors consists of the simply-typed λ-calculus, a type operator called Typerec, and three primitive constructors that correspond to types: bool, $\tau_1 \to \tau_2$, and $\tau_1 \times \tau_2$.

2.1. **Run-time type analysis.** The term **typecase** in $\lambda_{\text{SEC}i}$ can be used to define operations that depend on run-time type information. This term takes a constructor to scrutinize, τ, as well as three branches corresponding to the primitive constructors. As in Figure 1, we will frequently use the mnemonic subscripts $\cdot_{\text{bool}}$, $\cdot_{\to}$, and $\cdot_{\times}$to refer to entities that handle branches for booleans, functions types, and product types respectively.

During evaluation the constructor argument must of **typecase** be reduced to determine its head form so that a branch can be chosen. The bracketed argument to **typecase**, $[\gamma.\sigma]$, is only necessary for typechecking, so it can be ignored until we cover type checking. We write $e \leadsto e'$ to mean that term e reduces in a single step to e' and $\tau \leadsto \tau'$ to mean that constructor τ makes a weak-head reduction step to τ'. We write $\leadsto^*$ for the reflexive, transitive closure of the reduction relations. The complete dynamic semantics for $\lambda_{\text{SEC}i}$ terms can be found in Figures 2 and 3.

$\lambda_{\text{SEC}i}$ also includes a constructor, Typerec, for analyzing type information. Without Typerec, it is impossible to assign types to some useful terms that perform type analysis [HM95]. Typerec implements a *paramorphism* (a type of fold) over the structure of the argument constructor. When the head of the argument is one of the three primitive constructors, Typerec will apply the appropriate branch to the constituent types, as well as the recursive invocation of Typerec on them. The complete dynamic semantics of type constructors is given in Figure 4.

2.2. **The information content of constructors.** Information-flow type systems track the flow of information by annotating types with labels that specify the information content of the terms they describe. Because type constructors have computational content in $\lambda_{\text{SEC}i}$ (and influence the evaluation of terms) it is also necessary to label kinds.

Labels, ℓ, are drawn from an unspecified join semi-lattice, with a least element ($\bot$), joins ($\sqcup$) for finite subsets of elements in the lattice, and a partial order ($\sqsubseteq$). The actual lattice used by the type system is determined by the desired confidentiality and integrity policies of the program. However, for our proofs, it is necessary that at a minimum, computing label joins and ordering must be decidable. Intuitively, the higher a label is in the lattice, the more restricted the information content of a constructor or term should be. For our examples we will use a simple two point lattice ($\bot$ for low security, $\top$ for high security) that tracks the dynamic discovery of a single type definition. In practice, any lattice with the specified

$$\frac{}{(\lambda x{:}\sigma.e)v \rightsquigarrow e[v/x]}\ \text{EV:APP} \qquad \frac{}{(\Lambda\alpha{:}\kappa.e)[\tau] \rightsquigarrow e[\tau/\alpha]}\ \text{EV:TAPP} \qquad \frac{}{\mathbf{fst}\,\langle v_1,v_2\rangle \rightsquigarrow v_1}\ \text{EV:FST}$$

$$\frac{}{\mathbf{snd}\,\langle v_1,v_2\rangle \rightsquigarrow v_2}\ \text{EV:SND} \qquad \frac{}{\mathbf{fix}\ x{:}\sigma.e \rightsquigarrow e[\mathbf{fix}\ x{:}\sigma.e/x]}\ \text{EV:FIX}$$

$$\frac{}{\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_1}\ \text{EV:IF1} \qquad \frac{}{\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 \rightsquigarrow e_2}\ \text{EV:IF2}$$

$$\frac{\tau \rightsquigarrow^* \mathsf{bool}}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\mathsf{bool}}}\ \text{EV:TCASE-BOOL}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \rightarrow \tau_2}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\rightarrow}[\tau_1][\tau_2]}\ \text{EV:TCASE-ARR}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \times \tau_2}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\times}[\tau_1][\tau_2]}\ \text{EV:TCASE-PROD}$$

Figure 2: Term reduction rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 e_2 \rightsquigarrow e_1' e_2}\ \text{EV:APP1} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1 e_2 \rightsquigarrow v_1 e_2'}\ \text{EV:APP2} \qquad \frac{e_1 \rightsquigarrow e_1'}{\langle e_1,e_2\rangle \rightsquigarrow \langle e_1',e_2\rangle}\ \text{EV:PAIR1}$$

$$\frac{e_2 \rightsquigarrow e_2'}{\langle v_1,e_2\rangle \rightsquigarrow \langle v_1,e_2'\rangle}\ \text{EV:PAIR2} \qquad \frac{e \rightsquigarrow e'}{\mathbf{fst}\,e \rightsquigarrow \mathbf{fst}\,e'}\ \text{EV:FST-CON} \qquad \frac{e \rightsquigarrow e'}{\mathbf{snd}\,e \rightsquigarrow \mathbf{snd}\,e'}\ \text{EV:SND-CON}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3}\ \text{EV:IF-CON} \qquad \frac{e \rightsquigarrow e'}{e[\tau] \rightsquigarrow e'[\tau]}\ \text{EV:TAPP-CON}$$

Figure 3: Term congruence rules for $\lambda_{\mathrm{SEC}i}$.

structure could be used. An example of a rich lattice structure is the Decentralized Label Model of Myers and Liskov [ML00].

The labels on kinds describe the information content of type constructors. The kind of a constructor (and therefore its information content) is described using the judgment $\Delta \vdash \tau : \kappa$, read as "constructor τ is well-formed having kind κ with respect to the type variable context Δ". Figure 6 shows the definition of this judgment. The operator $\mathcal{L}(\kappa)$, defined in Figure 7, extracts the label of a kind.

The kind system is conservative: If the label of κ is ℓ, then the information content of a constructor of kind κ is *at most* ℓ. The information level of a constructor can be raised via subsumption. Because kinds are labeled, the ordering $\sqsubseteq$ on labels induces a sub-kinding relation, $\kappa_1 \leq \kappa_2$. A kind $\star^{\ell_1}$ is a sub-kind of $\star^{\ell_2}$ if $\ell_1 \sqsubseteq \ell_2$. Sub-kinding for function kinds is

$$\frac{\tau_1 \rightsquigarrow \tau_1'}{\tau_1\tau_2 \rightsquigarrow \tau_1'\tau_2}\ \text{WHR:APP-CON} \qquad \frac{}{(\lambda\alpha{:}\kappa.\tau_1)\tau_2 \rightsquigarrow \tau_1[\tau_2/\alpha]}\ \text{WHR:APP}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \mathsf{Typerec}\ \tau'\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times}}\ \text{WHR:TREC-CON}$$

$$\frac{}{\mathsf{Typerec}\ (\mathsf{bool})\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \tau_{\mathsf{bool}}}\ \text{WHR:TREC-BOOL}$$

$$\frac{}{\mathsf{Typerec}\ (\tau_1 \rightarrow \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \tau_{\rightarrow}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times})\ (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times})}\ \text{WHR:TREC-ARR}$$

$$\frac{}{\mathsf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \tau_{\times}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times})\ (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times})}\ \text{WHR:TREC-PROD}$$

Figure 4: Constructor reduction rules for $\lambda_{\mathrm{SEC}i}$.

standard. The relation is reflexive and transitive by definition; the complete definition of subkinding can be found in Figure 5.

The label of a constructor τ, of kind $\star^{\ell}$, also describes the information gained when the constructor is analyzed. For example, the kind of a Typerec constructor must be labeled at least as high as the scrutinized type constructor τ, as shown in the rule below. This requirement accounts for the fact that the constructor that is equivalent to reducing the Typerec constructor will depend on the structure of τ.

$$\frac{\begin{array}{ll}\Delta \vdash \tau : \star^{\ell} & \Delta \vdash \tau_{\rightarrow} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathsf{bool}} : \kappa & \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa\end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \tau_{\times} : \kappa}\ \text{WFC:TREC}$$

By default the label on the bool constructor is $\bot$, as defined by WFC:BOOL in Figure 6. The label of the kind for function and product constructors must be at least as high as the join of its two constituent constructors. This is because the label must reflect the information content of the entire constructor.

To propagate information flows through type applications, the kinds of type functions, $\kappa_1 \xrightarrow{\ell} \kappa_2$, have a label ℓ that represents the information propagated by invoking the function. The information, ℓ, is propagated into the result of application as $\kappa_2 \sqcup \ell$. This is shorthand for relabeling κ_2 with $\mathcal{L}(\kappa_2) \sqcup \ell$. The definition for lifting label joins to kinds is given in Figure 7.

2.3. **Tracking information flow in terms.**

Definition 2.1 (Type variable context restriction)**.** We will write $\Delta^{\star}$ for those type variable contexts Δ where $\forall\alpha{:}\kappa \in \Delta$, $\kappa = \star^{\ell}$ for some ℓ. ∎

$$\frac{}{\kappa \leq \kappa}\ \textsc{SBK:REFL} \qquad \frac{\kappa_1 \leq \kappa_2 \qquad \kappa_2 \leq \kappa_3}{\kappa_1 \leq \kappa_3}\ \textsc{SBK:TRANS} \qquad \frac{\ell_1 \sqsubseteq \ell_2}{\star^{\ell_1} \leq \star^{\ell_2}}\ \textsc{SBK:TYPE}$$

$$\frac{\kappa_3 \leq \kappa_1 \qquad \kappa_2 \leq \kappa_4 \qquad \ell_1 \sqsubseteq \ell_2}{\kappa_1 \xrightarrow{\ell_1} \kappa_2 \leq \kappa_3 \xrightarrow{\ell_2} \kappa_4}\ \textsc{SBK:ARR}$$

Figure 5: Sub-kinding rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsc{WFC:VAR} \qquad \frac{}{\Delta \vdash \mathsf{bool} : \star^{\bot}}\ \textsc{WFC:BOOL} \qquad \frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{WFC:ARR}$$

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{WFC:PROD} \qquad \frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa_1 \xrightarrow{\bot} \kappa_2}\ \textsc{WFC:ABS}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2 \sqcup \ell}\ \textsc{WFC:APP}$$

$$\frac{\begin{array}{ll}\Delta \vdash \tau : \star^{\ell} & \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathsf{bool}} : \kappa & \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}\ \textsc{WFC:TREC}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \leq \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \textsc{WFC:SUB}$$

Figure 6: Constructor well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

Kind information	$\mathcal{L}(\star^{\ell}) \triangleq \ell$	$\mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) \triangleq \ell$
Kind join	$\star^{\ell_1} \sqcup \ell_2 \triangleq \star^{(\ell_1 \sqcup \ell_2)}$	$(\kappa_1 \xrightarrow{\ell_1} \kappa_2) \sqcup \ell_2 \triangleq \kappa_1 \xrightarrow{\ell_1 \sqcup \ell_2} \kappa_2$
Type information	$\mathcal{L}((\tau)\ @\ \ell) \triangleq \ell$ $\mathcal{L}(\sigma_1 \times^{\ell} \sigma_2) \triangleq \ell$	$\mathcal{L}(\sigma_1 \xrightarrow{\ell} \sigma_2) \triangleq \ell$ $\mathcal{L}(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \triangleq \ell_1$
Type join	$(\tau)\ @\ \ell_1 \sqcup \ell_2 \triangleq (\tau)\ @\ (\ell_1 \sqcup \ell_2)$ $(\sigma_1 \times^{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \times^{(\ell_1 \sqcup \ell_2)} \sigma_2$	$(\sigma_1 \xrightarrow{\ell_1} \sigma_2) \sqcup \ell_2 \triangleq \sigma_1 \xrightarrow{\ell_1 \sqcup \ell_2} \sigma_2$ $(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \sqcup \ell_3 \triangleq \forall^{(\ell_1 \sqcup \ell_3)}\alpha{:}\star^{\ell_2}.\sigma$

Figure 7: Kind and type label operators for $\lambda_{\mathrm{SEC}i}$.

$$\frac{}{\Delta^\star \vdash \cdot}\ \textsc{wftc:empty} \qquad \frac{\Delta^\star \vdash \Gamma \qquad \Delta^\star \vdash \sigma}{\Delta^\star \vdash \Gamma, x{:}\sigma}\ \textsc{wftc:cons}$$

Figure 8: Context well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{true} : (\mathsf{bool})\ @\ \bot}\ \textsc{wft:true} \qquad \frac{\Delta^\star \vdash \Gamma}{\Delta^\star;\Gamma \vdash \mathbf{false} : (\mathsf{bool})\ @\ \bot}\ \textsc{wft:false}$$

$$\frac{\Delta^\star \vdash \Gamma \qquad x : \sigma \in \Gamma}{\Delta^\star;\Gamma \vdash x : \sigma}\ \textsc{wft:var} \qquad \frac{\Delta^\star;\Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \qquad \Delta^\star \vdash \sigma_1}{\Delta^\star;\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \xrightarrow{\bot} \sigma_2}\ \textsc{wft:abs}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \xrightarrow{\ell} \sigma_2 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_1}{\Delta^\star;\Gamma \vdash e_1 e_2 : \sigma_2 \sqcup \ell}\ \textsc{wft:app} \qquad \frac{\Delta^\star, \alpha{:}\star^\ell;\Gamma \vdash e : \sigma}{\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^\ell.e : \forall^\bot \alpha{:}\star^\ell.\sigma}\ \textsc{wft:tabs}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \forall^\ell \alpha{:}\star^{\ell'}.\sigma \qquad \Delta^\star \vdash \tau : \star^{\ell'}}{\Delta^\star;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \sqcup \ell}\ \textsc{wft:tapp} \qquad \frac{\Delta^\star;\Gamma \vdash e_1 : \sigma_1 \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma_2}{\Delta^\star;\Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times^\bot \sigma_2}\ \textsc{wft:pair}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell}\ \textsc{wft:fst} \qquad \frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{snd}\, e : \sigma_2 \sqcup \ell}\ \textsc{wft:snd}$$

$$\frac{\Delta^\star;\Gamma, x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \textsc{wft:fix}$$

$$\frac{\Delta^\star;\Gamma \vdash e_1 : (\mathsf{bool})\ @\ \ell \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma \qquad \Delta^\star;\Gamma \vdash e_3 : \sigma}{\Delta^\star;\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma \sqcup \ell}\ \textsc{wft:if}$$

$$\frac{\begin{array}{ccc}\Delta^\star \vdash \tau : \star^\ell & & \Delta^\star;\Gamma \vdash e_{\mathsf{bool}} : \sigma[\mathsf{bool}/\gamma] \\ \Delta^\star, \gamma{:}\star^\ell \vdash \sigma & \Delta^\star;\Gamma \vdash e_\rightarrow : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] & \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \\ \ell \sqsubseteq \ell' & \Delta^\star;\Gamma \vdash e_\times : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma] & \end{array}}{\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_\rightarrow\ e_\times : \sigma[\tau/\gamma]}\ \textsc{wft:tcase}$$

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \qquad \Delta^\star \vdash \sigma_1 \le \sigma_2}{\Delta^\star;\Gamma \vdash e : \sigma_2}\ \textsc{wft:sub}$$

Figure 9: Term well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

The labels on types describe the information content of terms. We use the judgment $\Delta^\star;\Gamma \vdash e : \sigma$ to mean that "term e is well-formed with type σ with respect to the term context Γ and the type context $\Delta^\star$." Figure 9 shows definition of this judgment. As we did for kinds, we define (in Figure 7) the operator $\mathcal{L}(\sigma)$ to extract the label of a type.

The judgment $\Delta^\star \vdash \sigma$ is used to indicate that "type σ is well-formed with respect to type context $\Delta^\star$." The rules for the well-formedness of types can be found in Figure 11. The types of $\lambda_{\mathrm{SEC}i}$ include the standard ones for functions $\sigma_1 \xrightarrow{\ell} \sigma_2$, products $\sigma_1 \times^\ell \sigma_2$, and quantified types $\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$, plus those that are computed by type constructors $(\tau)\ @\ \ell$. Note that in the well-formedness rule for types formed from type constructors,

$$\frac{\Delta^\star \vdash \tau : \star^{\ell_1}}{\Delta^\star \vdash (\tau)\ @\ \ell_2}\ \textsc{wftp:con}$$

there is no need for a connection between the label ℓ on the kind and the label on the type. That is because ℓ describes the information content of τ, while the label ℓ' on $(\tau)\ @\ \ell'$ describes the information content of a term with type $(\tau)\ @\ \ell'$. It is sound to discard ℓ, because once a constructor has been coerced to a type it can only be used statically to describe terms and cannot be analyzed.

Like constructors, the information content specified by labels for terms is conservative. The lattice ordering induces a subtyping judgment $\Delta^\star \vdash \sigma_1 \le \sigma_2$, and subsumption can be used to raise the information level of a term; the complete definition of subtyping can be found in Figure 12.

Information flow is tracked at the term level analogously to the type level. Term abstractions, of type $\sigma_1 \xrightarrow{\ell} \sigma_2$, like type functions, propagate some information ℓ when applied. Similarly, type abstractions, $\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$, propagate some information ℓ_1 when applied. The label ℓ_2 describes the information content of constructors that can be used to instantiate the type abstraction. For products, $\sigma_1 \times^\ell \sigma_2$, the label ℓ indicates the information propagated when one of its components is projected.[1]

Like $\mathsf{Typerec}$, the label ℓ' on the type of the **typecase** expression must be at least as high in the lattice as the label ℓ on the scrutinee. This is to account for the information learned when **typecase** examines the structure of the scrutinee.

$$\frac{\begin{array}{cc} \Delta^\star \vdash \tau : \star^\ell & \Delta^\star;\Gamma \vdash e_{\mathsf{bool}} : \sigma[\mathsf{bool}/\gamma] \\ \Delta^\star,\gamma{:}\star^\ell \vdash \sigma & \Delta^\star;\Gamma \vdash e_\to : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] \qquad \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \\ \ell \sqsubseteq \ell' & \Delta^\star;\Gamma \vdash e_\times : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma] \end{array}}{\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_\to\ e_\times : \sigma[\tau/\gamma]}\ \textsc{wft:tcase}$$

Unlike some other formulations of type analysis, $\lambda_{\mathrm{SEC}i}$'s **typecase** primitive does not introduce type equalities. For example, while typechecking e_{bool} it will not be the case that $\tau = \mathsf{bool} : \star^\ell$. Instead, $\lambda_{\mathrm{SEC}i}$ relies on the fact that **typecase** allows the type of its branches to depend upon the type it scrutinizes. That is, e_{bool} can produce a value of type $\sigma[\mathsf{bool}/\gamma]$, while $e_\to$ can produce a value of type $\sigma[\alpha \to \beta/\gamma]$, and $e_\times$ can produce a value of the type $\sigma[\alpha \times \beta/\gamma]$.

Because the type of a **typecase** term can depend upon the scrutinized constructor τ, it is not possible to deterministically synthesize its type solely from its subterms, τ, e_{bool}, $e_\to$, and $e_\times$. Therefore an annotation, $[\gamma.\sigma]$, is required for typechecking **typecase**.

[1] In the case of a pure functional language with only extensional equality the labels on functions, type abstractions, and products are technically unnecessary. For functions and type abstractions the information content can always be pushed into their range, and the information content of products can always be pushed into their components. In impure languages, and languages with pointer equality on values, the labels are necessary. The labels are present in $\lambda_{\mathrm{SEC}i}$ to avoid specializing too early.

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau = \tau : \kappa}\ \text{EQC:REFL} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 : \kappa \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa}{\Delta \vdash \tau_1 = \tau_3 : \kappa}\ \text{EQC:TRANS}$$

$$\frac{\Delta \vdash \tau_2 = \tau_1 : \kappa}{\Delta \vdash \tau_1 = \tau_2 : \kappa}\ \text{EQC:SYM} \qquad \frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 = \tau_3 \to \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \text{EQC:ARR}$$

$$\frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_3 \times \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \text{EQC:PROD}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 = \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \xrightarrow{\perp} \kappa_2}\ \text{EQC:ABS-CON} \qquad \frac{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 : \kappa_2}{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2}\ \text{EQC:ABS-BETA}$$

$$\frac{\Delta \vdash \tau_1 = \tau_3 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa_1}{\Delta \vdash \tau_1\tau_2 = \tau_3\tau_4 : \kappa_2 \sqcup \ell}\ \text{EQC:APP}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 = \tau_2 : \star^{\ell} \\ \Delta \vdash \tau_{\mathsf{bool}} = \tau'_{\mathsf{bool}} : \kappa \qquad \Delta \vdash \tau_{\to} = \tau'_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\times} = \tau'_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \qquad \ell \sqsubseteq \ell' \qquad \text{where } \ell' = \mathcal{L}(\kappa)\end{array}}{\Delta \vdash \mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} = \mathsf{Typerec}\ \tau_2\ \tau'_{\mathsf{bool}}\ \tau'_{\to}\ \tau'_{\times} : \kappa}\ \text{EQC:TREC-CON}$$

$$\frac{\Delta \vdash \mathsf{Typerec\ bool}\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathsf{Typerec\ bool}\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\mathsf{bool}} : \kappa}\ \text{EQC:TREC-BOOL}$$

$$\frac{\Delta \vdash \mathsf{Typerec}\ (\tau_1 \to \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathsf{Typerec}(\tau_1 \to \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\to}\tau_1\ \tau_2\ \begin{array}{l}(\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times}) \\ (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times})\end{array} : \kappa}\ \text{EQC:TREC-ARR}$$

$$\frac{\Delta \vdash \mathsf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa}{\Delta \vdash \mathsf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} = \tau_{\times}\tau_1\ \tau_2\ \begin{array}{l}(\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times}) \\ (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times})\end{array} : \kappa}\ \text{EQC:TREC-PROD}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa_1 \qquad \kappa_1 \le \kappa_2}{\Delta \vdash \tau_1 = \tau_2 : \kappa_2}\ \text{EQC:SUB}$$

Figure 10: Constructor equivalence for $\lambda_{\mathrm{SEC}i}$

2.4. **Soundness.** $\lambda_{\mathrm{SEC}i}$ has the basic property expected from a typed language, that well-typed programs will not go wrong [WF94]. More details on the supporting lemmas can be found in Appendix A.

Lemma 2.2 (Subject reduction)**.**
If $\Delta^\star;\Gamma \vdash e : \sigma$ *and* $e \rightsquigarrow e'$ *then* $\Delta^\star;\Gamma \vdash e' : \sigma$.

$$\frac{\Delta^\star \vdash \tau : \star^{\ell_1}}{\Delta^\star \vdash (\tau)\ @\ \ell_2}\ \textsc{wftp:con} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \xrightarrow{\ell} \sigma_2}\ \textsc{wftp:arr} \qquad \frac{\Delta^\star \vdash \sigma_1 \qquad \Delta^\star \vdash \sigma_2}{\Delta^\star \vdash \sigma_1 \times^{\ell} \sigma_2}\ \textsc{wftp:prod}$$

$$\frac{\Delta^\star, \alpha{:}\star^{\ell_1} \vdash \sigma}{\Delta^\star \vdash \forall^{\ell_2}\alpha{:}\star^{\ell_1}.\sigma}\ \textsc{wftp:all}$$

Figure 11: Type well-formedness rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{\Delta^\star \vdash \sigma}{\Delta^\star \vdash \sigma \le \sigma}\ \textsc{sbt:refl} \qquad \frac{\Delta^\star \vdash \sigma_1 \le \sigma_2 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_3}{\Delta^\star \vdash \sigma_1 \le \sigma_3}\ \textsc{sbt:trans}$$

$$\frac{\Delta^\star \vdash \tau_1 = \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \le (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con}$$

$$\frac{\Delta^\star \vdash \tau_1 \to \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1 \to \tau_2)\ @\ \ell_2 \le (\tau_1)\ @\ \ell_2 \xrightarrow{\ell_2} (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con-arr1}$$

$$\frac{\Delta^\star \vdash \tau_1 \to \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \xrightarrow{\ell_2} (\tau_2)\ @\ \ell_2 \le (\tau_1 \to \tau_2)\ @\ \ell_2}\ \textsc{sbt:con-arr2}$$

$$\frac{\Delta^\star \vdash \tau_1 \times \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1 \times \tau_2)\ @\ \ell_2 \le (\tau_1)\ @\ \ell_2 \times^{\ell_2} (\tau_2)\ @\ \ell_2}\ \textsc{sbt:con-prod1}$$

$$\frac{\Delta^\star \vdash \tau_1 \times \tau_2 : \star^{\ell_1}}{\Delta^\star \vdash (\tau_1)\ @\ \ell_2 \times^{\ell_2} (\tau_2)\ @\ \ell_2 \le (\tau_1 \times \tau_2)\ @\ \ell_2}\ \textsc{sbt:con-prod2}$$

$$\frac{\Delta^\star \vdash \sigma_3 \le \sigma_1 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_4 \qquad \ell_1 \sqsubseteq \ell_2}{\Delta^\star \vdash \sigma_1 \xrightarrow{\ell_1} \sigma_2 \le \sigma_3 \xrightarrow{\ell_2} \sigma_4}\ \textsc{sbt:arr}$$

$$\frac{\Delta^\star \vdash \sigma_1 \le \sigma_3 \qquad \Delta^\star \vdash \sigma_2 \le \sigma_4 \qquad \ell_1 \sqsubseteq \ell_2}{\Delta^\star \vdash \sigma_1 \times^{\ell_1} \sigma_2 \le \sigma_3 \times^{\ell_2} \sigma_4}\ \textsc{sbt:prod}$$

$$\frac{\Delta^\star, \alpha{:}\star^{\ell_4} \vdash \sigma_1 \le \sigma_2 \qquad \ell_4 \sqsubseteq \ell_2 \qquad \ell_1 \sqsubseteq \ell_3}{\Delta^\star \vdash \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1 \le \forall^{\ell_3}\alpha{:}\star^{\ell_4}.\sigma_2}\ \textsc{sbt:all}$$

Figure 12: Subtyping rules for $\lambda_{\mathrm{SEC}i}$.

$$\frac{\alpha \mapsto R \in \eta \qquad \nu_1 R \nu_2}{\eta \vdash \nu_1 \sim \nu_2 : \alpha}\ \text{LR:VAR} \qquad \frac{}{\eta \vdash \nu \sim \nu : \mathsf{bool}}\ \text{LR:BOOL}$$

$$\frac{\forall(\eta \vdash e_1 \approx e_2 : \sigma_1).\eta \vdash \nu_1 e_1 \approx \nu_2 e_2 : \sigma_2}{\eta \vdash \nu_1 \sim \nu_2 : \sigma_1 \to \sigma_2}\ \text{LR:ARR}$$

$$\frac{\eta \vdash \mathbf{fst}\ \nu_1 \approx \mathbf{fst}\ \nu_2 : \sigma_1 \qquad \eta \vdash \mathbf{snd}\ \nu_1 \approx \mathbf{snd}\ \nu_2 : \sigma_2}{\eta \vdash \nu_1 \sim \nu_2 : \sigma_1 \times \sigma_2}\ \text{LR:PROD}$$

$$\frac{\forall\tau_1,\tau_2.\forall(R \in \tau_1 \leftrightarrow \tau_2).\eta,\alpha \mapsto R \vdash \nu_1[\tau_1] \approx \nu_2[\tau_2] : \sigma \qquad R \text{ consistent}}{\eta \vdash \nu_1 \sim \nu_2 : \forall\alpha{:}\star.\sigma}\ \text{LR:ALL}$$

$$\frac{e_1 \rightsquigarrow^* \nu_1 \qquad e_2 \rightsquigarrow^* \nu_2 \qquad \eta \vdash \nu_1 \sim \nu_2 : \sigma}{\eta \vdash e_1 \approx e_2 : \sigma}\ \text{LR:TERM} \qquad \frac{e_1 \uparrow \qquad e_2 \uparrow}{\eta \vdash e_1 \approx e_2 : \sigma}\ \text{LR:DIVR}$$

Figure 13: Logically related terms in the polymorphic λ-calculus.

Proof. Follows by induction over the structure of $e \rightsquigarrow e'$ making use of Lemmas A.6 (Inversion for typing), A.2 (Inversion for constructor well-formedness), A.3 (Weak-head reduction equivalence), and A.10 (Substitution commutes with equivalence). □

Lemma 2.3 (Progress)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *then* e *is a value or there exists a derivation* $e \rightsquigarrow e'$*.*

Proof. By straightforward induction over the structure of $\cdot;\cdot \vdash e : \sigma$, using Lemmas A.15 (Canonical forms for terms), A.13 (Weak head reduction terminates), and A.14 (Canonical forms for constructors). □

Definition 2.4 (Nontermination)**.** If $\cdot;\cdot \vdash e : \sigma$ and there does not exist a derivation $e \rightsquigarrow^* \nu$ then $e \uparrow$. ■

Theorem 2.5 (Type safety)**.** *If* $\cdot;\cdot \vdash e : \sigma$ *then there exists a derivation that* $e \rightsquigarrow^* \nu$ *or* $e \uparrow$*.*

Proof. Proof by contradiction using Lemmas 2.2 and 2.3. □

3. Generalizing Parametricity

The parametricity theorem has long been used to reason about programs in languages with parametric polymorphism [Rey74, Rey83]. For example, the theorem can be used to show that different implementations of an abstract datatype do not influence the behavior of the program or to show that external modules cannot forge values of abstract types. These are only a few of the corollaries of the parametricity theorem. This subsection starts with an overview of the standard parametricity theorem, and then examines how it can be generalized for $\lambda_{\text{SEC}i}$.

$$\frac{\forall \alpha{:}\star \in \Delta^{\star}.(\eta(\alpha) \in \delta_1(\alpha) \leftrightarrow \delta_2(\alpha))}{\eta \vdash \delta_1 \approx \delta_2 : \Delta^{\star}}\ \textsc{tslr:base} \qquad \frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma}\ \textsc{slr:base}$$

Figure 14: Related substitutions in the polymorphic λ-calculus.

3.1. **Parametricity.** For expository purposes, this subsection and the following subsection only consider the core of $\lambda_{\mathrm{SEC}i}$ without type constructors, security labels, or type analysis. That is, we consider a simple predicative polymorphic λ-calculus [Gir72, Rey74]. None of the results presented in these sections are new. Informally, given a logical relation inductively defined on types, the parametricity theorem states that well-typed expressions, after applying related substitutions for their free type and term variables, are related to themselves by the logical relation. The power of the theorem comes from the fact that terms typed by universally quantified type variables can be related by any relation.

The logical relation used by the parametricity theorem is defined in Figure 13. Terms are related with the judgment $\eta \vdash e_1 \approx e_2 : \sigma$, read as "terms e_1 and e_2 are related at type σ with respect to the relations in η." Terms are related if they evaluate to related values, or both diverge.

The judgment $\eta \vdash v_1 \sim v_2 : \sigma$ means that "values v_1 and v_2 are related at type σ with respect to the relations in η". The relation between values is defined inductively over types σ, potentially containing free type variables. To account for these variables, the relations are parametrized by a map, η, from type variables to binary relations on values. This map is used when σ is a type variable (see rule LR:VAR). If σ is bool, the relation is identity. Typical for logical relations, values of function type are related only if, when applied to related arguments, they produce related results. Likewise, values of product types are related if the projections of their components are related.

The most important rule, LR:ALL, defines the relationship between values of type $\forall \alpha{:}\star.\sigma$. Polymorphic values are related if their instantiations with *any* pair of types are related. Furthermore, *any* consistent relation R between values of those types as the relation on α can be used.

Definition 3.1 (Relations between values)**.** We define $\sigma_1 \leftrightarrow \sigma_2$ to be the set of all binary relations between values of type σ_1 and values of type σ_2. We then use the notation $R \in \sigma_1 \leftrightarrow \sigma_2$ to mean that R is a binary relation between values with the closed type σ_1 and values with the closed type σ_2. ■

The properties of a consistent relation are dependent upon the details of the language and the proof. Our requirements for consistency are very easy to meet, but we will wait until it is required by the proofs to explain them. If quantification over types of higher kind were allowed, R would have to be a function on relations. This extension is orthogonal to our result, so we restrict ourselves to polymorphism over kind $\star$.

To state the parametricity theorem, the notion of related substitutions for types and related terms must be defined. In Figure 14, the rule TSLR:BASE states that a relation mapping η is well-formed with respect to two type substitutions $\delta_1$and δ_2 for the variables in the type context $\Delta^{\star}$. There are no restrictions on the range of the type substitutions. On the other hand, SLR:BASE requires that a pair of term substitutions for the variables in Γ must map to related terms. Even though $\lambda_{\mathrm{SEC}i}$ has a call-by-value semantics, term substitutions must

$$\frac{}{\mathbf{true} \preceq \mathbf{true}}\ \text{ER:TRUE} \qquad \frac{}{\mathbf{false} \preceq \mathbf{false}}\ \text{ER:FALSE} \qquad \frac{}{x \preceq x}\ \text{ER:VAR}$$

$$\frac{e_1 \preceq e_2}{\lambda x{:}\sigma.e_1 \preceq \lambda x{:}\sigma.e_2}\ \text{ER:ABS} \qquad \frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{e_1 e_2 \preceq e_3 e_4}\ \text{ER:APP} \qquad \frac{e_1 \preceq e_3 \qquad e_2 \preceq e_4}{\langle e_1, e_2\rangle \preceq \langle e_3, e_4\rangle}\ \text{ER:PAIR}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fst}\, e_1 \preceq \mathbf{fst}\, e_2}\ \text{ER:FST} \qquad \frac{e_1 \preceq e_2}{\mathbf{snd}\, e_1 \preceq \mathbf{snd}\, e_2}\ \text{ER:SND} \qquad \frac{e_1 \preceq e_2}{\Lambda\alpha{:}\star.e_1 \preceq \Lambda\alpha{:}\star.e_2}\ \text{ER:TABS}$$

$$\frac{e_1 \preceq e_2}{e_1[\tau] \preceq e_2[\tau]}\ \text{ER:TAPP} \qquad \frac{e_1 \preceq e_2}{\mathbf{fix}_\omega\ x{:}\sigma.e_1 \preceq \mathbf{fix}_\omega\ x{:}\sigma.e_2}\ \text{ER:FIX} \qquad \frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}_n\ x{:}\sigma.e_2}\ \text{ER:FIXN1}$$

$$\frac{e_1 \preceq e_2}{\mathbf{fix}_n\ x{:}\sigma.e_1 \preceq \mathbf{fix}_\omega\ x{:}\sigma.e_2}\ \text{ER:FIXN2} \qquad \frac{e_1 \preceq e_4 \qquad e_2 \preceq e_5 \qquad e_3 \preceq e_6}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \preceq \mathbf{if}\ e_4\ \mathbf{then}\ e_5\ \mathbf{else}\ e_6}\ \text{ER:IF}$$

Figure 15: The erasure relation.

map to terms, not values. Otherwise, it would it be impossible to prove the case for fixed points, which requires a term substitution.

With these definitions it is possible to state the parametricity theorem for our restricted language:

Theorem 3.2 (Parametricity). *If* $\Delta^\star;\Gamma \vdash e : \sigma$ *and*
$\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and*
$\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$*, then*
$\eta \vdash \delta_1(\gamma_1(e)) \approx \delta_2(\gamma_2(e)) : \sigma$.

Proof. By induction on the typing judgment with appeals to supporting lemmas. □

One complication in this proof arises in the case for type application, where we would like to show that a term $v[\tau]$ is related to itself (after appropriate substitutions) at type $\sigma[\tau/\alpha]$. By the induction hypothesis, we know that v is related to itself at type $\forall\alpha{:}\star.\sigma$, so by inversion of the rule LR:ALL we can conclude that $v[\tau]$ is related to itself at type σ, where the type α is mapped to any relation R. However, what we need to show is that $v[\tau]$ is related to itself at type $\sigma[\tau/\alpha]$. The trick is to instantiate R with the relation $\{(v_1,v_2) \mid \eta \vdash v_1 \approx v_2 : \tau\}$ and use the following substitution lemma.

Lemma 3.3 (Constructor substitution for related terms).
If $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *then*
$\eta \vdash e_1 \approx e_2 : \sigma[\tau/\alpha]$ *iff*
$\eta,\alpha \mapsto R \vdash e_1 \approx e_2 : \sigma$*, where*
R *is the relation* $\{(v_1,v_2) \mid \eta \vdash v_1 \approx v_2 : \tau\}$ *and* $\delta_i(\alpha) = \delta_i(\tau)$.

Proof. The proof in both directions of the biconditional is by induction on the structure of the term relation. □

Another significant complication in the proof of Theorem 3.2 is circularity in relating fixed points. To show that $\mathbf{fix}\ x{:}\sigma.e$ is related to itself we must show that e is related to itself under an extended term substitution where $\gamma_1(x) = \gamma_1(\mathbf{fix}\ x{:}\sigma.e)$ and $\gamma_2(x) = \gamma_2(\mathbf{fix}\ x{:}\sigma.e)$. However, for these substitutions to be related, we need to know that the fixed point is related to itself. But showing that the fixed point is related to itself is exactly what we are trying to show! To escape this circularity we apply a syntactic technique from Pitts [Pit05]. We define a bounded fixed point expression that can only be unfolded a finite number of times before diverging The term $\mathbf{fix}_{n+1}\ x{:}\sigma.e$ unwinds to $e[(\mathbf{fix}_n\ x{:}\sigma.e)/x]$, and $\mathbf{fix}_0\ x{:}\sigma.e$ unwinds to itself. While the intent is that $\mathbf{fix}_0\ x{:}\sigma.e$ always diverges, it is not directly an axiom.

Lemma 3.4 ($\mathbf{fix}_0$ always diverges)**.** $\mathbf{fix}_0\ x{:}\sigma.e \uparrow$.

Proof. Proof by contradiction, assuming there exists a derivation $\mathbf{fix}_0\ x{:}\sigma.e \rightsquigarrow^* v$. □

We will now write using the notation $\mathbf{fix}_\omega\ x{:}\sigma.e$ for what we wrote previously as $\mathbf{fix}\ x{:}\sigma.e$, to emphasize that it is a limit. However, this is merely notational and we are not truly using ω as an ordinal number. Therefore, fixed point terms like $\mathbf{fix}_{\omega+1}\ x{:}\sigma.e$ are not allowed.

Now that fixed points may be annotated with an index, we can define a partial order on terms called the erasure relation. The definition of this relation is given in Figure 15. The relation orders terms by whether fixed point expressions are annotated. The granularity of the order could be made finer by also ordering fixed point expressions by their bound, but it is unnecessary for our proofs. For example, the order $\mathbf{fix}_1\ y{:}\mathsf{bool}.\mathbf{true} \preceq \mathbf{fix}_\omega\ y{:}\mathsf{bool}.\mathbf{true}$ holds but $\mathbf{fix}_1\ y{:}\mathsf{bool}.\mathbf{true} \preceq \mathbf{fix}_2\ y{:}\mathsf{bool}.\mathbf{true}$ does not.

An important property of fixed point expressions is that if a fixed point expression reduces to a value, then it must have unfolded itself a finite number of times. The following lemma formalizes this property.

Lemma 3.5 (Unwinding evaluation equivalence)**.**

$\mathbf{fix}_\omega\ x{:}\sigma.e' \rightsquigarrow^* v$ *iff exists* n *such that for all* $m, m \geq n$ *implies* $\mathbf{fix}_m\ x{:}\sigma.e' \rightsquigarrow^* v'$ *where* $v' \preceq v$.

Proof. Both directions follow by straightforward induction over the number of reduction steps. □

At this point we can define our notion of consistency: only those relations R that cannot depend upon finite approximations of fixed points can be quantified over. More precisely, if $v_1 R v_2$ and v_1' is an erasure of v_2 and v_2' is an erasure of v_2 then R must also relate v_1' and v_2'. For example, the relation

$$\{(\lambda x{:}\mathsf{bool}.\mathbf{fix}_{n_1}\ y{:}\mathsf{bool}.\mathbf{true}, \lambda x{:}\mathsf{bool}.\mathbf{fix}_{n_2}\ y{:}\mathsf{bool}.\mathbf{true}) \mid n_1 = n_2\},$$

is not consistent because it will relate $\lambda x{:}\mathsf{bool}.\mathbf{fix}_7\ y{:}\mathsf{bool}.\mathbf{true}$ and $\lambda x{:}\mathsf{bool}.\mathbf{fix}_7\ y{:}\mathsf{bool}.\mathbf{true}$, but not $\lambda x{:}\mathsf{bool}.\mathbf{fix}_\omega\ y{:}\mathsf{bool}.\mathbf{true}$ and $\lambda x{:}\mathsf{bool}.\mathbf{fix}_7\ y{:}\mathsf{bool}.\mathbf{true}$.

The logical relation itself is closed under erasure, making it a consistent relation.

Lemma 3.6 (Logical relation is closed under erasure)**.**

- *If* $\eta \vdash v_1 \sim v_2 : \tau$ *and* $v_1 \preceq v_1'$ *and* $v_2 \preceq v_2'$ *then* $\eta \vdash v_1' \sim v_2' : \tau$
- *If* $\eta \vdash e_1 \approx e_2 : \tau$ *and* $e_1 \preceq e_1'$ *and* $e_2 \preceq e_2'$ *then* $\eta \vdash e_1' \approx e_2' : \tau$

Proof. The proof follows by straightforward mutual induction over the structure of $\eta \vdash v_1 \sim v_2 : \tau$ and $\eta \vdash e_1 \approx e_2 : \tau$. □

It is now straightforward to show that, for any n, $\mathbf{fix}_n\ x{:}\sigma.e$ is related to itself. Then the following continuity lemma can be used to prove that fixed point limits are related to themselves.

Lemma 3.7 (Continuity). *If* $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and*
for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$
where $\delta_1(\sigma) = \sigma_1$ *and* $\delta_2(\sigma) = \sigma_2$ *then*
$\eta \vdash \mathbf{fix}_\omega\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_\omega\ x{:}\sigma_2.e_2 : \sigma$.

Proof. There are four cases.

- If both $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ diverge, they are trivially related by LR:DIVR.
- If both $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ converge to a value, they must do so with some finite number of unwindings as specified by Lemma 3.5, m. It is possible to instantiate the assumption, for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$, accordingly, to obtain the a derivation $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$. By inversion this means either both $\mathbf{fix}_m\ x{:}\sigma_i.e_i$ diverge or converge to related values, $\eta \vdash v_1 \sim v_2 : \sigma$. However, they must converge after at most $m - 1$ unwindings, therefore it is the case that they converge to related values. Furthermore, $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ evaluates to v_i', which is an erasure of v_i. Because the logical relation is closed under erasure, it is the case that $\eta \vdash v_1' \sim v_2' : \sigma$. Finally because both $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ converge to v_i' the rule LR:TERM can be used to conclude $\eta \vdash \mathbf{fix}_\omega\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_\omega\ x{:}\sigma_2.e_2 : \sigma$.
- In the last two cases, one of $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ diverges and the other converges to a value. However, the fixed point that converged must do so in a finite number of unwindings m, as described by Lemma 3.5. Then instantiating for all n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$ with m we have a derivation that $\eta \vdash \mathbf{fix}_m\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_m\ x{:}\sigma_2.e_2 : \sigma$. By inversion we know that either both $\mathbf{fix}_m\ x{:}\sigma_i.e_i$ converge or diverge. However, we already know that one of the expressions converges, therefore the other must as well. However, we know that $\mathbf{fix}_n\ x{:}\sigma_i.e_i$ terminates iff $\mathbf{fix}_\omega\ x{:}\sigma_i.e_i$ does. This contradicts the assumption that only one of the two fixed points converged to a value.

□

3.2. **Applications of the parametricity theorem.** The parametricity theorem has been used for many purposes, most famously for deriving *free theorems* about functions in the polymorphic λ-calculus, from their types alone [Wad89]. Our purpose is more similar to that of Reynolds [Rey74, Rey83]: reasoning about representation independence properties.

Corollaries of Theorem 3.2 provide important results for reasoning about abstract types in programs. Many specific properties can be proven as a consequence of the parametricity theorem, but we believe the following two are representative of what a programmer desires.

This first corollary says that a programmer is free to change the implementation of an abstract type without affecting the behavior of a program. It is the essence behind parametric polymorphism – type information is not allowed to influence program execution, and values of abstract type are be treated as "black boxes".

Corollary 3.8 (Confidentiality). *If* $\cdot \vdash v_1 : \tau_1$ *and*
$\cdot \vdash v_2 : \tau_2$, *then*
$\alpha{:}\star; x{:}\alpha \vdash e : \mathsf{bool}$ *and*
$e[\tau_1/\alpha][v_1/x] \rightsquigarrow^* v$ *iff* $e[\tau_2/\alpha][v_2/x] \rightsquigarrow^* v$.

Proof. First construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star.\lambda x{:}\alpha.e : \forall\alpha{:}\star.\alpha \to \mathsf{bool}$ using the appropriate typing rules and then appeal to Theorem 3.2 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star.\lambda x{:}\alpha.e \sim \Lambda\alpha{:}\star.\lambda x{:}\alpha.e : \forall\alpha{:}\star.\alpha \to \mathsf{bool}.$$

Next, by inversion on LR:ALL and instantiation with the relation

$$R = \{(v_1, v_2) \mid (\cdot;\cdot \vdash v_1 : \tau_1), (\cdot;\cdot \vdash v_2 : \tau_2)\},$$

it can be concluded that

$$\cdot,\alpha \mapsto R \vdash (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_1] \approx (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_2] : \alpha \to \mathsf{bool}.$$

By straightforward application of LR:VAR it is possible to conclude

$$\cdot,\alpha \mapsto R \vdash v_1 \sim v_2 : \alpha,$$

so by application of LR:TERM, inversion on LR:ARR, and instantiation

$$\cdot,\alpha \mapsto R \vdash (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_1]v_1 \approx (\Lambda\alpha{:}\star.\lambda x{:}\alpha.e)[\tau_2]v_2 : \mathsf{bool}.$$

Finally, because the relation is closed under reduction we have LR:ARR, and by instantiation it is true that

$$\cdot,\alpha \mapsto R \vdash e[\tau_1/\alpha][v_1/x] \approx e[\tau_2/\alpha][v_2/x] : \mathsf{bool},$$

from which the desired conclusion can be obtained by simple inversion. □

This second corollary states that there is no way for a program to invent values of an abstract type, and thereby allowing the integrity of the abstraction to be violated. The integrity of the abstraction can be thought of as unspecified invariants.

Corollary 3.9 (Integrity)**.** *If* $\alpha{:}\star;\cdot \vdash e : \alpha$ *then* $e[\tau/\alpha]$ *for any* τ *must diverge.*

Proof. First construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star.e : \forall\alpha{:}\alpha$ using the appropriate typing rules, then appeal to Theorem 3.2 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star.e \sim \Lambda\alpha{:}\star.e : \forall\alpha{:}\star.\alpha.$$

Now assume an arbitrary τ. By inversion on LR:ALL and by instantiation it is possible to conclude

$$\cdot,\alpha \mapsto \varnothing \vdash (\Lambda\alpha{:}\star.e)[\tau] \approx (\Lambda\alpha{:}\star.e)[\tau] : \alpha.$$

Because the relation is closed under reduction it is true that

$$\cdot,\alpha \mapsto \varnothing \vdash e[\tau/\alpha] \approx e[\tau/\alpha] : \alpha.$$

Furthermore, by inversion either $e[\tau/\alpha] \rightsquigarrow^* v$ or $e[\tau/\alpha]\uparrow$. However in the former case that would mean that

$$\cdot,\alpha \mapsto \varnothing \vdash v \sim v : \alpha,$$

which by inversion on LR:VAR is impossible because there is no v such that $v\varnothing v$. Therefore $e[\tau/\alpha]\uparrow$. □

3.3. **Parametricity and type analysis.** We now consider the problem of extending the parametricity theorem to all of $\lambda_{\mathrm{SEC}i}$. There are two primary difficulties in doing so.

As an example of the first problem, the following $\lambda_{\mathrm{SEC}i}$ term (eliding labels) violates Corollary 3.8:

$$\textbf{typecase}\ [\gamma.\mathsf{bool}]\ \alpha\ \textbf{true}\ (\Lambda\beta{:}\star.\Lambda\delta{:}\star.\textbf{false})(\Lambda\beta{:}\star.\Lambda\delta{:}\star.\textbf{false}),$$

This expression contradicts confidentiality because substituting bool for α and substituting $\mathsf{bool}\times\mathsf{bool}$ for α will cause the expression to evaluate to different values: **true** versus **false**. It is not possible to directly extend the proof of parametricity to handle **typecase**. The proof would require that the two terms produce related results, even when they may analyze different constructors.

Still, we would like to state properties similar to Corollaries 3.8 and 3.9 for $\lambda_{\mathrm{SEC}i}$. The problem we describe above can be solved by strengthening the definition of the logical relation. Specifically, by changing the rule LR:ALL to require that τ_1 and τ_2 are β-equivalent:

$$\frac{\forall\tau_1,\tau_2.\tau_1=\tau_2:\star,\forall(R\in\tau_1\leftrightarrow\tau_2).\eta,\alpha\mapsto R\vdash\nu_1[\tau_1]\approx\nu_{\ 2}[\tau_2]:\sigma \qquad R\ \text{consistent}}{\eta\vdash\nu_1\sim\nu_2:\forall\alpha{:}\star.\sigma}\ \textsc{lr:all-eq}$$

This revised version of LR:ALL does allow a stronger version of Corollary 3.8 to be proven in the presence of **typecase**, but it is so strong that it is vacuous. The example above is resolved simply because the theorem only says anything about the behavior when substituting β-equivalent constructors for α.

This is why tracking information-flow is critical – it allows for a richer definition of equivalence for constructors than β-equivalence. For example, here is the earlier example annotated with information-flow labels:

$$\textbf{typecase}\ [\gamma.(\mathsf{bool})\ @\ \top]\ \alpha\ \textbf{true}\ (\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\textbf{false})\ (\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\textbf{false})$$

If α has kind $\star^{\top}$ then as specified by the typing rule WFT:TCASE, the entire expression will have type $(\mathsf{bool})\ @\ \top$. As before substituting $\mathsf{bool}\times\mathsf{bool}$ for α will cause the expression to evaluate to different values: **true** versus **false**. However, in an information-flow type system, equivalence is parametrized by an observer. If the observer is only allowed to observe data with an information content less than $\top$, to that observer **true** and **false** at type $(\mathsf{bool})\ @\ \top$ will be indistinguishable. The next section will explain in more detail what it means for constructors to be related in an information-flow kind system.

A second problem that arises when trying to prove a generalization of the parametricity theorem for $\lambda_{\mathrm{SEC}i}$ is simply defining the relation. Logical relations are defined inductively over the types of the language. However, in $\lambda_{\mathrm{SEC}i}$ the weak-head normal forms of types include (for example) $\mathsf{Typerec}$ with its scrutinee a variable. It is not obvious what it means for two values to be related at a type like

$$\mathsf{Typerec}\ \alpha\ \mathsf{bool}\ (\lambda\beta{:}\star^{\bot}\lambda\delta{:}\star^{\bot}.\mathsf{bool}\to\mathsf{bool})\ (\lambda\beta{:}\star^{\bot}\lambda\delta{:}\star^{\bot}.\mathsf{bool}\times\mathsf{bool}).$$

The solution that we use, for $\lambda_{\mathrm{SEC}i}$, is to quantify over families of relations between values instead of merely quantifying over relations between values of two specific types. We will explain how this works in more detail when we revisit the logical relation for expressions in Section 3.5.

$$\begin{array}{lll}
\textit{constructor contexts} & \xi ::= \bullet \mid \mathsf{Typerec}\ \xi\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} \mid \xi\,\tau \\
\textit{whnf constructors} & \nu ::= \xi\{\alpha\} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \lambda\alpha{:}\kappa.\tau \\
\textit{whnf types} & \zeta ::= (\mathsf{bool}) \mathbin{@} \ell \mid (\xi\{\alpha\}) \mathbin{@} \ell \mid \sigma_1 \xrightarrow{\ell} \sigma_2 \mid \sigma_1 \times^{\ell} \sigma_2 \mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma
\end{array}$$

Figure 16: The grammar of additional syntactic forms in $\lambda_{\mathrm{SEC}i}$.

$$\frac{\ell_1 \not\sqsubseteq \ell_o}{\nu_1 \sim_{\ell_o} \nu_2 : \star^{\ell_1}}\ \textsc{tslr:type-opaq} \qquad \frac{\ell_1 \sqsubseteq \ell_o}{\mathsf{bool} \sim_{\ell_o} \mathsf{bool} : \star^{\ell_1}}\ \textsc{tslr:type-bool}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \to \tau_2 \sim_{\ell_o} \tau_3 \to \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-arr}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_o \qquad \tau_1 \approx_{\ell_o} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_o} \tau_4 : \star^{\ell_2}}{\tau_1 \times \tau_2 \sim_{\ell_o} \tau_3 \times \tau_4 : \star^{\ell_3}}\ \textsc{tslr:type-prod}$$

$$\frac{\forall(\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1).\nu_1\tau_1 \approx_{\ell_o} \nu_2\tau_2 : \kappa_2 \sqcup \ell_1}{\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell_1} \kappa_2}\ \textsc{tslr:arr}$$

$$\frac{\tau_1 \rightsquigarrow^* \nu_1 \qquad \tau_2 \rightsquigarrow^* \nu_2 \qquad \nu_1 \sim_{\ell_o} \nu_2 : \kappa}{\tau_1 \approx_{\ell_o} \tau_2 : \kappa}\ \textsc{tsclr:base}$$

Figure 17: Logically related constructors in $\lambda_{\mathrm{SEC}i}$.

3.4. **Equivalence of constructors.** The first step towards a generalized parametricity theorem is formalizing what it means for type constructors to be equivalent in an information-flow kind system. Instead of defining the equivalence inductively over the structure of constructors, like in Figure 10, we define a logical relation between constructors inductively over their kinds.

We write $\tau_1 \approx_\ell \tau_2 : \kappa$ to mean closed constructors τ_1 and τ_2 are related at kind κ with respect to a label, ℓ, called the *observer*. Similarly, the judgment $\nu_1 \sim_\ell \nu_2 : \kappa$ is used to indicate that closed weak-head normal constructors ν_1 and ν_2 are related at kind κ with respect to an observer, ℓ. The grammar of weak-head normal constructors and relations on constructors is defined in Figures 16 and 17, respectively. We implicitly require for $\nu_1 \sim_\ell \nu_2 : \kappa$ and $\tau_1 \approx_\ell \tau_2 : \kappa$ that $\cdot \vdash \nu_1,\nu_2 : \kappa$ and $\cdot \vdash \tau_1,\tau_2 : \kappa$ respectively.

Making the distinction between constructors and weak-head normal constructors is especially useful because the head of closed weak-head normal form for constructors will never be Typerec.

Constructors that are not in normal form are related by TSCLR:BASE if and only if their weak-head normal forms are related. The rule for type functions, TSLR:ARR, is standard for logical relations.

An anthropomorphic interpretation of the observer is of an individual with the clearance to inspect data with an information content below a specific label in the label lattice. If the observer is an administrator she may be cleared to inspect data with an information content less than $\top$. Guest users of a system might only be allowed to inspect data with an information content of $\bot$. Because such users cannot inspect data with an information content higher than $\bot$, all data with such an information content will appear identical to them. This restriction is enforced by the rule TSLR:TYPE-OPAQ in Figure 17. For example, $\mathsf{bool} : \star^{\top}$ and $\mathsf{bool} \times \mathsf{bool} : \star^{\top}$ which carry "high-security" information $\top$, will be indistinguishable to an observer at a "low-security" level $\bot$. Otherwise, the standard equivalence rules TSLR:TYPE-BOOL, TSLR:TYPE-ARR, and TSLR:TYPE-PROD are used.

More formally, the observer label can be understood as a parameter that quotients the logical relation. If the observer is $\top$ then the relation is $\beta\eta$-equivalence of constructors. [2] If the observer is some label, ℓ, less than $\top$, then the relation is $\beta\eta$-equivalence for those constructors with an information content less than or equal to ℓ, and the universal relation for constructors with an information content greater than ℓ.

While the logical relation on constructors was designed so that the it will be the universal relation when the observer is lower than the information content of the constructors, it is not an axiom. Therefore, it is wise to check the definitions by proving the following lemma.

Lemma 3.10 (Obliviousness for constructors)**.** *If* $\cdot \vdash \tau_1,\tau_2 : \kappa$ *and* $\mathcal{L}(\kappa) \not\sqsubseteq \ell_o$*, then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa$.

Proof. Follows from the use of Lemma A.13 (Weak head reduction terminates) and straightforward induction upon κ. □

Another important property of the relation is that it is closed under subsumption. The following lemma verifies the intuition that two related constructors will always stay related when made more restricted.

Lemma 3.11 (Constructor relation is closed under subsumption)**.** *If* $\kappa_1 \leq \kappa_2$ *and*

- $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1$ *then* $\tau_1 \sim_{\ell_o} \tau_2 : \kappa_2$.
- $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$ *then* $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_2$

Proof. By straightforward mutual induction over κ_1. □

Finally, because we have defined equivalence on constructors in terms of a logical relation, it is useful (and later necessary) to prove a result for type constructors that is similar to parametricity for terms. However, first we must provide a revised definition of what it means for two constructor substitutions to be related.

Definition 3.12 (Related constructor substitutions)**.**

$$\frac{\forall \alpha{:}\kappa \in \Delta.(\delta_1(\alpha) \approx_{\ell_o} \delta_2(\alpha) : \kappa)}{\delta_1 \approx_{\ell_o} \delta_2 : \Delta}\ \text{TSSLR:BASE}$$

[2] The relation is $\beta\eta$-equivalence for type functions, but only β-equivalence for Typerec. The reason for this difference is because the logical relation for constructor equivalence is inductively defined on kinds, and because Typerec does not introduce a distinguished kind, the only equivalences defined for Typerec constructors are given by the rule TSCLR:BASE.

$$\frac{\tau \rightsquigarrow \tau'}{(\tau)\ @\ \ell \rightsquigarrow (\tau')\ @\ \ell}\ \textsc{WHR:INJ-TC} \qquad \frac{}{(\tau_1 \to \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \xrightarrow{\ell} (\tau_2)\ @\ \ell}\ \textsc{WHR:INJ-ARR}$$

$$\frac{}{(\tau_1 \times \tau_2)\ @\ \ell \rightsquigarrow (\tau_1)\ @\ \ell \times^{\ell} (\tau_2)\ @\ \ell}\ \textsc{WHR:INJ-PROD}$$

Figure 18: Type reduction in $\lambda_{\mathrm{SEC}i}$.

$$\frac{\alpha \mapsto R \in \eta \qquad (\ell_1 \sqsubseteq \ell_o) \Longrightarrow (v_1 R_{\xi}^{\ell_1} v_2)}{\eta \vdash v_1 \sim_{\ell_o} v_2 : (\xi\{\alpha\})\ @\ \ell_1}\ \textsc{SLR:CON} \qquad \frac{(\ell_1 \sqsubseteq \ell_o) \Longrightarrow (v_1 = v_2)}{\eta \vdash v_1 \sim_{\ell_o} v_2 : (\mathsf{bool})\ @\ \ell_1}\ \textsc{SLR:BOOL}$$

$$\frac{\forall(\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1).\eta \vdash v_1 e_1 \approx_{\ell_o} v_2 e_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \xrightarrow{\ell_1} \sigma_2}\ \textsc{SLR:ARR}$$

$$\frac{\eta \vdash \mathbf{fst}\, v_1 \approx_{\ell_o} \mathbf{fst}\, v_2 : \sigma_1 \sqcup \ell_1 \qquad \eta \vdash \mathbf{snd}\, v_1 \approx_{\ell_o} \mathbf{snd}\, v_2 : \sigma_2 \sqcup \ell_1}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \times^{\ell_1} \sigma_2}\ \textsc{SLR:PROD}$$

$$\frac{\begin{array}{l}\forall(\tau_1 \approx_{\ell_o} \tau_2 : \star^{\ell_2}).\forall(R_{\xi}^{\ell_2} \in \delta_1((\xi\{\tau_1\})\ @\ \ell_2) \leftrightarrow \delta_{\ 2}((\xi\{\tau_2\})\ @\ \ell_2)). \\ \quad \eta,\alpha \mapsto R \vdash v_1[\tau_1] \approx_{\ell_o} v_2[\tau_2] : \sigma \sqcup \ell_1 \qquad R \text{ consistent}\end{array}}{\eta \vdash v_1 \sim_{\ell_o} v_2 : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma}\ \textsc{SLR:ALL}$$

$$\frac{e_1 \rightsquigarrow^* v_1 \qquad e_2 \rightsquigarrow^* v_2 \qquad \sigma \rightsquigarrow^* \zeta \qquad \eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{SCLR:TERM}$$

$$\frac{e_1 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{SCLR:DIVR1} \qquad \frac{e_2 \uparrow}{\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma}\ \textsc{SCLR:DIVR2}$$

Figure 19: Logically related terms in $\lambda_{\mathrm{SEC}i}$.

Given the above definition the lemma is as follows:

Lemma 3.13 (Basic lemma for constructors)**.**
If $\Delta \vdash \tau : \kappa$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa$.

Proof. By induction over the structure of $\Delta \vdash \tau : \kappa$. See Appendix B for the complete details. □

Now that we have explained how equivalence on constructors is defined for $\lambda_{\mathrm{SEC}i}$, we will examine the revisions necessary to the logical relation on expressions.

3.5. **Related expressions.** As with constructors, we parametrize the logical relation on terms by an observer at level ℓ in the label lattice. We write $\eta \vdash e_1 \approx_\ell e_2 : \sigma$ to indicate that terms e_1 and e_2 are related to an observer at level ℓ at type σ, with the relation mapping η. As with constructors, we distinguish between related terms and related normal forms, writing the judgment $\eta \vdash v_1 \sim_\ell v_2 : \zeta$ to indicate that values v_1 and v_2 are related to an observer at level ℓ at the weak-head normal type ζ, with the relation mapping η. These relations, as defined in Figure 19, are similar to the ones in Figure 13. We implicitly require for $\eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta$ and $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ that $\cdot;\cdot \vdash v_1 : \delta_1(\zeta)$, $\cdot;\cdot \vdash v_2 : \delta_2(\zeta)$ and $\cdot;\cdot \vdash e_1 : \delta_1(\sigma)$, $\cdot;\cdot \vdash e_2 : \delta_2(\sigma)$ respectively where $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ and $\delta_1,\delta_2 \vdash \eta : \Delta^\star$.

One difference from the rules in Figure 13, is that we only relate values at weak-head normal types ζ, as defined in Figure 16.

Restricting the value relation to weak-head normal types makes the logical relation much easier to state and understand. For example, the term $\langle \mathbf{true}, \mathbf{false} \rangle$ is well typed with the equivalent types $(\mathsf{bool} \times \mathsf{bool})\ @\ \ell$ and $(\mathsf{bool})\ @\ \ell \times^\ell (\mathsf{bool})\ @\ \ell$. However, restricting the relation to weak-head normal types means that only the case for $(\mathsf{bool})\ @\ \ell \times^\ell (\mathsf{bool})\ @\ \ell$ must be considered in the inductive proof.

In Figure 15 we defined the erasure relation for all the terms in $\lambda_{\mathrm{SEC}i}$ except **typecase**. However, the rule for **typecase** is straightforward

$$\frac{e_{\mathsf{bool}} \preceq e'_{\mathsf{bool}} \qquad e_\to \preceq e'_\to \qquad e_\times \preceq e'_\times}{\mathbf{typecase}[\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_\to\ e_\times \preceq \mathbf{typecase}[\gamma.\sigma]\ \tau\ e'_{\mathsf{bool}}\ e'_\to\ e'_\times}\ \textsc{ler:tcase}$$

We will not restate Lemma 3.5 because **typecase** does not alter the proof in any interesting fashion.

Like constructors, the relation over terms is defined so that terms with a greater information content than the observer will be indistinguishable. This is enforced by the precondition $\ell_1 \sqsubseteq \ell_o$ found in SLR:CON and SLR:BOOL. The antecedent relations in SLR:ALL, SLR:ARR, and SLR:PROD all have their types joined with ℓ_1; this accounts for information gained by destructing the value. The following lemma verifies the intuitions concerning indistinguishability:

Lemma 3.14 (Obliviousness for terms)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\mathcal{L}(\zeta) \not\sqsubseteq \ell_o$ *and*

- $\Delta^\star;\cdot \vdash v_1,v_2 : \zeta$ *then* $\eta \vdash \delta_1(v_1) \sim_{\ell_o} \delta_2(v_2) : \zeta$.
- $\Delta^\star;\cdot \vdash e_1,e_2 : \sigma$ *then* $\eta \vdash \delta_1(e_1) \approx_{\ell_o} \delta_2(e_2) : \sigma$.

Proof. The first part follows from induction on ζ and the second part from Theorem 2.5 (Type safety). □

There are two other significant differences between Figures 13 and 19: additional preconditions in SLR:ALL, and generalizing LR:VAR to SLR:CON. The rule SLR:CON solves the problem with Typerec appearing in the weak-head normal form of types. It generalizes LR:VAR to terms related at a constructor that cannot be normalized further because of an undetermined type variable. We characterize these constructors with constructor contexts, ξ, defined in Figure 16. Contexts are holes $\bullet$, Typerecs of a context, or a context applied to an arbitrary constructor. We write $\xi\{\tau\}$ for filling a context's hole with τ.

Previously, values were related at a type variable only if they were in the relation mapped to that variable by η. Here η maps to families of relations.

Definition 3.15 (Parameterized relation)**.** A parameterized relation R is a function that when given a label ℓ and a type context ξ yields a binary relation between values of two

types. For conciseness, we use the notation R^{ℓ}_{ξ} for the application of a label and a type context to a parameterized relation. We will sometimes express that a parametrized relation belongs to the set of binary relations between closed values of type $\delta_1((\xi\{\tau_1\})\ @\ \ell)$ and closed values of type $\delta_2((\xi\{\tau_2\})\ @\ \ell)$ using the notation

$$\mathsf{R}^{\ell}_{\xi} \in \delta_1((\xi\{\tau_1\})\ @\ \ell) \leftrightarrow \delta_2((\xi\{\tau_2\})\ @\ \ell).$$

This notation can be roughly understood with dependent types as

$$\mathsf{R} : \Pi\ell.\Pi\xi.\delta_1((\xi\{\tau_1\})\ @\ \ell) \leftrightarrow \delta_2((\xi\{\tau_2\})\ @\ \ell).$$

■

This move from relations to families of relations makes it more difficult to use the resulting generalized parametricity theorem. This is primarily because in standard parametricity it is only necessary to choose a relationship between values of two fixed types, while in generalized parametricity it is necessary to choose a family of relationship between values of arbitrary type. This is because the constructor context, ξ, determines the types of the values R^{ℓ}_{ξ} must relate.

To date we have been unable to devise any non-trivial families of relations that are not parametric in their constructor context. It is open question whether there are interesting families of relations that are not parametric in their constructor context. Because constructor contexts were introduced to handle Typerec, if it were removed from $\lambda_{\mathrm{SEC}i}$ this problem would go away. There may be less drastic solutions and we will discuss some of our ideas in Section 5. Fortunately, the families of relations used to prove the confidentiality and integrity corollaries, the universal relation and the null relation, respectively, are parametric in their constructor context.

Definition 3.16 (Parameterized relation consistency)**.** We say that a parameterized relation $\mathsf{R}^{\ell}_{\rho} \in \sigma_1 \leftrightarrow \sigma_2$ is consistent if

(1) $v_1 \mathsf{R}^{\ell_1}_{\rho} v_2$ and $\ell_1 \sqsubseteq \ell_2$ then $v_1 \mathsf{R}^{\ell_2}_{\rho} v_2$ (moving up in the lattice does not change relatedness)
(2) $v_1 \preceq v_2$ and $v_3 \preceq v_4$ and $v_1 \mathsf{R}^{\ell_1}_{\rho} v_3$ then $v_2 \mathsf{R}^{\ell_1}_{\rho} v_4$ (relation does not treat finite approximations differently)

■

As with standard parametricity, quantification over R is required to be consistent. In addition to being closed under erasure of fixed point annotations, as we described for the vanilla parametricity theorem in Section 3.1, relations are required to be closed under subtyping. That means if $v_1 \mathsf{R}^{\ell_1}_{\xi} v_2$ and $\ell_1 \sqsubseteq \ell_2$ then it must also be the case that $v_1 \mathsf{R}^{\ell_2}_{\xi} v_2$.

It is important that the logical relation on values is itself consistent, that is, closed under subsumption and erasure.

Lemma 3.17 (Term relations are closed under erasure)**.**

(1) *If* $v_1' \preceq v_1$ *and* $v_2' \preceq v_2$ *and* $\eta \vdash v_1' \sim_{\ell_\circ} v_2' : \zeta$ *then* $\eta \vdash v_1 \sim_{\ell_\circ} v_2 : \zeta$.
(2) *If* $e_1' \preceq e_1$ *and* $e_2' \preceq e_2$ *and* $\eta \vdash e_1' \approx_{\ell_\circ} e_2' : \sigma$ *then* $\eta \vdash e_1 \approx_{\ell_\circ} e_2 : \sigma$

Proof. The proof of Parts 1 and 2 follows by straightforward mutual induction over the structure of $\eta \vdash v_1' \sim_{\ell_\circ} v_2' : \zeta$ and $\eta \vdash e_1' \approx_{\ell_\circ} e_2' : \sigma$. □

Lemma 3.18 (Term relations are closed under subsumption)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*

- $\Delta^\star \vdash \zeta_1 \leq \zeta_2$ *and* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_1$ *then* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_2$.
- $\Delta^\star \vdash \sigma_1 \leq \sigma_2$ *and* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_1$ *then* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma_2$

Proof. Follows from straightforward mutual induction over σ_1 and ζ_1, with uses of Part 3.11, Definition 3.16, and Lemmas A.5 (Inversion for subtyping) and B.2 (Inversion for subtyping on normal types). □

Corollary 3.19 (Value relation is consistent)**.** *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $\Delta^\star \vdash \tau : \star^\top$*, then the relation*

$$R^\rho_\ell = \{(\nu_1,\nu_2) \mid \eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : (\rho\{\tau\}) \ @ \ \ell\}$$

is consistent.

Proof. A direct consequence of Definition 3.16, Lemma 3.18 Part 3.18, and 3.17 Part 2. □

We write $\delta_1,\delta_2 \vdash \eta :\Delta^\star$ to mean that the mapping η is well-formed with respect to a pair of type substitutions, δ_1 and δ_2, as described in the following definition.

Definition 3.20 (Relation mapping regularity)**.** If $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ then

$$\frac{\forall \alpha{:}\star^{\ell_1} \in \Delta^\star.(\eta(\alpha)^{\ell_1}_\xi \in \delta_1((\xi\{\alpha\}) \ @ \ \ell_1) \leftrightarrow \delta_{\ 2}((\xi\{\alpha\}) \ @ \ \ell_1)) \qquad \eta(\alpha) \text{ consistent}}{\delta_1,\delta_2 \vdash \eta : \Delta^\star} \ \textsc{relm:reg}$$

■

The last significant difference in Figure 13 is that LR:DIVR has been split into SLR:DIVR1 and SLR:DIVR2. Terms in λ_{SECi} are related if either diverges, as opposed to our earlier definition where divergent terms were only related to other divergent terms. As a consequence, the proof of continuity for λ_{SECi} is slightly simpler:

Lemma 3.21 (Fixpoint continuity)**.** *If for all* n, $\eta \vdash \mathbf{fix}_n \ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_n \ x{:}\sigma_2.e_2 : \sigma$ *then* $\eta \vdash \mathbf{fix}_\omega \ x{:}\sigma_1.e_1 \approx_{\ell_o} \mathbf{fix}_\omega \ x{:}\sigma_2.e_2 : \sigma$ *where* $\delta_i(\sigma) = \sigma_i$.

Proof. The proof is very similar to Lemma 3.7, except that the cases where either $\mathbf{fix}_\omega \ x{:}\sigma_1.e_1$ or $\mathbf{fix}_\omega \ x{:}\sigma_2.e_2$ diverge follow trivially from SCLR:DIVR1 and SCLR:DIVR2 respectively. □

At first, this change might seem like a significant weakening of the relation. In particular, the logical relation is no longer transitive. However, this definition is standard for information-flow logical relations proofs with recursion [ABHR99, Zda02]. We will discuss how this requirement is merely an artifact of call-by-value information-flow in the next subsection.

3.6. **Generalized parametricity.** Before stating the generalized parametricity theorem, the notion of related term substitutions must be defined. Given related type substitutions, $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$, and a well-formed mapping, $\delta_1,\delta_2 \vdash \eta : \Delta^\star$, term substitutions are related if they map variables to related terms.

Definition 3.22 (Related term substitutions)**.** If $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ and $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ then

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx_{\ell_o} \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma} \ \textsc{sslr:base}$$

■

The only change from SLR:BASE is the additional of a label ℓ_o for the observer.

Finally, as for the proof of standard parametricity proving the case for type applications requires proving a substitution lemma:

Lemma 3.23 (Constructor substitution for related terms). *If* $\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and* $R^\ell_\rho = \{(\nu_1,\nu_2) \mid \eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_2\}$ *and* $\delta_i(\alpha) = \delta_i(\tau)$ *then*

(1) $\eta,\alpha \mapsto R \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_1$ *and* $(\rho\{\tau\})\ @\ \ell \rightsquigarrow^* \zeta_2$ *iff* $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta_3$ *where* $\zeta[\tau/\alpha] \rightsquigarrow^* \zeta_3$.
(2) $\eta,\alpha \mapsto R \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *and* $(\rho\{\tau\})\ @\ \ell \rightsquigarrow^* \zeta$ *iff* $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma[\tau/\alpha]$.

Proof. Follows from mutual induction over the logical relations, making use of Lemma 3.14 and Corollary 3.19. □

Theorem 3.24 (Generalized parametricity). *If* $\Delta^\star;\Gamma \vdash e : \sigma$ *and*
$\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and*
$\delta_1,\delta_2 \vdash \eta : \Delta^\star$ *and*
$\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ *then*
$\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma$.

Proof. As with standard parametricity, the proof is by induction over $\Delta^\star;\Gamma \vdash e : \sigma$. In addition to Lemma 3.23, the proof makes use of the lemmas mentioned in Section 3.4 and Section 3.5. See Appendix B for the complete details. □

We call this theorem generalized parametricity because we conjecture that Theorem 3.2 can be recovered via an encoding:

- Restrict the label lattice to two elements, $\bot$ and $\top$ where $\bot \sqsubseteq \top$.
- For every kind κ in $\Delta^\star$, Γ, e, and σ require $\mathcal{L}(\kappa) = \top$.
- For every type σ' in Γ, e, and σ require $\mathcal{L}(\sigma') = \bot$.
- Require that the observer be $\bot$.

Figure 20 makes this encoding explicit, allowing the relationship between standard parametricity and generalized parametricity to be described formally.

Conjecture 3.25 (Generalized parametricity subsumes standard parametricity).
If $\Delta^\star;\Gamma \vdash e : \sigma$ and $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ and $\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$, then
$[\![\eta]\!] \vdash [\![\delta_1(\gamma_1(e))]\!] \approx_\bot [\![\delta_2(\gamma_2(e))]\!] : [\![\sigma]\!]$ where $\delta_1(\gamma_1(e))\uparrow$ iff $\delta_2(\gamma_2(e))\uparrow$.

We expect that the proof will follow by induction over the structure of the polymorphic λ-calculus typing judgment, $\Delta^\star;\Gamma \vdash e : \sigma$.

However, this encoding is not perfect because LR:DIVR has been split into a disjunction with the rules SLR:DIVR1 and SLR:DIVR2. Therefore, Theorem 3.24 makes a weaker claim about the termination behavior of related terms than Theorem 3.2. This difference is accounted for in Conjecture 3.25 by the side condition $\delta_1(\gamma_1(e))\uparrow$ iff $\delta_2(\gamma_2(e))\uparrow$. Furthermore, the difference in how the theorems treat non-termination does impact our results – consider the generalized version of Corollary 3.8:

Corollary 3.26 (Confidentiality). *If* $\alpha{:}\star^\top; x{:}(\alpha)\ @\ \bot \vdash e : (\mathsf{bool})\ @\ \bot$ *then for any* $\cdot \vdash \nu_1 : \tau_1$ *and* $\vdash \nu_2 : \tau_2$ *if* $e[\tau_1/\alpha][\nu_2/x]$ *and* $e[\tau_2/\alpha][\nu_2/x]$ *both terminate, they will produce the same value.*

Proof. Then construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e : \forall\alpha{:}\star^\top.(\alpha)\ @\ \bot \xrightarrow{\bot} (\mathsf{bool})\ @\ \bot$ using the appropriate typing rules and then appeal to Theorem B.4 Part 2 to obtain

$$\cdot \vdash \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e \sim_\bot \Lambda\alpha{:}\star^\top.\lambda x{:}(\alpha)\ @\ \bot.e : \forall\alpha{:}\star^\top.(\alpha)\ @\ \bot \xrightarrow{\bot} (\mathsf{bool})\ @\ \bot$$

Kinds

$$\begin{array}{rcl} [\![\star]\!] & \coloneqq & \star^{\top} \\ [\![\kappa_1 \to \kappa_2]\!] & \coloneqq & [\![\kappa_1]\!] \xrightarrow{\top} [\![\kappa_2]\!] \\ \\ [\![\mathsf{bool}]\!] & \coloneqq & (\mathsf{bool}) \mathbin{@} \bot \end{array}$$

Types

$$\begin{array}{rcl} [\![\sigma_1 \times \sigma_2]\!] & \coloneqq & [\![\sigma_1]\!] \times^{\bot} [\![\sigma_2]\!] \\ [\![\sigma_1 \to \sigma_2]\!] & \coloneqq & [\![\sigma_1]\!] \xrightarrow{\bot} [\![\sigma_2]\!] \\ [\![\forall \alpha{:}\kappa.\sigma]\!] & \coloneqq & \forall^{\bot} \alpha{:}[\![\kappa]\!].[\![\sigma]\!] \end{array}$$

Expressions

$$\begin{array}{rcl} [\![\mathbf{true}]\!] & \coloneqq & \mathbf{true} \\ [\![\mathbf{false}]\!] & \coloneqq & \mathbf{false} \\ [\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & \coloneqq & \mathbf{if}\ \langle [\![e_1]\!]\ \mathbf{then}\ [\![e_2]\!]\ \mathbf{else}\ [\![e_3]\!] \\ [\![\langle e_1, e_2 \rangle]\!] & \coloneqq & \langle [\![e_1]\!], [\![e_2]\!] \rangle \\ [\![\mathbf{fst}\, e]\!] & \coloneqq & \mathbf{fst}\, [\![e]\!] \\ [\![\mathbf{snd}\, e]\!] & \coloneqq & \mathbf{snd}\, [\![e]\!] \\ [\![\lambda x{:}\sigma.e]\!] & \coloneqq & \lambda x{:}[\![\sigma]\!].[\![e]\!] \\ [\![e_1 e_2]\!] & \coloneqq & [\![e_1]\!][\![e_2]\!] \\ [\![\Lambda \alpha{:}\kappa.e]\!] & \coloneqq & \Lambda \alpha{:}[\![\kappa]\!].[\![e]\!] \\ [\![e[\tau]]\!] & \coloneqq & [\![e]\!][\![\tau]\!] \end{array}$$

Relations

$$\begin{array}{rcl} [\![\cdot]\!] & \coloneqq & \cdot \\ [\![\eta, \alpha \mapsto R]\!] & \coloneqq & [\![\eta]\!], \alpha \mapsto \begin{cases} \{(v_1, v_2) \mid v_1\, R\, v_2, (\cdot; \cdot \vdash v_1 : \tau_1), (\cdot; \cdot \vdash v_2 : \tau_2)\} & \xi = \bullet \text{ and } \ell = \bot \\ \{(v_1, v_2) \mid (\cdot; \cdot \vdash v_1 : \tau_1), (\cdot; \cdot \vdash v_2 : \tau_2)\} & \xi = \bullet \text{ and } \ell \neq \bot \\ \varnothing & \text{otherwise} \end{cases} \\ & & \text{where } R \in \tau_1 \leftrightarrow \tau_2 \end{array}$$

Figure 20: The encoding for standard parametricity.

By Lemma 3.10 we have that $\tau_1 \approx_{\bot} \tau_2 : \star^{\top}$. Next, by inversion on SLR:ALL and instantiation with the constructor relation, $\tau_1 \approx_{\bot} \tau_2 : \star^{\top}$, and the relation

$$R^{\ell}_{\rho} = \{(v_1, v_2) \mid (\cdot; \cdot \vdash v_1 : (\rho\{\tau_1\}) \mathbin{@} \ell), (\cdot; \cdot \vdash v_2 : (\rho\{\tau_2\}) \mathbin{@} \ell)\},$$

we can conclude that

$$\cdot, \alpha \mapsto R \vdash (\Lambda \alpha{:}\star^{\top}.\lambda x{:}(\alpha) \mathbin{@} \bot.e)[\tau_1] \approx_{\bot} (\Lambda \alpha{:}\star^{\top}.\lambda x{:}(\alpha) \mathbin{@} \bot.e)[\tau_2] : (\alpha) \mathbin{@} \bot \xrightarrow{\bot} (\mathsf{bool}) \mathbin{@} \bot$$

By straightforward application of SLR:VAR we have that

$$\cdot, \alpha \mapsto R \vdash v_1 \sim_{\bot} v_2 : (\alpha) \mathbin{@} \bot$$

so by application of SCLR:TERM, inversion on SCLR:ARR, and instantiation we know

$$\cdot, \alpha \mapsto R \vdash (\Lambda \alpha{:}\star^{\top}.\lambda x{:}(\alpha) \mathbin{@} \bot.e)[\tau_1] v_1 \approx_{\bot} (\Lambda \alpha{:}\star^{\top}.\lambda x{:}(\alpha) \mathbin{@} \bot.e)[\tau_2] v_2 : (\mathsf{bool}) \mathbin{@} \bot$$

Finally, because the relation is closed under reduction we have SLR:ARR and instantiation we have

$$\cdot, \alpha \mapsto R \vdash e[\tau_1/\alpha][v_1/x] \approx_{\bot} e[\tau_2/\alpha][v_2/x] : (\mathsf{bool}) \mathbin{@} \bot$$

from which the desired conclusion can be obtained by simple inversion. □

This corollary states that what is substituted for α and x will not affect the value computed by e. However, it is possible that the choice of α and x could cause e to diverge. What is happening?

Unlike standard parametricity, Theorem 3.24 has an explicit observer. Standard parametricity has an implicit observer that can observe all computations. What makes information-flow techniques work is that some computations are opaque to the observer. Furthermore, the results of these computations are also inaccessible to the observer, making them effectively dead code. However, because the operational semantics we chose to use for $\lambda_{\mathrm{SEC}i}$ is call-by-value, dead code must be executed even though the result is never used.

For example, the following expression is well-typed in $\lambda_{\mathrm{SEC}i}$ with type $\mathsf{bool}^{\bot}$ under the assumption α has kind $\star^{\top}$:

$$\begin{aligned}(\lambda x{:}(\mathsf{bool}) \mathbin{@} \top.\mathbf{true})\ (&\mathbf{typecase}[\gamma.(\mathsf{bool}) \mathbin{@} \top]\alpha\\ &(\mathbf{true})\\ &(\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\mathbf{fix}_{\omega}\ y{:}(\mathsf{bool}) \mathbin{@} \top.y)\\ &(\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\mathbf{false}))\end{aligned}$$

Corollary 3.26 states that if two related constructors are substituted for the free type variable α, in the expression above, that the two resulting expressions will be related. If bool is substituted for α the expression will evaluate to **true**, but if $\mathsf{bool} \to \mathsf{bool}$ is substituted for α then the expression will diverge. Therefore, because one of the expressions diverges, the corollary has not been contradicted.

However, note that the expression

$$(\mathbf{typecase}[\gamma.(\mathsf{bool}) \mathbin{@} \top]\alpha\ (\mathbf{true})(\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\mathbf{fix}_{\omega}\ y{:}(\mathsf{bool}) \mathbin{@} \top.y)(\Lambda\beta{:}\star^{\top}.\Lambda\delta{:}\star^{\top}.\mathbf{false})),$$

is completely dead code because when it does evaluate to a value, it is simply thrown away. If $\lambda_{\mathrm{SEC}i}$ is given a call-by-name operational semantics, the original expression above is operationally equivalent to the expression **true**. We conjecture that all such discrepancies in termination behavior are a result of dead code. Therefore, by using a call-by-name operational semantics, an exact correspondence between standard parametricity and generalized parametricity could be recovered. We believe the only part of the proof for Theorem 3.24 that would need to change is the proof of obliviousness for terms, Lemma 3.14.

3.7. **Applications of generalized parametricity.** A typical corollary of Theorem 3.24 is normally called noninterference; the property that it is possible to substitute values indistinguishable to the present observer and get indistinguishable results.

Corollary 3.27 (Noninterference)**.** *If* $\cdot, x{:}\sigma_1 \vdash e : \sigma_2$ *where* $\mathcal{L}(\sigma_1) \not\sqsupseteq \mathcal{L}(\sigma_2)$ *then for any* $\vdash v_1 : \sigma_1$ *and* $\vdash v_2 : \sigma_1$ *it is the case that if both* $e[v_1/x]$ *and* $e[v_2/x]$ *terminate, they will both produce the same value*

Proof. Proceeds in a similar fashion to Corollary 3.26. □

More importantly, it is also possible to restate the corollaries of standard parametricity proven earlier. The previous subsection stated the revised corollary for confidentiality. The same can be done for integrity:

Corollary 3.28 (Integrity)**.** *If* $\alpha{:}\star^{\top};\cdot \vdash e : (\alpha)\ @\ \bot$ *then* $e[\tau/\alpha]$ *for any* τ *must diverge.*

Proof. First construct a derivation that $\cdot;\cdot \vdash \Lambda\alpha{:}\star^{\top}.e : \forall\alpha^{\top}{:}(\alpha)\ @\ \bot$ using the appropriate typing rules, then appeal to Theorem B.4 Part 2 to obtain to obtain

$$\cdot \vdash \Lambda\alpha{:}\star^{\top}.e \sim_{\bot} \Lambda\alpha{:}\star^{\top}.e : \forall\alpha{:}\star^{\top}.(\alpha)\ @\ \bot$$

Now assume an arbitrary τ. It is straightforward to show that $\tau \approx_{\bot} \tau :\star^{\top}$. By inversion on SLR:ALL and instantiation we can conclude

$$\cdot,\alpha \mapsto \varnothing \vdash (\Lambda\alpha{:}\star^{\top}.e)[\tau] \approx_{\bot} (\Lambda\alpha{:}\star^{\top}.e)[\tau] : (\alpha)\ @\ \bot$$

Because the relation is closed under reduction we have that

$$\cdot,\alpha \mapsto \varnothing \vdash e[\tau/\alpha] \approx_{\bot} e[\tau/\alpha] : (\alpha)\ @\ \bot$$

Furthermore, by inversion either $e[\tau/\alpha] \rightsquigarrow^{*} v$ or $e[\tau/\alpha]\uparrow$. However in the former case that would mean that

$$\cdot,\alpha \mapsto \varnothing \vdash v \sim_{\bot} v : (\alpha)\ @\ \bot$$

which by inversion on SLR:VAR is impossible because there is no v such that $v\varnothing v$. Therefore $e[\tau/\alpha]\uparrow$. □

While these corollaries are very similar in spirit to the ones derived from standard parametricity, it is possible to make much richer and more refined claims because the label lattice expands upon the implicit two level lattice used by parametricity. For example, it is possible to label each abstract data type with a distinct label. This makes it possible to understand which abstract types depend upon each other; the fact that all abstract types in standard parametricity are labeled with $\top$ means that it is not possible to discern their interdependenices. Furthermore, using distinct labels makes it possible to reason about the abstraction properties of each data type separately.

4. Related work

The design of $\lambda_{\mathrm{SEC}i}$ and the proof of generalized parametricity draws heavily upon previous work on type analysis, parametricity, and information flow.

Most information flow systems use a lattice model originating from work by Bell and La Padula [BL75] and Denning [Den76]. The earliest work on static information flow dates back to Denning and Denning [DD77]. Volpano, Smith, and Irvine [VSI96] showed that Denning's work could be formulated as a type system and proved its soundness with respect to noninterference. Heintze and Riecke [HR98] formalized information-flow and integrity in a typed λ-calculus with references, the SLam calculus, and proved a number of soundness and noninterference results. Pottier and Simonet [PS03] have developed an extension of ML, called FlowCaml, and have shown noninterference using an alternative syntactic technique.

Prior to this research, FlowCaml was the only language with type polymorphism and a noninterference proof. However, FlowCaml does not have any mechanisms for TDP and can rely on standard parametricity for types. There was some prior research on noninterference with principal polymorphism by Tse and Zdancewic [TZ04a], and later concurrently with this research they investigated a language with type polymorphism where labels and principals were integrated into the language of types [TZ05]. Furthermore, because their goal was to

support runtime decisions based upon principals, and because principals in their formalization are a special form of type, their language provides a form of runtime type analysis. However, their noninterference theorems focus on how related terms affect computation and do not consider how related types would alter computations.

While research into abstraction properties predates his work, Reynolds [Rey74, Rey83] was the first to show how the parametricity theorem could be used to prove properties about representation independence in the polymorphic λ-calculus. Reynold's proofs were for a polymorphic λ-calculus without higher-kinded types. While we have restricted $\lambda_{\mathrm{SEC}i}$ to disallow polymorphic functions over higher-kinded types, most of the machinery necessary to handle higher-kinded types has been developed because type operators are allowed to abstract over higher-kinded type variables. Girard, in his dissertation [Gir72], did present a form of logical relation for the λ-calculus without higher-kinded types. However, Girard was primarily interested in strong normalization so he only studied unary relations.

Gallier [Gal90] later gave a detailed survey of variations on formalizing what Girard called the method of "Candidats de Reductibilité", including the extensions to higher-kinds. However Gallier focused on strong normalization, so he only studied a unary logical relation. Kučan, in his dissertation [Kuč07], did consider an interpretation for the λ-calculus without higher-kinded types that extended to n-ary relations, but his interpretation is untyped.

Finally, following the publication of our original work on generalized parametricity, Vytiniotis and Weirich [VW07b] developed a detailed formalization of parametricity for the higher-order polymorphic λ-calculus. However, instead of building their formalization around the canonical forms of types, as we have done, they require an additional consistency requirement that their relations must behave the same on β-equivalent types.

Our generalized parametricity result for $\lambda_{\mathrm{SEC}i}$ directly builds upon the methods of Zdancewic [Zda02] and Pitts [Pit05]. Other researchers have noticed the connection between parametricity and noninterference. For example, the work of Tse and Zdancewic [TZ04b] compliments our research by showing how parametricity can be used to prove noninterference. Tse and Zdancewic do so by encoding Abadíet al.'s [ABHR99] dependency core calculus into the polymorphic λ-calculus.

The fact that runtime type analysis (and other forms of ad-hoc polymorphism) breaks parametricity has been long understood, but little has been done to reconcile the two. Leifer et al. [LPSW03] design a system that preserves type abstraction in the presence of (un)marshalling. This is a weaker result because marshalling is merely a single instance of an operation using run-time type analysis. Rossberg [Ros03] and Vytiniotis, Washburn, and Weirich [VWW05] use generative types to hide type information in the presence of run-time analysis, relying on colored-brackets [GMZ00] to provide easy access. However, none of this work has formalized the abstraction properties that their systems provide.

Finally, following the original publication of the work on generalizing parametricity, Vytiniotis and Weirich [VW07a, VW07b] have investigated a more traditional parametricity result for a language with type representations in the style of λ_R. Their work is the most closely related to the research on generalized parametricity.

Their initial work [VW07a] does not handle type operators and type analysis is based upon type representations. There are three significant differences between the language they studied in that work and $\lambda_{\mathrm{SEC}i}$.

The first difference is that they provide a special "top" type representation called $\mathsf{R}_{\mathsf{any}}$. They can use this representation to prove properties that have no correspondence in generalized parametricity as stated here. If $\mathsf{R}_{\mathsf{any}}$ is omitted from their language, the

properties that can be proven in their language are a subset of those that can be derived from generalized parametricity. Using $\mathsf{R}_{\mathsf{any}}$ as an argument to a type analyzing function is a way of forcing functions to behave parametrically at runtime. It is possible to label programs in $\lambda_{\mathrm{SEC}i}$ to force functions to behave parametrically statically, but there is no dynamic analog.

The second difference is that their language allows impredicative rather than predicative type quantification. Therefore, it is possible to write programs and free theorems about polymorphic functions that can be instantiated with polymorphic types themselves. However, because they do not provide a type representation for polymorphic types, there is no interesting interaction between type analysis and polymorphic types just as in $\lambda_{\mathrm{SEC}i}$. The primary obstacle to allowing impredicative type quantification in $\lambda_{\mathrm{SEC}i}$ comes from Typerec. Naïve extensions for analyzing higher-order types at the level of types rather than terms will make type equality undecidable. Extending $\lambda_{\mathrm{SEC}i}$ with a top type would be one way to allow impredicative quantification and avoid this problem.

The third difference is that because type representations are required to perform type analysis, it is possible to completely prevent type analysis by simply not providing a corresponding representation for an abstract type. Using type representations in this fashion is a form of access control. However, we conjecture that nearly all free theorems that can be derived by withholding representations can be emulated in $\lambda_{\mathrm{SEC}i}$ with appropriate labeling. For example, the type $\forall^{\bot}\alpha{:}\star^{\bot}.(\alpha) \mathbin{@} \bot \xrightarrow{\bot} (\alpha) \mathbin{@} \bot$ should have similar inhabitants to the type $\forall\alpha.\mathsf{R}[\alpha] \to \alpha \to \alpha$ in their language. Correspondingly, a function with the type $\forall\alpha.\alpha \to \alpha$ in their language should have similar inhabitants to the type $\forall^{\bot}\alpha{:}\star^{\top}.(\alpha) \mathbin{@} \bot \xrightarrow{\bot} (\alpha) \mathbin{@} \bot$ in $\lambda_{\mathrm{SEC}i}$ (modulo the termination discrepancy described in Section 3.6).

The more recent work by Vytiniotis and Weirich [VW07b] on the language R_{ω} does address type-operators, as described above in our discussion of higher-order parametricity, but does not examine the problems that arise from including type-level type analysis. Again they make use of type representations, but do not include the $\mathsf{R}_{\mathsf{any}}$ type representation. Unlike their prior work, in R_{ω} it is possible to prove interesting results, that have no analog in $\lambda_{\mathrm{SEC}i}$, about the static behavior of programs that use type analysis. Again there are three significant differences between R_{ω} and $\lambda_{\mathrm{SEC}i}$. The first two differences are impredicative polymorphism and the use of type representations for access control, which we discussed earlier. The third significant difference arises because they allow quantification over higher-kinded types. Their central result is a proof of partial correctness for *generic* type-safe cast from the free theorem for its type. In $\lambda_{\mathrm{SEC}i}$, a type-safe cast can be written and has the type

$$\forall^{\bot}\alpha{:}\star^{\bot}.\forall^{\bot}\beta{:}\star^{\bot}.(\alpha) \mathbin{@} \bot \xrightarrow{\bot} ((\beta) \mathbin{@} \bot +^{\bot} \mathbb{1}).$$

In R_{ω} a generic type-safe cast quantifies over a type-operator and has the type

$$\forall\delta{:}\star \to \star.\forall\alpha{:}\star.\forall\beta{:}\star.\mathsf{R}[\alpha] \to \mathsf{R}[\beta] \to \delta\alpha \to (\delta\beta) + \mathbb{1}.$$

Their parametricity theorem can be used to derive that any implementation of this type, if it returns a value of type $\delta\beta$, that value will be identical to the input value with type $\delta\alpha$. That is, a generic type-safe cast cannot subtly modify the input based upon its representation.

However, Vytiniotis and Weirich's [VW07a, VW07b] results are based upon the use of type representations, which, as I described is a form of dynamic access control. Because access control mechanisms cannot capture dependencies, they cannot be used to prove results about confidentiality and integrity independently, like can be done using generalized parametricity. Furthermore, type representations are values; there is no mechanism in their language to reason about the dependencies between abstract types. Finally, the use of an explicit lattice

in $\lambda_{\mathrm{SEC}i}$ allows for cleanly reasoning about the confidentiality and integrity of several ADTs simultaneously, along with the relationships between them.

Lastly, in his dissertation, Washburn [Was07] developed a language called InforML based upon the ideas found in $\lambda_{\mathrm{SEC}i}$, and showed how it might be used in practical programming. InforML offers nearly all the features found in a typical ML-like language: algebraic data types, pattern matching, mutable references, and a simple module system. Unlike $\lambda_{\mathrm{SEC}i}$, InforML has no distinction between constructors and types which simplifies some aspects of programming, yet complicates others by requiring a more complex language of labels. At present there is no formal model of InforML, so type safety and generalized parametricity are only conjectured.

5. Conclusion

With $\lambda_{\mathrm{SEC}i}$, we address the conflict between run-time type analysis and enforceable data abstractions. By labeling their type abstractions, software developers can easily observe dependencies.

One problem we mentioned in Section 3.5 is that it is very difficult to define families of relations for use with the generalized parametricity theorem that are not parametric in the constructor context. If Typerec were removed from $\lambda_{\mathrm{SEC}i}$, this problem goes away. However, Typerec can be very useful, so it would be worthwhile finding a way to resolve the difficulty.

This problem seems very similar to the problem of proving contextual equivalence directly. For contextual equivalence the problem is proving by induction over all possible program contexts that two expressions will behave the same when placed in those contexts. Usually the solution is to develop some other property for relating two expressions, and show that property is equivalent to contextual equivalence. For example, CIU-equivalence, as defined by Pitts [Pit05], is one such property.

The problem with defining families of relations for generalized parametricity is the need to develop a function from any possible context to a relation. Constructing such a function is isomorphic to constructing an inductive proof over contexts, so perhaps similar ideas to those used to prove contextual equivalence indirectly could be used to indirectly specify functions on constructor contexts.

Another angle on the problem with constructor contexts is that there is a mismatch between how we have extended parametricity to non-standard types and how parametricity has typically been extended to higher-kinds. Generalized parametricity quantifies over functions from labels and constructor contexts to relations. For example, we use the following notation for relations in Section 3.5:

$$\mathsf{R}^{\ell}_{\xi} \in ((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow ((\xi\{\tau_2\}) \mathbin{@} \ell),$$

The essence of R can be understood better type-theoretically as an entity with the type

$$\Pi\ell.\Pi\xi.((\xi\{\tau_1\}) \mathbin{@} \ell) \leftrightarrow ((\xi\{\tau_2\}) \mathbin{@} \ell),$$

where $\cdot \leftrightarrow \cdot$ can be understood as the "type constructor" of relations.

However, when standard parametricity is extended to higher kinds [VW07b], such as $\star \to \star$, functions from relations to relations are quantified over. For example, if ψ is a quantified type variable with kind $\star \to \star$, it would be necessary to quantify over an entity with the type

$$[\![\psi]\!] : \Pi\alpha{:}\star.\Pi\beta{:}\star.(\alpha \leftrightarrow \beta) \to (\tau_1\alpha \leftrightarrow \tau_2\beta),$$

that is, a function from an arbitrary pair of types α and β, and a relation between them, $\alpha \leftrightarrow \beta$, to the a relation $\tau_1\alpha \leftrightarrow \tau_1\beta$, for some $\tau_1 : \star\rightarrow\star$ and $\tau_2 : \star\rightarrow\star$. Furthermore, the type of this entity is completely derived from ψ's kind and the choice of τ_1 and τ_2:

$$\begin{array}{rcl} [\![\star]\!](\tau_1,\tau_2) & \triangleq & \tau_{\ 1} \leftrightarrow \tau_2 \\ [\![\kappa_1 \rightarrow \kappa_2]\!](\tau_1,\tau_2) & \triangleq & \Pi\alpha{:}\kappa_1.\Pi\beta{:}\kappa_1.[\![\kappa_1]\!](\alpha,\beta) \rightarrow [\![\kappa_2]\!](\tau_1\alpha,\tau_2\beta) \end{array}$$

Therefore, it seems plausible that a similar approach could be used to express more interesting relationships between abstract data types in generalized parametricity. Such a solution would give R a type something like

$$\Pi\ell.\Pi\xi.[\![\xi]\!] \rightarrow ((\xi\{\tau_1\}) \ @ \ \ell) \leftrightarrow ((\xi\{\tau_2\}) \ @ \ \ell),$$

where $[\![\xi]\!]$ is some function on relations. In the case of the constructor context hole, $\bullet$, $[\![\bullet]\!]$ should most likely be the identity function on relations. Determining the definition of $[\![\cdot]\!]$ on more complex constructor contexts, and whether this is even the most suitable formulation, will require further study.

We conjecture the problem with the expressive power of constructor contexts will also arise in attempts to prove parametricity like properties for other languages with expressive type systems. Such languages would include, for example, those with dependent or indexed types where type equivalence is non-parametric with respect to abstract indices or values. Therefore, we expect that having a better understanding of how to deal with constructor contexts for generalized parametricity will have much wider applicability.

Acknowledgements

This paper benefitted greatly from conversations with Benjamin Pierce, Val Tannen, and Stephen Tse, David Walker, Steve Zdancewic. We also appreciate the insightful comments by anonymous reviewers on earlier revisions of this work. Finally, we especially thank Derek Dreyer for pointing out some subtle errors in our original proof. This work was supported by NSF grant 0347289, Career: Type-Directed Programming in Object-Oriented Languages.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages*, pages 147–160, San Antonio, TX, January 1999.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Gal90] Jean H. Gallier. On Girard's "Candidats de Reductibilité". *Logic and Computer Science*, 31:123–203, 1990.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, San Francisco, California, January 1995. Acm Press.

[HR98] Nevin C. Heintz and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, San Diego, CA, January 1998. Acm Press.

[HWC01] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.

[Kuč07] Jakov Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS transform and "Free Theorems"*. PhD thesis, Massachusetts Institute of Technology, 2007.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 87–98, Uppsala, Sweden, 2003. Acm Press.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. Published as CMU Tech Report number CMU-CS-95-226.

[Pit05] Andrew Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 245–289. MIT Press, 2005.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, pages 513–523, Paris, France, September 1983. Elsevier Science Publishers.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In Dale Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003. Acm Press.

[TZ04a] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*, Oakland, CA, May 2004.

[TZ04b] Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proc. of the 9th ACM International Conference on Functional Programming*, Snowbird, Utah, September 2004.

[TZ05] Stephen Tse and Steve Zdancewic. Designing a security-typed language with certificate-based declassification. In Shmuel Sagiv, editor, *Proceedings of the 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 279–294, Edinburgh, UK, 2005. Springer-Verlag.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[VW07a] Dimitrios Vytiniotis and Stephanie Weirich. Free theorems and runtime type representations. In *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics*, volume 173, pages 357–373, New Orleans, LA, April 2007. Elsevier Science Publishers.

[VW07b] Dimitrios Vytiniotis and Stephanie Weirich. Type-safe cast does no harm. Draft available from `http://www.cis.upenn.edu/~sweirich/papers/popl08-parametricity.pdf`., July 2007.

[VWW05] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation*, Longbeach, California, January 2005.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, London, UK, September 1989. Acm Press.

[Was07] Geoffrey Washburn. *Principia Narcissus: How to Avoid Being Caught by Your Reflection*. PhD thesis, University of Pennsylvania, 2007.

[Wei00] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of theFifth International Conference on Functional Programming (ICFP)*, pages 58–67, Montreal, September 2000.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WW05] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information flow. In *The Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71, Chicago, IL, June 2005. IEEE Computer Society, IEEE Computer Society Press.

[Zda02] Stephan Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

Appendix A. $\lambda_{\mathrm{SEC}i}$ soundness lemmas

Lemma A.1 (Inversion on sub-kinding)**.**

(1) *If* $\star^{\ell} \leq \kappa$ *then* $\kappa = \star^{\ell'}$ *where* $\ell \sqsubseteq \ell'$.
(2) *If* $\kappa_1 \xrightarrow{\ell} \kappa_2 \leq \kappa$ *then* $\kappa = \kappa_3 \xrightarrow{\ell'} \kappa_4$ *where* $\kappa_3 \leq \kappa_1$ *and* $\kappa_2 \leq \kappa_4$ *and* $\ell \sqsubseteq \ell'$.

Proof. Straightforward induction over the structure of the sub-kinding derivation. □

Lemma A.2 (Inversion for constructor well-formedness)**.**

(1) *If* $\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell}$ *then* $\Delta \vdash \tau_1 : \star^{\ell_1}$ *and* $\Delta \vdash \tau_2 : \star^{\ell_2}$ *and* $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$.
(2) *If* $\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell}$ *then* $\Delta \vdash \tau_1 : \star^{\ell_1}$ *and* $\Delta \vdash \tau_2 : \star^{\ell_2}$ *and* $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$.
(3) *If* $\Delta \vdash \tau_1 \tau_2 : \kappa$ *then* $\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2$ *and* $\Delta \vdash \tau_2 : \kappa_1$ *and* $\kappa_2 \sqcup \ell \leq \kappa$.
(4) *If* $\Delta \vdash \lambda\alpha{:}\kappa.\tau : \kappa_1 \xrightarrow{\ell} \kappa_2$ *then* $\Delta, \alpha{:}\kappa \vdash \tau : \kappa_3$ *and* $\kappa_1 \leq \kappa$ *and* $\kappa_3 \leq \kappa_2$.
(5) *If* $\Delta \vdash \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \tau_{\times} : \kappa$ *then* $\Delta \vdash \tau : \star^{\ell}$ *and* $\Delta \vdash \tau_{\mathsf{bool}} : \kappa'$ *and* $\Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa'$ *and* $\Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa' \xrightarrow{\ell'} \kappa'$ *where* $\ell' = \mathcal{L}(\kappa')$ *and* $\kappa' \leq \kappa$.

Proof. By induction over the structure of the well-formedness derivation, making use of Lemma A.1. □

Lemma A.3 (Weak-head reduction equivalence)**.**

(1) *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow \tau'$ *then* $\Delta \vdash \tau = \tau' : \kappa$.
(2) *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow^* \tau'$ *then* $\Delta \vdash \tau = \tau' : \kappa$.
(3) *If* $\Delta^{\star} \vdash \sigma$ *and* $\sigma \rightsquigarrow \sigma'$ *then* $\Delta^{\star} \vdash \sigma = \sigma'$.
(4) *If* $\Delta^{\star} \vdash \sigma$ *and* $\sigma \rightsquigarrow^* \sigma'$ *then* $\Delta^{\star} \vdash \sigma = \sigma'$.

Proof. Part 1 follows from straightforward induction over the structure of $\tau \rightsquigarrow \tau'$ and use of Lemma A.2. Part 2 follows from Part 1 and induction on the number of reduction steps. Part 3 follows from straightforward induction over the structure of $\sigma \rightsquigarrow \sigma'$ using Part 1. Finally, Part 4 follows from Part 3 and induction on the number of reduction steps. □

Lemma A.4 (Inversion for type well-formedness)**.**
If $\Delta^{\star} \vdash (\tau)\ @\ \ell$ *then* $\Delta^{\star} \vdash \tau : \star^{\ell'}$.

Proof. Proof by induction over the structure of $\Delta^{\star} \vdash (\tau)\ @\ \ell$. □

Lemma A.5 (Inversion for subtyping)**.**

(1) *If* $\Delta^{\star} \vdash \sigma_1 \xrightarrow{\ell_1} \sigma_2 \leq \sigma$ *then* $\Delta^{\star} \vdash \sigma \leq \sigma_3 \xrightarrow{\ell_2} \sigma_4$ *and* $\Delta^{\star} \vdash \sigma_3 \leq \sigma_1$ *and* $\Delta^{\star} \vdash \sigma_2 \leq \sigma_4$ *and* $\ell_1 \sqsubseteq \ell_2$.
(2) *If* $\Delta^{\star} \vdash \sigma_1 \times^{\ell_1} \sigma_2 \leq \sigma$ *then* $\Delta^{\star} \vdash \sigma \leq \sigma_3 \times^{\ell_2} \sigma_4$ *and* $\Delta^{\star} \vdash \sigma_1 \leq \sigma_3$ *and* $\Delta^{\star} \vdash \sigma_2 \leq \sigma_4$ *and* $\ell_1 \sqsubseteq \ell_2$.
(3) *If* $\Delta^{\star} \vdash \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1 \leq \sigma_2$ *then* $\Delta^{\star} \vdash \sigma_2 \leq \forall^{\ell_3}\alpha{:}\star^{\ell_4}.\sigma_3$ *and* $\Delta^{\star}, \alpha{:}\star^{\ell_4} \vdash \sigma_3 \leq \sigma_1$ *and* $\ell_1 \sqsubseteq \ell_3$ *and* $\ell_4 \sqsubseteq \ell_2$.

Proof. By straightforward induction over the structure of the subtyping derivation. □

Lemma A.6 (Inversion for typing)**.**

(1) *If* $\Delta^\star;\Gamma \vdash \lambda x{:}\sigma_1.e : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_2 \xrightarrow{\ell} \sigma_3$ *and* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_4$ *where* $\Delta^\star \vdash \sigma_2 \leq \sigma_1$ *and* $\Delta^\star \vdash \sigma_4 \leq \sigma_3$.
(2) *If* $\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^\ell.e : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma_1$ *and* $\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma_2$ *where* $\Delta^\star,\alpha{:}\star^{\ell_2} \vdash \sigma_2 \leq \sigma_1$ *and* $\ell_2 \sqsubseteq \ell$.
(3) *If* $\Delta^\star;\Gamma \vdash \mathbf{fix}\ x{:}\sigma_1.e : \sigma_2$ *then* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_1$ *where* $\Delta^\star \vdash \sigma_1 \leq \sigma_2$.
(4) *If* $\Delta^\star;\Gamma \vdash \langle e_1,e_2\rangle : \sigma$ *then* $\Delta^\star \vdash \sigma \leq \sigma_1 \times^\ell \sigma_2$ *and* $\Delta^\star;\Gamma \vdash e_1 : \sigma_3$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma_4$ *where* $\Delta^\star \vdash \sigma_3 \leq \sigma_1$ *and* $\Delta^\star \vdash \sigma_4 \leq \sigma_2$.
(5) *If* $\Delta^\star;\Gamma \vdash \mathbf{fst}\, e : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2$ *where* $\Delta^\star \vdash \sigma_1 \sqcup \ell \leq \sigma$.
(6) *If* $\Delta^\star;\Gamma \vdash \mathbf{snd}\, e : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \sigma_1 \times^\ell \sigma_2$ *where* $\Delta^\star \vdash \sigma_2 \sqcup \ell \leq \sigma$.
(7) *If* $\Delta^\star;\Gamma \vdash e_1e_2 : \sigma_1$ *then* $\Delta^\star;\Gamma \vdash e_1 : \sigma_2 \xrightarrow{\ell} \sigma_3$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma_2$ *and* $\Delta^\star \vdash \sigma_3 \sqcup \ell \leq \sigma_1$.
(8) *If* $\Delta^\star;\Gamma \vdash e[\tau] : \sigma$ *then* $\Delta^\star;\Gamma \vdash e : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma'$ *and* $\Delta^\star \vdash \tau : \star^{\ell_2}$ *and* $\Delta^\star \vdash \sigma'[\tau/\alpha] \sqcup \ell_1 \leq \sigma$.
(9) *If* $\Delta^\star;\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma$ *then* $\Delta^\star;\Gamma \vdash e_1 : (\mathsf{bool})\ @\ \ell$ *and* $\Delta^\star;\Gamma \vdash e_2 : \sigma'$ *and* $\Delta^\star;\Gamma \vdash e_3 : \sigma'$ *where* $\Delta^\star \vdash \sigma' \sqcup \ell \leq \sigma$.
(10) *If* $\Delta^\star;\Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\rightarrow}\ e_{\times} : \sigma'$ *then*
$\Delta^\star \vdash \tau : \star^\ell$ *and*
$\Delta^\star,\gamma{:}\star^\ell \vdash \sigma$ *and*
$\Delta^\star;\Gamma \vdash e_{\mathsf{bool}} : \sigma[\mathsf{bool}/\gamma]$ *and*
$\Delta^\star;\Gamma \vdash e_{\rightarrow} : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \rightarrow \beta/\gamma]$ *and*
$\Delta^\star;\Gamma \vdash e_{\times} : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \times \beta/\gamma]$ *where*
$\ell' = \mathcal{L}(\sigma[\tau/\gamma])$ *and*
$\ell \sqsubseteq \ell'$ *and*
$\Delta^\star \vdash \sigma[\tau/\gamma] \leq \sigma'$.

Proof. By straightforward induction on the structure of the typing derivation with uses of Lemma A.5. □

Lemma A.7 (Substitution for constructors)**.** *If* $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 : \kappa_2$ *and* $\Delta \vdash \tau_2 : \kappa_1$ *then* $\Delta \vdash \tau_1[\tau_2/\alpha] : \kappa_2$.

Proof. By straightforward induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 : \kappa_2$. □

Lemma A.8 (Substitution for equivalence)**.** *If* $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2$ *and* $\Delta \vdash \tau : \kappa_1$ *then* $\Delta \vdash \tau_1[\tau/\alpha] = \tau_2[\tau/\alpha] : \kappa_2$.

Proof. By straightforward induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2$, making use of Lemma A.7. □

Lemma A.9 (Substitution for types)**.**

(1) *If* $\Delta^\star,\alpha{:}\star^\ell \vdash \sigma_1 \leq \sigma_2$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star \vdash \sigma_1[\tau/\alpha] \leq \sigma_2[\tau/\alpha]$.
(2) *If* $\Delta^\star,\alpha{:}\star^\ell \vdash \sigma$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star \vdash \sigma[\tau/\alpha]$.

Proof. By mutual induction over the structure of $\Delta,\alpha{:}\star^\ell \vdash \sigma_1 \leq \sigma_2$ and $\Delta,\alpha{:}\star^\ell \vdash \sigma$, using Lemmas A.7 and A.8. □

Lemma A.10 (Substitution commutes with equivalence)**.**

(1) *If* $\Delta \vdash \tau_1 = \tau_2 : \kappa_1$ *and* $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$ *then* $\Delta \vdash \tau[\tau_1/\alpha] = \tau[\tau_2/\alpha] : \kappa_2$.
(2) *If* $\Delta \vdash \tau_1 = \tau_2 : \star^\ell$ *and* $\Delta,\alpha{:}\star^\ell \vdash \sigma$ *then* $\Delta \vdash \sigma[\tau_1/\alpha] \leq \sigma[\tau_2/\alpha]$ *and* $\Delta \vdash \sigma[\tau_2/\alpha] \leq \sigma[\tau_1/\alpha]$.

Proof. Part 1 follows from induction over the structure of $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$. Part 2 follows from induction over the structure of $\Delta,\alpha{:}\star^\ell \vdash \sigma$ making use of Part 1. □

Lemma A.11 (Substitution for terms)**.**

(1) *If* $\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma$ *and* $\Delta^\star \vdash \tau : \star^\ell$ *then* $\Delta^\star;\Gamma \vdash e[\tau/\alpha] : \sigma[\tau/\alpha]$.
(2) *If* $\Delta^\star;\Gamma,x{:}\sigma_1 \vdash e : \sigma_2$ *and* $\Delta^\star;\Gamma \vdash e' : \sigma_1$ *then* $\Delta^\star;\Gamma \vdash e[e'/x] : \sigma_2$.

Proof. By straightforward induction over the typing derivations, using Lemmas A.7 and A.9. □

Lemma A.12 (Subject reduction for constructors and types)**.**

(1) *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow \tau'$ *then* $\Delta \vdash \tau' : \kappa$.
(2) *If* $\Delta \vdash \tau : \kappa$ *and* $\tau \rightsquigarrow^* \tau'$ *then* $\Delta \vdash \tau' : \kappa$.
(3) *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow \sigma'$ *then* $\Delta^\star \vdash \sigma'$.
(4) *If* $\Delta^\star \vdash \sigma$ *and* $\sigma \rightsquigarrow^* \sigma'$ *then* $\Delta^\star \vdash \sigma'$.

Proof. Part 1 follows by induction over the structure of $\tau \rightsquigarrow \tau'$ making use of Lemmas A.2 and A.7. Part 2 is a direct corollary of Part 1. Part 3 follows by induction over the structure of $\sigma \rightsquigarrow \sigma'$ making use of Lemma A.4 and Part 1. Part 4 is a direct corollary of Part 3. □

Lemma A.13 (Weak head reduction terminates)**.**

(1) *If* $\cdot \vdash \tau : \kappa$ *then* $\tau \rightsquigarrow^* \nu$.
(2) *If* $\Delta^\star \vdash \sigma$ *then* $\sigma \rightsquigarrow^* \zeta$.

Proof. Follows from a standard logical relations proof that we omit here. See Morrisett's thesis [Mor95]. □

Lemma A.14 (Canonical forms for constructors)**.** *If* $\cdot \vdash \nu : \kappa$

(1) $\kappa = \star^\ell$ *then* $\nu = \mathsf{bool}$ *or* $\nu = \tau_1 \to \tau_2$ *or* $\nu = \tau_1 \times \tau_2$.
(2) $\kappa = \kappa_1 \xrightarrow{\ell} \kappa_2$ *then* $\nu = \lambda\alpha{:}\kappa_3.\tau$ *where* $\kappa_1 \leq \kappa_3$.

Proof. By straightforward induction over the structure of $\Delta \vdash \nu : \kappa$. □

Lemma A.15 (Canonical forms for terms)**.** *If* $\cdot;\cdot \vdash \nu : \sigma$

(1) $\sigma = \mathsf{bool}$ *then* $\nu = \mathbf{true}$ *or* $\nu = \mathbf{false}$.
(2) $\sigma = \sigma_1 \xrightarrow{\ell'} \sigma_2$ *then* $\nu = \lambda x{:}\sigma_3.e$ *where* $\Delta^\star \vdash \sigma_1 \leq \sigma_3$.
(3) $\sigma = \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma'$ *then* $\nu = \Lambda\alpha{:}\star^{\ell_3}.e$ *where* $\ell_1 \sqsubseteq \ell_3$.
(4) $\sigma = \sigma_1 \times^\ell \sigma_2$ *then* $\nu = \langle \nu_1, \nu_2 \rangle$.

Proof. By straightforward induction over the structure of $\cdot;\cdot \vdash \nu : \sigma$. □

Appendix B. $\lambda_{\mathrm{SEC}i}$ generalized parametricity

Lemma B.1 (Logical relations are closed under reduction)**.**

(1) $\tau_1 \approx_{\ell_o} \tau_2 : \kappa$ *iff* $\tau_1 \rightsquigarrow^* \tau_1'$ *and* $\tau_2 \rightsquigarrow^* \tau_2'$ *and* $\tau_1' \approx_{\ell_o} \tau_2' : \kappa$.
(2) $\eta \vdash e_1 \approx_{\ell_o} e_2 : \sigma$ *iff* $e_1 \rightsquigarrow^* e_1'$ *and* $e_2 \rightsquigarrow^* e_2'$ *and* $\sigma \rightsquigarrow^* \sigma'$ *and* $\eta \vdash e_1' \approx_{\ell_o} e_2' : \sigma'$.

Proof. Follows from straightforward inversion upon the logical relations and from the properties of reduction. □

Lemma B.2 (Inversion for subtyping on normal types)**.**

(1) *If* $\Delta^\star \vdash (\rho\{\alpha\})\ @\ \ell_1 \le \zeta$ *then* $\zeta = (\rho\{\alpha\})\ @\ \ell_2$ *where* $\ell_1 \sqsubseteq \ell_2$.
(2) *If* $\Delta^\star \vdash (\mathsf{bool})\ @\ \ell_1 \le \zeta$ *then* $\zeta = (\mathsf{bool})\ @\ \ell_2$ *where* $\ell_1 \sqsubseteq \ell_2$.

Proof. By straightforward induction over the structure of the subtyping derivations. □

Lemma B.3 (Constructor relation closed under $\mathsf{Typerec}$)**.** *If* $\tau \approx_{\ell_o} \tau' : \star^\ell$ *and*

- $\tau_{\mathsf{bool}} \approx_{\ell_o} \tau'_{\mathsf{bool}} : \kappa$ *and*
- $\tau_\to \approx_{\ell_o} \tau'_\to : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ *and*
- $\tau_\times \approx_{\ell_o} \tau'_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$.

where $\ell' = \mathcal{L}(\kappa)$ *then* $\mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_\to\ \tau_\times \approx_{\ell_o} \mathsf{Typerec}\ \tau'\ \tau'_{\mathsf{bool}}\ \tau'_\to\ \tau'_\times : \kappa$.

Proof. Straightforward induction over the structure of $\tau \approx_{\ell_1} \tau' : \star^\ell$ making use of Lemma 3.10. □

Theorem B.4 (Substitution)**.**

(1) *If* $\Delta \vdash \tau : \kappa$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa$.
(2) *If* $\Delta^\star; \Gamma \vdash e : \sigma$ *and* $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ *and* $\delta_1, \delta_2 \vdash \eta : \Delta^\star$ *and* $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ *then* $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma$.

Proof. Part 1 follows by induction over the structure of $\Delta \vdash \tau : \kappa$.

Case:

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsc{wfc:var}$$

- Immediate by inversion upon $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$.

Case:

$$\frac{}{\Delta \vdash \mathsf{bool} : \star^\bot}\ \textsc{wfc:bool}$$

- By the definition of substitution $\delta_i(\mathsf{bool}) = \mathsf{bool}$, and $\mathsf{bool} \rightsquigarrow^* \mathsf{bool}$ by TRC:REFL, therefore $\delta_i(\mathsf{bool}) \rightsquigarrow^* \delta_i(\mathsf{bool})$.
- $\bot \sqsubseteq \ell_o$ for any ℓ_o so it follows trivially from TSLR:TYPE-BOOL that $\mathsf{bool} \sim_{\ell_o} \mathsf{bool} : \star^\bot$.
- By TSCLR:BASE on $\mathsf{bool} \sim_{\ell_o} \mathsf{bool} : \star^\bot$ and $\delta_i(\mathsf{bool}) \rightsquigarrow^* \delta_i(\mathsf{bool})$ we can conclude that $\mathsf{bool} \approx_{\ell_o} \mathsf{bool} : \star^\bot$.

Case:

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsc{wfc:arr}$$

- By the definition of substitution $\delta_i(\tau_1 \to \tau_2) = \delta_i(\tau_1) \to \delta_{\,i}(\tau_2)$ and $\delta_i(\tau_1) \to \delta_{\,i}(\tau_2) \rightsquigarrow^* \delta_i(\tau_1) \to \delta_{\,i}(\tau_2)$, by TRC:REFL, therefore $\delta_i(\tau_1 \to \tau_2) \rightsquigarrow^* \delta_i(\tau_1 \to \tau_2)$.
- Lattice joins and order are decidable, so either $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$ or $\ell_1 \sqcup \ell_2 \not\sqsubseteq \ell_o$.

  Sub-Case: $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$.

  – Appeal to the induction hypothesis on $\Delta \vdash \tau_1 : \star^{\ell_1}$ and $\Delta \vdash \tau_2 : \star^{\ell_2}$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ yielding $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \star^{\ell_1}$ and $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \star^{\ell_2}$.
  – Using TSLR:TYPE-ARR on these along with $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$ (by reflexivity) and $\ell_1 \sqcup \ell_2 \sqsubseteq \ell_o$ yields

  $$\delta_1(\tau_1) \to \delta_{\,1}(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \to \delta_{\,2}(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

  Sub-Case: $\ell_1 \sqcup \ell_2 \not\sqsubseteq \ell_o$

– It follows trivially from TSLR:TYPE-OPAQ that

$$\delta_1(\tau_1) \rightarrow\!\delta_{\,1}(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \rightarrow\!\delta_{\,2}(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

- Using TSCLR:BASE on $\delta_i(\tau_1) \rightarrow\!\delta_{\,i}(\tau_2) \rightsquigarrow^* \delta_i(\tau_1) \rightarrow\!\delta_{\,i}(\tau_2)$ and

$$\delta_1(\tau_1) \rightarrow\!\delta_{\,1}(\tau_2) \sim_{\ell_o} \delta_2(\tau_1) \rightarrow\!\delta_{\,2}(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

gives us

$$\delta_1(\tau_1) \rightarrow\!\delta_{\,1}(\tau_2) \approx_{\ell_o} \delta_2(\tau_1) \rightarrow\!\delta_{\,2}(\tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

which by the equality described above, is the same as

$$\delta_1(\tau_1 \to \tau_2) \approx_{\ell_o} \delta_2(\tau_1 \to \tau_2) : \star^{\ell_1 \sqcup \ell_2}$$

Case: The case for WFC:PROD is symmetric to the case for WFC:ARR.

Case:

$$\frac{\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa_1 \xrightarrow{\perp} \kappa_2}\ \textsc{wfc:abs}$$

- By the definition of substitution $\delta_i(\lambda\alpha{:}\kappa_1.\tau) = \lambda\alpha{:}\kappa_1.\delta_i(\tau)$ and by TRC:REFL we know
  $\lambda\alpha{:}\kappa_1.\delta_i(\tau) \rightsquigarrow^* \lambda\alpha{:}\kappa_1.\delta_i(\tau)$, therefore $\delta_i(\lambda\alpha{:}\kappa_1.\tau) \rightsquigarrow^* \delta_i(\lambda\alpha{:}\kappa_1.\tau)$.
- Assume $\tau_1 \approx_{\ell_o} \tau_2 : \kappa_1$. Therefore, $\delta_1,[\tau_1/\alpha] \approx_{\ell_o} \delta_2,[\tau_2/\alpha] : \Delta,\alpha{:}\kappa_1$ by Definition 3.12 and inversion upon $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$.
- Appealing to the induction hypothesis on $\Delta,\alpha{:}\kappa_1 \vdash \tau : \kappa_2$ with $\delta_1,[\tau_1/\alpha] \approx_{\ell_o} \delta_2,[\tau_2/\alpha] : \Delta,\alpha{:}\kappa_1$ we have that

$$(\delta_1,[\tau_1/\alpha])(\tau) \approx_{\ell_o} (\delta_2,[\tau_2/\alpha])(\tau) : \kappa_2$$

- By Lemma B.1 we know that this is the same as

$$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2$$

Furthermore by Lemma 3.18 Part 3.11 on $\kappa_2 \sqsubseteq \kappa_2 \sqcup \perp$ and

$$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2$$

we know that

$$(\lambda\alpha{:}\kappa_1.\delta_1(\tau))\tau_1 \approx_{\ell_o} (\lambda\alpha{:}\kappa_1.\delta_2(\tau))\tau_2 : \kappa_2 \sqcup \perp$$

- Consequently, discharging our assumption we have that

$$\lambda\alpha{:}\kappa_1.\delta_1(\tau) \sim_{\ell_o} \lambda\alpha{:}\kappa_1.\delta_2(\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

Use of TSCLR:BASE on this and $\lambda\alpha{:}\kappa_1.\delta_i(\tau) \rightsquigarrow^* \lambda\alpha{:}\kappa_1.\delta_i(\tau)$ yields

$$\lambda\alpha{:}\kappa_1.\delta_1(\tau) \approx_{\ell_o} \lambda\alpha{:}\kappa_1.\delta_2(\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

By the above identity, this is the same as

$$\delta_1(\lambda\alpha{:}\kappa_1.\tau) \approx_{\ell_o} \delta_2(\lambda\alpha{:}\kappa_1.\tau) : \kappa_1 \xrightarrow{\perp} \kappa_2$$

Case:

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\tau_2 : \kappa_2 \sqcup \ell}\ \textsc{wfc:app}$$

- Appealing to the induction hypothesis on $\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2$ and $\Delta \vdash \tau_2 : \kappa_1$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ gives us $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \kappa_1 \xrightarrow{\ell} \kappa_2$ and $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \kappa_1$.

- By inversion upon $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_1) : \kappa_1 \xrightarrow{\ell} \kappa_2$ we have that $\delta_i(\tau_1) \rightsquigarrow^* \nu_i$ and $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell} \kappa_2$. By further inversion upon $\nu_1 \sim_{\ell_o} \nu_2 : \kappa_1 \xrightarrow{\ell} \kappa_2$ we know that
$$\forall(\tau_1' \approx_{\ell_o} \tau_2' : \kappa_1).\nu_1\tau_1' \approx_{\ell_o} \nu_2\tau_2' : \kappa_2 \sqcup \ell$$
- Instantiating this with $\delta_1(\tau_2) \approx_{\ell_o} \delta_2(\tau_2) : \kappa_1$ gives us
$$\nu_1(\delta_1(\tau_2)) \approx_{\ell_o} \nu_2(\delta_2(\tau_2)) : \kappa_2 \sqcup \ell$$
By inversion on this we get that $\nu_i(\delta_i(\tau_2)) \rightsquigarrow^* \nu_i'$ and $\nu_1' \sim_{\ell_o} \nu_2' : \kappa_2 \sqcup \ell_2$.
- Given $\delta_i(\tau_1) \rightsquigarrow^* \nu_i$ and $\nu_i(\delta_i(\tau_2)) \rightsquigarrow^* \nu_i'$ we know that $\delta_i(\tau_1)\delta_i(\tau_2) \rightsquigarrow^* \nu_i'$. As $\delta_i(\tau_1)\delta_i(\tau_2) = \delta_i(\tau_1\tau_2)$, this is the same as $\delta_i(\tau_1\tau_2) \rightsquigarrow^* \nu_i'$.
- We have what we need and can conclude $\delta_1(\tau_1\tau_2) \approx_{\ell_o} \delta_2(\tau_1\tau_2) : \kappa_2 \sqcup \ell$ by TSCLR:BASE.

Case:

$$\frac{\begin{array}{ll}\Delta \vdash \tau : \star^\ell & \Delta \vdash \tau_\to : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\mathsf{bool}} : \kappa & \Delta \vdash \tau_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa\end{array} \quad \text{where } \ell' = \mathcal{L}(\kappa) \text{ and } \ell \sqsubseteq \ell'}{\Delta \vdash \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_\to\ \tau_\times : \kappa}\ \textsc{wfc:trec}$$

- By appealing to the induction hypothesis on $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ and
  - $\Delta \vdash \tau : \star^\ell$ and
  - $\Delta \vdash \tau_{\mathsf{bool}} : \kappa$ and
  - $\Delta \vdash \tau_\to : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ and
  - $\Delta \vdash \tau_\times : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$

  yields
  - $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \star^\ell$ and
  - $\delta_1(\tau_{\mathsf{bool}}) \approx_{\ell_o} \delta_2(\tau_{\mathsf{bool}}) : \kappa$ and
  - $\delta_1(\tau_\to) \approx_{\ell_o} \delta_2(\tau_\to) : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$ and
  - $\delta_1(\tau_\times) \approx_{\ell_o} \delta_2(\tau_\times) : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa$

- Using Lemma B.3 on these facts gives us that

$$\mathsf{Typerec}\ \delta_1(\tau)\ \delta_1(\tau_{\mathsf{bool}})\ \delta_1(\tau_\to)\ \delta_1(\tau_\times) \approx_{\ell_o} \mathsf{Typerec}\ \delta_2(\tau)\ \delta_2(\tau_{\mathsf{bool}})\ \delta_2(\tau_\to)\ \delta_2(\tau_\times) : \kappa$$

By the definition of substitution this is identical to

$$\delta_1(\mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_\to\ \tau_\times) \approx_{\ell_o} \delta_2(\mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_\to\ \tau_\times) : \kappa$$

Case:

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \le \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \textsc{wfc:sub}$$

- First, appeal to the induction hypothesis on $\Delta \vdash \tau : \kappa_1$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta$ to conclude $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa_1$.
- Using Lemma 3.18 Part 3.11. on this with $\kappa_1 \sqsubseteq \kappa_2$ we can conclude the desired result, $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \kappa_2$.

Part 2 follows by induction over the structure/heights of typing derivations.

Cases: The cases for WFT:TRUE and WFT:FALSE are analogous to that for WFC:BOOL.

Case:

$$\frac{\Delta^\star \vdash \Gamma \qquad x : \sigma \in \Gamma}{\Delta^\star; \Gamma \vdash x : \sigma}\ \textsc{wft:var}$$

- Follows immediately by inversion upon $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$.

Cases: The cases for WFT:ABS and WFT:APP are analogous to those for WFC:ABS and WFC:APP.

Case:

$$\frac{\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma}{\Delta^\star;\Gamma \vdash \Lambda\alpha{:}\star^\ell.e : \forall^\perp\alpha{:}\star^\ell.\sigma}\ \textsc{wft:tabs}$$

- By the definition of substitution, we know that $\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) = \Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e))$. Furthermore, by TRC:REFL we know that $\Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e)) \rightsquigarrow^* \Lambda\alpha{:}\star^\ell.\delta_i(\gamma_i(e))$. Therefore, we have that $(\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) \rightsquigarrow^* (\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e))$.
- Assume $\delta_1(\tau_1) \approx_{\ell_o} \delta_2(\tau_2) : \star^\ell$ and a consistent R such that

$$R^{\ell_2}_\rho \in \delta_1((\rho\{\tau_1\}) \mathbin{@} \ell_2) \leftrightarrow \delta_2((\rho\{\tau_2\}) \mathbin{@} \ell_2).$$

- Therefore, by Definition 3.12 and RELM:REG we know that $\delta_1,\delta_2 \vdash \eta,\alpha \mapsto R : \Delta^\star,\alpha{:}\star^\ell$ and $\delta_1,[\delta_1(\tau_1)/\alpha] \approx_{\ell_o} \delta_2,[\delta_2(\tau_2)/\alpha] : \Delta^\star,\alpha{:}\star^\ell$.
- Appealing to the induction hypothesis on $\Delta^\star,\alpha{:}\star^\ell;\Gamma \vdash e : \sigma$ with the above gives us that

$$\eta,\alpha \mapsto R \vdash (\delta_1,[\delta_1(\tau_1)/\alpha])(\gamma_1(e)) \approx_{\ell_o} (\delta_2,[\delta_2(\tau_2)/\alpha])(\gamma_2(e)) : \sigma$$

- Using Lemma B.1 we can conclude that

$$\eta,\alpha \mapsto R \vdash \delta_1(\gamma_1((\Lambda\alpha{:}\star^\ell.e)[\tau_1])) \approx_{\ell_o} \delta_2(\gamma_2((\Lambda\alpha{:}\star^\ell.e)[\tau_2])) : \sigma$$

  Furthermore, by Lemma 3.18 and $\Delta^\star \vdash \sigma \le \sigma \sqcup \perp$ we know that

$$\eta,\alpha \mapsto R \vdash \delta_1(\gamma_1((\Lambda\alpha{:}\star^\ell.e)[\tau_1])) \approx_{\ell_o} \delta_2(\gamma_2((\Lambda\alpha{:}\star^\ell.e)[\tau_2])) : \sigma \sqcup \perp$$

- Discharging our assumptions, we have that

$$\eta \vdash \delta_1(\gamma_1(\Lambda\alpha{:}\star^\ell.e)) \sim_{\ell_o} \delta_2(\gamma_2(\Lambda\alpha{:}\star^\ell.e)) : \forall^\perp\alpha{:}\star^\ell.\sigma$$

  Using this along with $(\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e)) \rightsquigarrow^* (\delta_i(\gamma_i(\Lambda\alpha{:}\star^\ell.e))$ and SCLR:TERM we can conclude that

$$\eta \vdash \delta_1(\gamma_1(\Lambda\alpha{:}\star^\ell.e)) \approx_{\ell_o} \delta_2(\gamma_2(\Lambda\alpha{:}\star^\ell.e)) : \forall^\perp\alpha{:}\star^\ell.\sigma$$

Case:

$$\frac{\Delta^\star;\Gamma \vdash e : \forall^\ell\alpha{:}\star^{\ell'}.\sigma \qquad \Delta^\star \vdash \tau : \star^{\ell'}}{\Delta^\star;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \sqcup \ell}\ \textsc{wft:tapp}$$

- Appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e : \forall^\ell\alpha{:}\star^{\ell'}.\sigma$, we get that $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_1} \delta_2(\gamma_2(e)) : \forall^\ell\alpha{:}\star^{\ell'}.\sigma$.
- By inversion on $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \forall^\ell\alpha{:}\star^{\ell'}.\sigma$ we know that either $\delta_i(\gamma_i(e)) \rightsquigarrow^* \nu_i$ or $\delta_i(\gamma_i(e))\uparrow$.

  Sub-Case: $\delta_i(\gamma_i(e)) \rightsquigarrow^* \nu_i$.

  – Also inversion we know that, $\forall^\ell\alpha{:}\star^{\ell'}.\sigma' \rightsquigarrow^* \zeta$ and $\eta \vdash \nu_1 \sim_{\ell_o} \nu_2 : \zeta$. By inversion on the weak-head reduction we know that $\zeta = \forall^\ell\alpha{:}\star^{\ell'}.\sigma$. Inverting $\eta \vdash \nu_1 \sim_{\ell_1} \nu_2 : \forall^\ell\alpha{:}\star^{\ell'}.\sigma$ we know that

$$\begin{array}{l}\forall(\delta_1(\tau_1') \approx_{\ell_o} \delta_2(\tau_2') : \star^{\ell'}).\\ \quad \forall(R^{\ell'}_\rho \in \delta_1((\rho\{\tau_1'\}) \mathbin{@} \ell') \leftrightarrow \delta_2((\rho\{\tau_2'\}) \mathbin{@} \ell').\\ \quad\quad \eta,\alpha \mapsto R \vdash \nu_1[\tau_1] \approx_{\ell_1} \nu_2[\tau_2] : \sigma \sqcup \ell\end{array}$$

  – Using Part 1 on $\Delta^\star \vdash \tau : \star^{\ell'}$ we have that $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \star^{\ell'}$.

– Choose $\mathsf{R}_\rho^{\ell'}$ to be

$$\{(v_1, v_2) \mid \eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta, (\rho\{\tau\}) \mathbin{@} \ell' \rightsquigarrow^* \zeta\}.$$

– Applying $\delta_1(\tau) \approx_{\ell_o} \delta_2(\tau) : \star^{\ell'}$ and R gives us that

$$\eta, \alpha \mapsto \mathsf{R} \vdash v_1[\delta_1(\tau)] \approx_{\ell_1} v_2[\delta_2(\tau)] : \sigma \sqcup \ell$$

Using Lemma 3.23 on this we can conclude

$$\eta \vdash v_1[\delta_1(\tau)] \approx_{\ell_1} v_2[\delta_2(\tau)] : \sigma[\tau/\alpha] \sqcup \ell$$

– Given that $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ we know that $\delta_i(\gamma_i(e))[\delta_i(\tau)] \rightsquigarrow^* v_i[\delta_i(\tau)]$. Using Lemma B.1 we can conclude that

$\eta \vdash \delta_1(\gamma_1(e))[\delta_1(\tau)] \approx_{\ell_1} \delta_1(\gamma_2(e))[\delta_2(\tau)] : \sigma[\tau/\alpha] \sqcup \ell$

which by the definition of substitution is identical to the desired result

$$\eta \vdash \delta_1(\gamma_1(e[\tau])) \approx_{\ell_1} \delta_1(\gamma_2(e[\tau])) : \sigma[\tau/\alpha] \sqcup \ell$$

Sub-Case: $\delta_i(\gamma_i(e)) \uparrow$.

– Then we know that $\delta_i(\gamma_i(e[\tau])) \uparrow$ as well. Using SCLR:DIVR1 or SCLR:DIVR2 we can conclude $\eta \vdash \delta_1(\gamma_1(e[\tau])) \approx_{\ell_o} \delta_2(\gamma_2(e[\tau])) : \sigma[\tau/\alpha] \sqcup \ell$.

Case:

$$\frac{\Delta^\star; \Gamma \vdash e_1 : \sigma_1 \qquad \Delta^\star; \Gamma \vdash e_2 : \sigma_2}{\Delta^\star; \Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times^\perp \sigma_2}\ \textsc{wft:pair}.$$

- By appealing to the induction hypothesis on $\Delta^\star; \Gamma \vdash e_1 : \sigma_1$ and $\Delta^\star; \Gamma \vdash e_2 : \sigma_2$ with $\delta_1 \approx_{\ell_o} \delta_2 : \Delta^\star$ and $\delta_1, \delta_2 \vdash \eta : \Delta^\star$ and $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ we have that

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$$

and

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_2$$

- By inversion on $\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$ either $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$ or $\delta_i(\gamma_i(e_1)) \uparrow$.

Sub-Case: $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$.

– By inversion upon $\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma_2$ either $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$ or $\delta_i(\gamma_i(e_2)) \uparrow$.

Sub-Sub-Case: $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$.

∗ Because $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_{1i}$ and $\delta_i(\gamma_i(e_2)) \rightsquigarrow^* v_{2i}$ we can conclude that $\langle \delta_i(\gamma_i(e_1)), \delta_i(\gamma_i(e_2)) \rangle \rightsquigarrow^* \langle v_{1i}, v_{2i} \rangle$ which by the definition of substitution is identical to $\delta_i(\gamma_i(\langle e_1, e_2 \rangle)) \rightsquigarrow^* \langle v_{1i}, v_{2i} \rangle$.

∗ Therefore, $\mathbf{fst}\, \delta_i(\gamma_i(\langle e_1, e_2 \rangle)) \rightsquigarrow^* v_{1i}$ and $\mathbf{snd}\, \delta_i(\gamma_i(\langle e_1, e_2 \rangle)) \rightsquigarrow^* v_{2i}$ respectively. Also by the above inversions upon

$$\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : \sigma_1$$

and

$$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma_2$$

we know that $\eta \vdash v_{11} \sim_{\ell_o} v_{12} : \zeta_1$ and $\eta \vdash v_{21} \sim_{\ell_o} v_{22} : \zeta_2$ for $\sigma_1 \rightsquigarrow^* \zeta_1$ and $\sigma_2 \rightsquigarrow^* \zeta_2$.

∗ Using Lemma 3.18 on these along with $\Delta^\star \vdash \zeta_i \le \zeta_i \sqcup \perp$ and $\Delta^\star \vdash \sigma_i \le \sigma_i \sqcup \perp$ we have that $\eta \vdash v_{11} \sim_{\ell_o} v_{12} : \zeta_1 \sqcup \perp$ and $\eta \vdash v_{21} \sim_{\ell_o} v_{22} : \zeta_2 \sqcup \perp$ for $\sigma_1 \sqcup \perp \rightsquigarrow^* \zeta_1 \sqcup \perp$ and $\sigma_2 \sqcup \perp \rightsquigarrow^* \zeta_2 \sqcup \perp$.

∗ Consequently, by SCLR:TERM we have that

$$\eta \vdash \mathbf{fst}\,\delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \mathbf{fst}\,\delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \sqcup \bot$$

and

$$\eta \vdash \mathbf{snd}\,\delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \mathbf{snd}\,\delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_2 \sqcup \bot$$

∗ Finally, by SLR:PROD we can conclude

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \sim_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^{\bot} \sigma_2$$

Using this along with $\langle \delta_i(\gamma_i(e_1)),\delta_i(\gamma_i(e_2))\rangle \rightsquigarrow^* \langle v_{1i},v_{2i}\rangle$ gives us the desired result

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^{\bot} \sigma_2$$

Sub-Sub-Case: $\delta_i(\gamma_i(e_2))\uparrow$.

∗ Then we know that $\delta_i(\gamma_i(\langle e_1,e_2\rangle))\uparrow$ and we can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^{\bot} \sigma_2$$

Sub-Case: $\delta_i(\gamma_i(e_1))\uparrow$.

– Then we know that $\delta_i(\gamma_i(\langle e_1,e_2\rangle))\uparrow$ and we can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(\gamma_1(\langle e_1,e_2\rangle)) \approx_{\ell_o} \delta_2(\gamma_2(\langle e_1,e_2\rangle)) : \sigma_1 \times^{\bot} \sigma_2$$

Case:

$$\frac{\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta^\star;\Gamma \vdash \mathbf{fst}\,e : \sigma_1 \sqcup \ell}\ \text{WFT:FST}$$

- Appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2$ we know that $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$.
- By inversion upon $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$ we know that either $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ or $\delta_i(\gamma_i(e))\uparrow$.

Sub-Case: $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$,

– Also by inversion upon

$$\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_o} \delta_2(\gamma_2(e)) : \sigma_1 \times^{\ell} \sigma_2$$

we have that $\sigma_1 \times^{\ell} \sigma_2 \rightsquigarrow^* \sigma'$ $\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma'$.

– By inversion upon $\sigma_1 \times^{\ell} \sigma_2 \rightsquigarrow^* \sigma'$ we know that $\sigma' = \sigma_1 \times^{\ell} \sigma_2$.

– By inversion upon $\eta \vdash v_1 \sim_{\ell_o} v_2 : \sigma_1 \times^{\ell} \sigma_2$ we know that $\eta \vdash \mathbf{fst}\,v_1 \approx_{\ell_o} \mathbf{fst}\,v_2 : \sigma_1 \sqcup \ell$ and $\eta \vdash \mathbf{snd}\,v_1 \approx_{\ell_o} \mathbf{snd}\,v_2 : \sigma_2 \sqcup \ell$.

– Given that $\delta_i(\gamma_i(e)) \rightsquigarrow^* v_i$ we know that $\mathbf{fst}\,\delta_i(\gamma_i(e)) \rightsquigarrow^* \mathbf{fst}\,v_i$ which by the definition of substitution is the same as $\delta_i(\gamma_i(\mathbf{fst}\,e)) \rightsquigarrow^* \mathbf{fst}\,v_i$. Therefore by Lemma B.1 we can conclude that

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fst}\,e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fst}\,e)) : \sigma_1 \sqcup \ell$$

Sub-Case: $\delta_i(\gamma_i(e))\uparrow$

– Therefore, we can conclude that $\mathbf{fst}\,\delta_i(\gamma_i(e))\uparrow$, which by the definition of substitution is the same as $\delta_i(\gamma_i(\mathbf{fst}\,e))\uparrow$. Therefore, by SCLR:DIVR1 or SCLR:DIVR2 we have that $\eta \vdash \delta_1(\gamma_1(\mathbf{fst}\,e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fst}\,e)) : \sigma_1 \sqcup \ell$.

Case: The case for WFT:SND is symmetric to the case for WFT:FST.

Case:

$$\frac{\Delta^\star;\Gamma \vdash e_1 : (\mathsf{bool})\ @\ \ell \qquad \Delta^\star;\Gamma \vdash e_2 : \sigma \qquad \Delta^\star;\Gamma \vdash e_3 : \sigma}{\Delta^\star;\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma \sqcup \ell}\ \textsc{wft:if}$$

Sub-Case: $\ell \not\sqsubseteq \ell_o$.

- Then by Lemma 3.14 we know that

$$\eta \vdash \delta_1(\gamma_1(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) : \sigma \sqcup \ell$$

Sub-Case: $\ell \sqsubseteq \ell_o$.

- By appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e_1 : (\mathsf{bool})\ @\ \ell$ we know that $\eta \vdash \delta_1(\gamma_1(e_1)) \approx_{\ell_o} \delta_2(\gamma_2(e_1)) : (\mathsf{bool})\ @\ \ell$. By inversion on this we know that either $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_i$ or $\delta_i(\gamma_i(e_1)) \uparrow$.

  Sub-Sub-Case: $\delta_i(\gamma_i(e_1)) \rightsquigarrow^* v_i$.

  - Also by inversion we know that $\eta \vdash v_1 \sim_{\ell_o} v_2 : \zeta$ where $(\mathsf{bool})\ @\ \ell \rightsquigarrow^* \zeta$. And by inversion on the weak-head reduction we know that $\zeta = (\mathsf{bool})\ @\ \ell$.
  - Therefore, by inversion upon $\eta \vdash v_1 \sim_{\ell_o} v_2 : (\mathsf{bool})\ @\ \ell$ we can conclude $\ell \sqsubseteq \ell_o \Rightarrow v_1 = v_2$. We assumed that $\ell \sqsubseteq \ell_o$, so $v_1 = v_2$.
  - By Lemma A.15 we know that $v_i = \mathbf{true}$ or $v_i = \mathbf{false}$.

    Sub-Sub-Sub-Case: $v_i = \mathbf{true}$. By appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash e_1 : (\mathsf{bool})\ @\ \ell$ we know that

    $$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma$$

    By Lemma 3.18 we can conclude

    $$\eta \vdash \delta_1(\gamma_1(e_2)) \approx_{\ell_o} \delta_2(\gamma_2(e_2)) : \sigma \sqcup \ell$$

    We know that $\delta_i(\gamma_i(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) \rightsquigarrow^* \delta_i(\gamma_i(e_2))$, therefore by
    Lemma B.1 we can conclude the desired result

$$\eta \vdash \delta_1(\gamma_1(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) : \sigma \sqcup \ell$$

    Sub-Sub-Sub-Case: The case for $v_i = \mathbf{false}$ is symmetric.

  Sub-Sub-Case: $\delta_i(\gamma_i(e_1)) \uparrow$.

  - Then we know that $\delta_i(\gamma_i(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) \uparrow$ and can use either SCLR:DIVR1 or SCLR:DIVR2 to conclude that

$$\eta \vdash \delta_1(\gamma_1(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)) : \sigma \sqcup \ell$$

Case:

$$\frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}_n\ x{:}\sigma.e : \sigma}\ \textsc{wft:fixn}$$

- By the definition of substitution, we know that $\delta_i(\gamma_i(\mathbf{fix}_n\ x{:}\sigma.e)) = \mathbf{fix}_n\ x{:}\sigma.\delta_i(\gamma_i(e))$.
- The case follows from induction upon n.

  Sub-Case: $n = o$.

  - By Lemma 3.4 we know that $\mathbf{fix}_o\ x{:}\sigma.\delta_i(\gamma_i(e)) \uparrow$. Therefore, by SCLR:DIVR1 or SCLR:DIVR2o we can conclude that

$$\eta \vdash \mathbf{fix}_o\ x{:}\sigma.\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_o\ x{:}\sigma.\delta_2(\gamma_2(e)) : \sigma$$

– By the above identity, this means that we have

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_o\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_o\ x{:}\sigma.e)) : \sigma$$

Sub-Case: $n = m + 1$.

– By appealing to the local induction hypothesis on m gives us that $\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)) : \sigma$.

– By Definition 3.22 and inversion upon $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ we can conclude that

$$\eta \vdash \gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x] \approx_{\ell_o} \gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x] : \Gamma,x{:}\sigma$$

– Appealing to the global induction hypothesis on $\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma$ with

$$\eta \vdash \gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x] \approx_{\ell_o} \gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x] : \Gamma,x{:}\sigma$$

gives us that

$$\eta \vdash \delta_1((\gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) \approx_{\ell_o} \delta_2((\gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) : \sigma$$

– Trivially, $n - 1 = m$, so using Lemmas B.1 on

$$\eta \vdash \delta_1((\gamma_1,[\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) \approx_{\ell_o} \delta_2((\gamma_2,[\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)/x])(e)) : \sigma$$

we can conclude

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_n\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_n\ x{:}\sigma.e)) : \sigma$$

Case:

$$\frac{\Delta^\star;\Gamma,x{:}\sigma \vdash e : \sigma \qquad \Delta^\star \vdash \sigma}{\Delta^\star;\Gamma \vdash \mathbf{fix}_\omega\ x{:}\sigma.e : \sigma}\ \textsc{wft:fixlim}$$

- By inversion on WFT:FIXLIM and we can apply WFT:FIXN to show that for any n, $\Delta^\star;\Gamma \vdash \mathbf{fix}_n\ x{:}\sigma.e : \sigma$.
- Therefore, assume an arbitrary m. Appealing to the induction hypothesis on $\Delta^\star;\Gamma \vdash \mathbf{fix}_m\ x{:}\sigma.e : \sigma$ with $\eta \vdash \gamma_1 \approx_{\ell_o} \gamma_2 : \Gamma$ gives us that $\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_m\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_m\ x{:}\sigma.e)) : \sigma$.
- By the definition of substitution $\delta_i(\gamma_i(\mathbf{fix}_m\ x{:}\sigma.e)) = \mathbf{fix}_m\ x{:}\delta_i(\sigma).\delta_i(\gamma_i(e))$. Therefore, we have that

$$\eta \vdash \mathbf{fix}_m\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_m\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

- Discharging our assumption we have that for all n,

$$\eta \vdash \mathbf{fix}_n\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_n\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

Using Lemma 3.7 we can conclude

$$\eta \vdash \mathbf{fix}_\omega\ x{:}\delta_1(\sigma).\delta_1(\gamma_1(e)) \approx_{\ell_o} \mathbf{fix}_\omega\ x{:}\delta_2(\sigma).\delta_2(\gamma_2(e)) : \sigma$$

- Again by the definition of substitution, $\delta_i(\gamma_i(\mathbf{fix}_\omega\ x{:}\sigma.e)) = \mathbf{fix}_\omega\ x{:}\delta_i(\sigma).\delta_i(\gamma_i(e))$. Therefore, we have the desired result

$$\eta \vdash \delta_1(\gamma_1(\mathbf{fix}_\omega\ x{:}\sigma.e)) \approx_{\ell_o} \delta_2(\gamma_2(\mathbf{fix}_\omega\ x{:}\sigma.e)) : \sigma$$

Case: The case for WFT:TCASE is analogous to WFT:IF and WFT:TAPP.

Case: The case for WFT:SUB is analogous to that for WFC:SUB.

□