Generalizing Parametricity Using Information Flow

Geoffrey Washburn Stephanie Weirich
{geoffw, sweirich}@cis.upenn.edu
Department of Computer and Information Science
University of Pennsylvania

Abstract

Run-time type analysis allows programmers to easily and concisely define many operations that are determined by type structure, such as serialization, iterators, and structural equality. Simultaneously, when types may be inspected at run time, nothing is secret. It is impossible for a module writer to use type abstraction to hide implementation details from clients: those clients may use type analysis to determine the structure of these supposedly "abstract" data types. Furthermore, access control mechanisms do not help to isolate the implementation of abstract datatypes from their clients. Authorized modules may simply leak type information to unauthorized clients, so module implementors cannot reliably tell what parts of a program rely on their type definitions and what parts do not.

Currently, module implementors rely on parametric polymorphism to provide guarantees about the use of their abstract datatypes. Although standard type parametricity does not hold for a language with run-time type analysis, in this paper we show how to generalize parametricity so that it does hold in the presence of type analysis, yet still encompasses the integrity and confidentiality policies that are normally derived from parametricity. The key is to augment the type system with annotations about information flow. Because the type system tracks the flow of dynamic type information, the implementor of an abstract data type can easily see what parts of the program depend on the implementation of that type.

1 Introduction

Run-time type analysis makes it possible to write programs that depend on the type structure of data. Applications include marshalling, generic iteration, the ability to work with data structures from dynamically loaded code, and many others.

However, the flexibility provided by run-time type analysis is at odds with type abstraction. Module implementors rely on *parametric* polymorphism to provide guarantees about the use of their abstract datatypes. With type parametricity, a program must treat values with abstract types as black boxes; such values can only be "pushed around", not inspected, modified or even created. However, because run-time type analysis allows types to be analyzed dynamically, the clients of a given module may open those black boxes. At best, this loss of abstraction hinders modular development, and at worst it allows malicious clients to subvert a complex system.

How can a module implementer be sure that the invariants of her abstract datatypes are maintained? One answer is to simply prohibit run-time type analysis. However, we believe that the benefits of type directed programming are too compelling to abandon altogether. Therefore, it makes sense to study the problem and seek a more practical solution. We would like some way for the type system to enforce abstraction boundaries, yet still allow the most common uses of type analysis.

Another solution that has been suggested is to use access control to restrict type analysis. At its simplest, access control can distinguish between analyzable and unanalyzable types. A more elaborate access control scheme could distinguish between the authority under which a program fragment operates, and specify for each abstract type, the clients that have permission to analyze its structure.

Unfortunately, access control does not properly protect type abstraction. It only restricts which clients may analyze a type, not which clients may make use of information learned from type analysis. Therefore, access control requires us to place undue trust in a client not to provide others with the capabilities and information it has been granted.

Instead, we propose the use of an information flow type system to give programmers the ability to reason about the integrity and confidentiality of their type abstractions. Part of the novelty of our approach is that existing information-flow languages only track the information content of terms—we propose that the information content of types also be considered. Our key result is that by tracking information flows we can prove a theorem that generalizes the standard parametricity theorem to languages with run-time type analysis, and can be used in the same manner as parametricity to establish integrity and confidentiality properties of programs.

The structure of this paper is as follows. We examine the problem and our solution in more detail in Section 2. In Section 3 we describe a core calculus with information flow and type analysis, and follow with an examination of a generalized notion of parametricity for that language in Section 4. Section 5 considers an extension to the core calculus with bounded label polymorphism, and Section 6 discusses related work.

2 Tracking the dynamic use of type information

Type analysis is an important tool for programming. It enables concise expression of operations that traverse the structure of data. Traditional examples include serialization, structural equality, cloning and iteration. However, many systems use type analysis for more sophisticated purposes such as automatically generating graphical user interfaces, debuggers, testing code, and XML support. For this reason, it is important to support type analysis in modern programming languages.

For example, we may use type analysis to implement structural equality in a functional programming language.

```
fun eq['a] =
  typecase 'a of
   int     => fn (x:int, y:int) => x == y
   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 first analyzes its type argument 'a and then returns an equality function for terms of that type. For product types, eq must call itself recursively on the subcomponents of the product.

More examples of type analysis include generic functions for reading and writing arbitrary datatypes, and a type-safe cast function that compares two types for equality. Casting is especially important in systems that support dynamic loading, because it can be used to verify that newly loaded values have the expected type [11]. The interface to some of these generic functions appears Figure 1.

```
module Generic : sig
  (* polymorphic equality *)
  val eq : forall 'a . 'a -> 'a -> bool
  (* marshall a value to a string *)
  val write : forall 'a . 'a -> string
  (* unmarshall a value from a string *)
  val read : forall 'a . string -> 'a
  (* if 'a = 'b, the identity function *)
  val cast : forall 'a 'b . ('a -> 'b) option
end
```

Figure 1: An example generic library

Designers of abstract datatypes can use generic operations to get quick implementations of some of the operations for their datatypes. For example, even though the type employee is known statically, it's easy to define an equality routine for it by using the generic equality function.

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

Although type analysis is very useful, it may 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 to hold. Using type-safe cast, anyone may create a value of type Employee.t. Although the type will be correct, other constraints about this value that are not present in the type system may not be maintained. For example, the following malicious code creates an employee with an invalid 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 some aspects of the employee data type hidden, another module can simply use the generic operations to discover them. For example, even if no access was provided to the salary component of an Employee.t, the following malicious code can still extract this component.

```
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!"
```

The purpose of this paper is to propose a language that permits type analysis, yet allows module writers to define integrity and confidentiality policies for their abstract datatypes. We are interested in a language where the authors of an abstract datatype can answer the following questions:

> *If I change the definition of my abstract datatype, how will that affect the rest of the program? If there are changes elsewhere in the program, how will they affect my code?*

In traditional module systems, that question is easy to answer: if the type Employee.t is abstract outside of the module Employee, then the programmer knows that all other code must treat values of type Employee.t parametrically. Therefore, any changes to Employee.t will only require modifications to code that is inside the Employee module.

However, in the presence of type analysis, the programmer cannot know what code may depend on the definition of an abstract datatype. Any other part of the program could dynamically discover the underlying type and therefore introduce dependencies on its definition.

Past work has suggested that a distinction between analyzable and unanalyzable types should be considered [9]. Unfortunately, just controlling which parts of the program are allowed to analyze a type does not allow programmers to answer the above questions. Imagine an extension where it is possible to specify which modules may analyze a type. For example, in the following code only the modules A and B may analyze the type A.t, and only modules B and C may analyze the type B.u.

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

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

module C = struct
  val z =
    case (cast [B.u] [int]) with
     Some f => "I know it's an int"
     None   => "I know it's not an int"
end
```

It is not the case that module C is parametric with respect to the definition of 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 the module C. 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 module A cannot know about this dependency. Therefore, we must look beyond access-control for a solution.

Our proposal is that by tracking the flow of type information through the program with *information-flow labels*, the programmer can determine how type definitions influence the results of computation. Information-flow types label a standard type system with elements of a lattice that describes information content for each computation. For example, given a simple lattice containing two points L (low) and H (high), then an expression of type $\texttt{int}^{\mathsf{H}}$ produces an integer using high-security information, while an expression of type $\texttt{int}^{\mathsf{L}}$ is only requires low-security information in its computation. The novelty of our approach compared to previous information flow type systems in the literature that we also label kinds with the information content of type constructors.

Therefore, to regain the ability to reason about abstract types in the presence of type analysis, we label types with an information content that is "high-security". Consequently, computations that depend on those types must also be high-security. For example, if we label Employee.t with H, computations that require analysis of the type Employee.t will also be labeled H. The value forged above must be assigned the type $\texttt{Employee.t}^{\mathsf{H}}$ because type analysis (in the cast) is used to compute this value. Alternately, those computations that are parametric in Employee.t may be labeled with L.

Furthermore, only those values created using the module itself will be able to have the type $\texttt{Employee.t}^{\mathsf{L}}$, because there is no other way to create values of type

`Employee.t` without using type analysis and tainting the result. This means that if the `Employee` module requires all of its inputs be of type `Employee.t`$^{\mathsf{L}}$, then it is impossible to provide them with forged values. Consequently, the module writer has more guarantees about the integrity of the module because she knows that the module will encounter values that it created. The module invariants will be maintained. For example, a function that computes the amount of money needed to pay a list of employees could merely add up their salaries—it does not need to check that some employee might have a negative salary.

One of the key reasons that tracking information flow avoids the problem with access control given above, is that security levels are propagated through code even if it does not analyze type information. For example, we can assign an identity function both the type `Employee.t`$^{\mathsf{L}} \rightarrow^{\mathsf{L}}$ `Employee.t`$^{\mathsf{L}}$ and the type `Employee.t`$^{\mathsf{H}} \rightarrow^{\mathsf{L}}$ `Employee.t`$^{\mathsf{H}}$ witnessing that it propagates the information content of the argument to the result. Here the function type constructor $\rightarrow^{\mathsf{L}}$ is labeled itself to indicate the information content of creating the function—generating the identity function does not require any type analysis.

Because some functions, such as identity, are parametric in the information content of their arguments, some security-type systems include *security-label polymorphism*—allowing functions to be polymorphic over the security labels of their inputs. For example, the `spy` function above could be assigned the type `forall` ℓ. `Employee.t`$^{\ell} \rightarrow^{\ell}$ `int`$^{\ell \sqcup \mathsf{H}}$. It can be used on employees at any security level, but because the resulting value is computed via type analysis, the result type must be at least as high as H. Another place that label polymorphism would be useful is in the functions of the module `Generic`. Because we would like to be able to apply these generic operations to values regardless of their security label, the functions must be polymorphic in the label of their inputs.

Using security labels to track the dynamic flow of type information works well with existing mechanisms for controlling how type information is propagated statically. Returning to module `A` from above, we give a type definition a low-security label in the implementation, but seal the module with a signature that makes the type definition carry a high-level information content.

```
module A = struct
  type tL = int
  val x = 3
end :> sig
  type tH
  val x : tL
end
```

This frees the module's author to employ type analysis in the implementation of the module, but prevents others from analyzing `A.t` without their computations being marked high-security.

3 The λ_{iSEC} language

Next, we describe λ_{iSEC}, a core calculus with information flow and type analysis. To focus on the difficulties type analysis presents, we designed λ_{iSEC} to be as simple as possible while still retaining the flavor of the problem. This language is derived from the type-analyzing language λ_i^{ML} developed by Harper and Morrisett [9] and the information-flow security language λ_{SEC} of Zdancewic [32]. We base λ_{iSEC} on λ_i^{ML} because it provides a simple yet expressive model of run-time type analysis. This core language was developed as a typed intermediate language for efficient compilation of parametric polymorphism. Similarly, λ_{SEC} is a core language developed to study information flow in the context of the simply-typed λ-calculus.

3.1 Run-time type analysis

The structure of λ_{iSEC} follows that of λ_i^{ML}. The grammar for λ_{iSEC} appears in Figure 3. (For reference, the complete semantics appears in the Appendix.) This language is a predicative, polymorphic lambda calculus with integers, functions and recursion. As in λ_i^{ML}, type constructors, τ, which are analyzed at run-time, are separated from types, σ, which describe terms. For technical reasons that we describe in Section 4 we restrict polymorphism to type constructors of base kind, and do not allow higher-order polymorphism.

As in λ_i^{ML}, the term form **typecase** may be used to define operations that depend on run-time type information. This term takes a type constructor to scrutinize, τ, as well as branches $(e_{\mathsf{int}}, e_{\rightarrow}, e_{\times})$ for each of the primitive type constructors. During, evaluation, the type constructor argument must be reduced to determine its head form, and the corresponding branch. In addition to the "simply typed" λ-calculus, the language of constructors has three forms that correspond to types: int, $\tau_1 \rightarrow \tau_2$, and $\tau_1 \times \tau_2$.

$$\frac{\tau \rightsquigarrow^* \mathsf{int}}{\textbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\mathsf{int}}}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \rightarrow \tau_2}{\textbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\rightarrow}[\tau_1][\tau_2]}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \times \tau_2}{\textbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\rightarrow}\ e_{\times} \rightsquigarrow e_{\times}[\tau_1][\tau_2]}$$

We write $e \rightsquigarrow e'$ to mean that term e reduces in a single step to e' and $\tau \rightsquigarrow \tau'$ to mean that constructor τ makes a weak-head reduction step to τ'.

Similar to the term level **typecase**, λ_i^{ML} also includes a type constructor, Typerec, that analyzes type information. Without Typerec, it is impossible to assign types to some useful type analyzing terms. Typerec implements a *paramorphism* (a type of fold) over the structure of the argument constructor. It takes as arguments a scrutinized constructor τ and branches $(\tau_{int}, \tau_{\rightarrow}, \tau_{\times})$ to handle each of the primitive type constructors. As is shown in the following constructor reduction rules, when the head of the argument is one of the three primitive constructors, Typerec will apply the constituents as well as the recursive invocation of Typerec on them to the appropriate branch.

$$\frac{}{\mathsf{Typerec}\ (\mathsf{int})\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \tau_{int}}$$

$$\frac{}{\begin{array}{l}\mathsf{Typerec}\ (\tau_1 \rightarrow \tau_2)\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \\ \quad \tau_{\rightarrow}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times}) \\ \qquad\qquad (\mathsf{Typerec}\ \tau_2\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times})\end{array}}$$

$$\frac{}{\begin{array}{l}\mathsf{Typerec}\ (\tau_1 \times \tau_2)\tau_{int}\ \tau_{\rightarrow}\ \tau_{\times} \rightsquigarrow \\ \quad \tau_{\times}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times}) \\ \qquad\qquad (\mathsf{Typerec}\ \tau_2\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times})\end{array}}$$

3.2 The information content of type constructors

Security type systems annotate types with labels that specify the information content of terms. Because our type constructors have computational content (and influence the evaluation of terms) in λ_{iSEC}, we must also label kinds. Labels, ℓ, are drawn from an unspecified lattice with a least element ($\bot$), joins ($\sqcup$) and a partial order ($\sqsubseteq$). The actual lattice that the type system uses is determined by the confidentiality and integrity policies of the program. Intuitively, the higher a label is in the lattice, the more information is required during the computation of that constructor or value. In the simple example in the previous section, we used a two point lattice that tracks the dynamic discovery of a single type definition, but any arbitrary lattice may be used. One could also use a lattice with richer internal structure, such as the Decentralized Label Model (DLM) of Myers and Liskov [20].

The labels of kinds describe the information content of type constructors. The kind of a constructor (and therefore its security level) is described using the judgement $\Delta \vdash \tau : \kappa$, read as "constructor τ is well-formed having kind κ with respect to the type variable context

kinds

$\kappa ::= \star^{\ell} \mid \kappa_1 \xrightarrow{\ell} \kappa_2$

type constructors

$\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	λ-*calculus*
$\mid\ \mathsf{int}$	*integers*
$\mid\ \tau_1 \rightarrow \tau_2$	*functions*
$\mid\ \tau_1 \times \tau_2$	*products*
$\mid\ \mathsf{Typerec}\tau\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times}$	*analysis*

normal-form constructors

$\nu ::= \alpha \mid \nu\,\tau \mid \mathsf{int} \mid \tau_1 \rightarrow \tau_2$
$\quad \mid\ \tau_1 \times \tau_2 \mid \mathsf{Typerec}\nu\ \tau_{int}\ \tau_{\rightarrow}\ \tau_{\times}$

types

$\sigma ::= (\tau)^{\ell}$	*injection*
$\mid\ \sigma_1 \xrightarrow{\ell} \sigma_2$	*functions*
$\mid\ \sigma_1 \times^{\ell} \sigma_2$	*products*
$\mid\ \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$	*con poly*

terms

$e ::= i$	*integers*
$\mid\ x \mid \lambda x{:}\sigma.e \mid e_1 e_2$	λ-*calculus*
$\mid\ \langle e_1, e_2\rangle \mid$ **fst** $e \mid$ **snd** e	*tuples*
$\mid\ \Lambda\alpha{:}\star^{\ell}.e \mid e[\tau]$	*con poly*
$\mid$ **fix** $x{:}\sigma.e$	*fix-point*
$\mid$ **typecase**$[\gamma.\sigma]\ \tau\ e_{int}\ e_{\rightarrow}\ e_{\times}$	*analysis*

values

$v ::= i \mid \lambda x{:}\sigma.e \mid \langle v_1, v_2\rangle \mid \Lambda\alpha{:}\star^{\ell}.e$

term substitutions	γ	$::=$	$\cdot \mid \gamma, [e/x]$
type substitutions	δ	$::=$	$\cdot \mid \delta, [\tau/\alpha]$
term contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \sigma$
type contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha : \kappa$

Figure 2: The λ_{iSEC} language

$$\mathcal{L}(\star^{\ell}) \triangleq \ell \qquad \mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) \triangleq \ell$$

$$\mathcal{L}((\tau)^{\ell}) \triangleq \ell \qquad \mathcal{L}(\sigma_1 \xrightarrow{\ell} \sigma_2) \triangleq \ell$$
$$\mathcal{L}(\sigma_1 \times^{\ell} \sigma_2) \triangleq \ell \qquad \mathcal{L}(\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma) \triangleq \ell_1$$

Figure 3: Kind and type label operators

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{wfc:var} \qquad \frac{}{\Delta \vdash \textsf{int} : \star^{\bot}}\ \textsf{wfc:int}$$

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

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_1 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsf{wfc:prod}$$

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

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

$$\frac{\begin{array}{c}\Delta \vdash \tau : \star^{\ell} \qquad \ell \sqsubseteq \ell' \qquad \Delta \vdash \tau_{\textsf{int}} : \kappa \\ \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \text{where } \ell' = \mathcal{L}(\kappa)\end{array}}{\Delta \vdash \textsf{Typerec}\ \tau\ \tau_{\textsf{int}}\ \tau_{\to}\ \tau_{\times} : \kappa}\ \textsf{wfc:trec}$$

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

Figure 4: Constructor well-formedness

Δ." The rules of this judgement appear in Figure 4. The notation $\mathcal{L}(\kappa)$ (defined in Figure 3) refers to the label of κ.

Our calculus is conservative. If the label of κ is ℓ, then the information content of a constructor of kind κ is *at most* ℓ. A subsumption rule (at the bottom of Figure 4) allows constructors to be assigned higher security levels. Because kinds are labeled, the ordering $\sqsubseteq$ on labels induces a subkinding relation, $\kappa_1 \le \kappa_2$. A kind $\star^{\ell_1}$ is a subtype of $\star^{\ell_2}$ if $\ell_1 \sqsubseteq \ell_2$. Subkinding for function kinds is standard. The relation is by definition reflexive and transitive.

The label of a constructor τ, of base kind $\star^{\ell}$, describes the information required to produce that constructor, and the information gained when that constructor is analyzed. Type variables (such as `Employee.t`) may be given a high security level so that their information content may be traced throughout the program, as types are analyzed. For example, the kind of a Typerec expression must produce a constructor with kind labeled at least as high as that of the analyzed constructor, τ, to account for the information "learned" by inspecting τ. Likewise, the result type of a **typecase** expression must be at least as high as the label of the kind of τ.

By default the labels on the kinds for the three primitive type constructors are set as low as possible. Therefore, for int the initial kind is $\star^{\bot}$. The label of the kind for function and product types, must be at least as high as the join of its two constituent constructors, because it must reflect the information content of the entire constructor.

So that we can trace how information flows through type application, the kinds of type functions, $\kappa_1 \xrightarrow{\ell} \kappa_2$, have a label ℓ that represents the information propagated by invoking the function. Here the information, ℓ, carried by the function 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 rule for Typerec explicitly requires that the label of the kind of each branch, (and therefore the label of the whole constructor) be at least as high as that of the argument. Because we defined labels on arrow and product constructors to be at least as high in the lattice as its constituents, we can conservatively label the branch arguments that receive them with ℓ.

3.3 Tracking information flow in the term language

Like the kinds of constructors, the types of terms describe their information content. We use the judgement (defined in Figure 5) $\Delta \mid \Gamma \vdash e : \sigma$ to mean that "term e is well-formed with type σ with respect to the term context Γ and the type context Δ." This judgement requires a few auxiliary definitions. As we did for kinds, we define (in Figure 3) the operator $\mathcal{L}(\cdot)$ to refer the label of a type. Also, the judgement $\Delta \vdash \sigma$ is used to indicate that "type σ is well-formed with respect to type context Δ." Finally, the information labels for terms are also conservative. We can use the lattice ordering to derive a subtyping judgement $\Delta \vdash \sigma_1 \le \sigma_2$, and use a subsumption rule to raise the security level of an expression.

The types of λ_{iSEC} 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 $(\tau)^{\ell}$. It is important to note that in the well-formedness rule for types formed from type constructors, shown below

$$\frac{\Delta \vdash \tau : \star^{\ell_1}}{\Delta \vdash (\tau)^{\ell_2}}$$

there does not need to be a connection between ℓ_1 and ℓ_2. That is because ℓ_1 describes the information content of τ, while ℓ_2 describes the information content of a

$$\frac{}{\Delta \mid \Gamma \vdash i : \mathsf{int}^{\bot}}\ \mathsf{wft{:}int} \qquad \frac{x : \sigma \in \Gamma}{\Delta \mid \Gamma \vdash x : \sigma}\ \mathsf{wft{:}var}$$

$$\frac{\Delta \mid \Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \qquad \Delta \vdash \sigma_1}{\Delta \mid \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \xrightarrow{\bot} \sigma_2}\ \mathsf{wft{:}abs}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \sigma_1 \xrightarrow{\ell} \sigma_2 \qquad \Delta \mid \Gamma \vdash e_2 : \sigma_1}{\Delta \mid \Gamma \vdash e_1 e_2 : \sigma_2 \sqcup \ell}\ \mathsf{wft{:}app}$$

$$\frac{\Delta, \alpha{:}\star^{\ell} \mid \Gamma \vdash e : \sigma}{\Delta \mid \Gamma \vdash \Lambda\alpha{:}\star^{\ell}.e : \forall^{\bot}\alpha{:}\star^{\ell}.\sigma}\ \mathsf{wft{:}tabs}$$

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

$$\frac{\Delta \mid \Gamma \vdash e_1 : \sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : \sigma_2}{\Delta \mid \Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times^{\bot} \sigma_2}\ \mathsf{wft{:}pair}$$

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta \mid \Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell}\ \mathsf{wft{:}fst}$$

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta \mid \Gamma \vdash \mathbf{snd}\, e : \sigma_2 \sqcup \ell}\ \mathsf{wft{:}snd}$$

$$\frac{\Delta \mid \Gamma, x{:}\sigma \vdash e : \sigma \qquad \Delta \vdash \sigma}{\Delta \mid \Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \mathsf{wft{:}fix}$$

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

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \qquad \Delta \vdash \sigma_1 \leq \sigma_2}{\Delta \mid \Gamma \vdash e : \sigma_2}\ \mathsf{wft{:}sub}$$

Figure 5: Term well-formedness

term with type τ. It is sound to discard ℓ_1 because this type constructor will never be analyzed at run time. Once it has been coerced to a type, it may only be used statically to describe terms.

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

The **typecase** operation propagates information from the type constructor language to the term language.

$$\frac{\begin{array}{c} \Delta \vdash \tau : \star^{\ell} \\ \Delta, \gamma{:}\star^{\ell} \vdash \sigma \qquad \ell \sqsubseteq \ell' \qquad \Delta \mid \Gamma \vdash e_{\mathsf{int}} : \sigma[\mathsf{int}/\gamma] \\ \Delta \mid \Gamma \vdash e_{\rightarrow} : \forall^{\ell'}\alpha{:}\star^{\ell}.\forall^{\ell'}\beta{:}\star^{\ell}.\sigma[\alpha \rightarrow \beta/\gamma] \\ \Delta \mid \Gamma \vdash e_{\times} : \forall^{\ell'}\alpha{:}\star^{\ell}.\forall^{\ell'}\beta{:}\star^{\ell}.\sigma[\alpha \times \beta/\gamma] \\ \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma]) \end{array}}{\Delta \mid \Gamma \vdash \mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\rightarrow}\ e_{\times} : \sigma[\tau/\gamma]}$$

Like Typerec, because **typecase** examines the structure of the scrutinee and learns any information it might carry, we require the label ℓ' on the type to be at least as high in the lattice as the label ℓ on the scrutinee. Furthermore because we know that the label ℓ is at least as high as the labels on the constituents of τ, we are allowed to conservatively label the kinds of α and β with ℓ. Also, because we allow the result type of **typecase** to depend upon the scrutinized constructor τ, we must provide an annotation $[\gamma.\sigma]$, to make type checking easier.

3.4 Type-safety

λ_{iSEC} has the basic property one would expect from a typed language, that well-typed programs will not go wrong.

Theorem 3.1 (Type Safety). *If $\cdot \mid \cdot \vdash e : \sigma$ then e either evaluates to a value or diverges.*

We prove this theorem syntactically using the standard lemmas [31]: Preservation (Theorem 3.2) and Progress (Theorem 3.3). Because evaluation involves type constructors, we must also show that these lemmas hold for them [14].

Lemma 3.2 (Preservation).

1. *If $\Delta \vdash \tau : \star^{\ell}$ and $\tau \rightsquigarrow \tau'$ then $\Delta \vdash \tau' : \star^{\ell}$.*
2. *If $\Delta \mid \Gamma \vdash e : \sigma$ and $e \rightsquigarrow e'$ then $\Delta \mid \Gamma \vdash e' : \sigma$.*

$$\frac{\alpha \mapsto R \in \eta \qquad v_1 R v_2}{\eta \vdash v_1 \approx v_2 : \alpha}\ \textsf{lr:var} \qquad \frac{}{\eta \vdash i \approx i : \textsf{int}}\ \textsf{lr:int}$$

$$\frac{\forall(\eta \vdash e_1 \approx e_2 : \sigma_1).\eta \vdash v_1 e_1 \approx v_2 e_2 : \sigma_2}{\eta \vdash v_1 \approx v_2 : \sigma_1 \to \sigma_2}\ \textsf{lr:arr}$$

$$\frac{\begin{array}{c}\eta \vdash \mathbf{fst}\, v_1 \approx \mathbf{fst}\, v_2 : \sigma_1 \\ \eta \vdash \mathbf{snd}\, v_1 \approx \mathbf{snd}\, v_2 : \sigma_2\end{array}}{\eta \vdash v_1 \approx v_2 : \sigma_1 \times \sigma_2}\ \textsf{lr:prod}$$

$$\frac{\begin{array}{l}\forall \sigma_1, \sigma_2.\forall R : \sigma_1 \leftrightarrow \sigma_2. \\ \quad \eta, \alpha \mapsto R \vdash v_1[\sigma_1] \approx v_2[\sigma_2] : \sigma\end{array}}{\eta \vdash v_1 \approx v_2 : \forall \alpha.\sigma}\ \textsf{lr:all}$$

$$\frac{\begin{array}{l}e_1 \rightsquigarrow^* v_1, e_2 \rightsquigarrow^* v_2, \sigma \rightsquigarrow^* \sigma' \implies \\ \quad \eta \vdash v_1 \approx v_2 : \sigma'\end{array}}{\eta \vdash e_1 \approx e_2 : \sigma}\ \textsf{lr:term}$$

$$\frac{(e_1 \Uparrow) \vee (e_2 \Uparrow)}{\eta \vdash e_1 \approx e_2 : \sigma}\ \textsf{lr:divr}$$

$$\frac{\forall \alpha{:}\star \in \Delta.\eta(\alpha) \in \delta_1(\alpha) \leftrightarrow \delta_2(\alpha))}{\eta \vdash \delta_1 \approx \delta_2 : \Delta}\ \textsf{lr:tbase}$$

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma}\ \textsf{lr:base}$$

Figure 6: Logically related terms for parametricity

Proof. By induction on $\Delta \vdash \tau : \star^\ell$ and $\Delta \mid \Gamma \vdash e : \sigma$. □

Lemma 3.3 (Progress).

1. *If* $\cdot \vdash \tau : \star^\ell$ *then* τ *is in weak-head normal form or* $\tau \rightsquigarrow \tau'$.
2. *If* $\cdot \mid \cdot \vdash e : \sigma$ *then* e *is a value or* $e \rightsquigarrow e'$.

Proof. By induction on $\cdot \vdash \tau : \star^\ell$ and $\cdot \mid \cdot \vdash e : \sigma$. □

4 Generalizing Parametricity

Parametricity has long been used to reason about programs in languages with parametric polymorphism. It can be used for a variety of purposes: to show that two different implementations of an abstract datatype do not influence the behavior of the program, to show that external modules cannot forge values of abstract types, or to show that certain components of an abstract type cannot be determined by external observers, are only a few of the corollaries of the powerful parametricity theorem. In this section, we show how the parametricity theorem for languages with parametric polymorphism may be generalized to λ_{iSEC}. We begin with a little background: a presentation of the standard parametricity theorem for the parametric core of λ_{iSEC} and its proof. After setting up that scaffolding, we are in a better position to describe our contributions.

$$\frac{\tau \rightsquigarrow \tau'}{(\tau)^\ell \rightsquigarrow (\tau')^\ell} \qquad \frac{}{(\tau_1 \to \tau_2)^\ell \rightsquigarrow (\tau_1)^\ell \xrightarrow{\ell} (\tau_2)^\ell}$$

$$\frac{}{(\tau_1 \times \tau_2)^\ell \rightsquigarrow (\tau_1)^\ell \times^\ell (\tau_2)^\ell}$$

Figure 7: Type reduction

4.1 Background

The core of λ_{iSEC}, without type analysis operators or security labels, is a simple polymorphic lambda calculus. (Although it is predicative, that does not matter much.) Informally, the parametricity theorem states that well-typed expressions, after related substitutions for their free type and term variables, are related. The power of this theorem comes from the fact that any pair of types may be related, but related terms must have similar evaluation behavior. (We explain what this means more formally below.) In Section 4.2 we show how the parametricity theorem may be use to formally derive some of the properties mentioned above.

The relation between closed terms mentioned in the parametricity theorem is defined in Figure 6. Our version of this relation is termination insensitive: Two related terms either both produce related values (rule lr:term) or one of them diverges (rule lr:divr). We write $e \Uparrow$ here to mean that term e diverges. (It is possible to define the relation so that only divergent values are related, but that requires more work.)

The definition of related values is indexed by a type σ. For simplicity, we only define this relation for types that are in *normal form*—all constructors that appear in the types have been reduced as much as possible, and have been replaced by corresponding types. The rule lr:term also allows type reduction, using the rules in Figure 7.

The (normalized) type σ that indexes the value relation may contain free type variables. To account for those variables, the definition of the relation is parameterized by η, a map between type variables and relations between values. This map is used when σ is a type variable (see rule lr:var). If σ is int, the relation is the identity relation. As is typical for logical relations, values of function type are related if, when applied to related arguments, they produce related results. Likewise, values of product types are related if their corresponding components are related. The most important part of this definition is the relation between values of type $\forall\alpha.\sigma$. Two polymorphic values are related if their instantiations with *any* pair of types are related. Furthermore, we can use *any* relation R between values of those types as the relation for α. (We use the notation $\mathsf{R} : \sigma_1 \leftrightarrow \sigma_2$ to mean that R is a relation between values of type σ_1 and of type σ_2.) It is because of this definition that we restrict polymorphism to variables of base kind, so that we do not have to define relations for variables of higher kind.

Before we can state the parametricity theorem we must define related substitutions for terms and types. The rule $\mathsf{lr{:}tbase}$ states that η is well-formed with respect to two type substitutions δ_1 and δ_2 for the variables in the type context Δ. There are no restrictions on the range of the type substitutions. On the other hand, a pair of term substitutions for the term variables in Γ must only map to related expressions.

With these definitions, we can state the parametricity theorem for this core language in the following way:

Theorem 4.1 (Parametricity for the polymorphic lambda calculus). *If* $\Delta; \Gamma \vdash e : \sigma$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \Delta$ *and* $\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$ *then then* $\eta \vdash \delta_1(\gamma_1(e)) \approx \delta_2(\gamma_2(e)) : \sigma$.

The proof of the parametricity theorem is by induction on the typing judgement $\Delta; \Gamma \vdash e : \sigma$. There are a few tricky parts to this proof that we cover below.

The primary complication of this proof arises in the case for type application, where we would like to show that a term $v[\sigma']$ is related to itself (after appropriate substitutions) at type $\sigma[\sigma'/\alpha]$. By induction, we know that v is related to itself at type $\forall\alpha.\sigma$, so by the rule $\mathsf{lr{:}all}$ we may conclude that $v[\sigma']$ is related to itself a type s, where the type αmaps to any relation. By choosing the relation $\eta \vdash \bullet \approx \bullet : \tau$ we can use the following substitution lemma to show the result.

Lemma 4.2 (Type substitution for parametricity). $\eta, \alpha \mapsto \mathsf{R} \vdash e_1 \approx e_2 : \sigma$ *if and only if* $\eta \vdash e_1 \approx e_2 : \sigma[\tau/\alpha]$ *where* R *is the relation* $\eta \vdash \bullet \approx \bullet : \tau$.

4.2 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 lambda calculus, just by looking at their types [29].

Our purpose is more similar to Reynolds original purpose: showing how to reason about program equivalence in the presence of type abstraction. However, while Reynolds saw the need to separate parametric polymorphism from ad-hoc polymorphism, here we show how to generalize his work to encompass both sorts of polymorphism.

Corollaries of Reynolds' parametricity theorem provide important results for reasoning about abstract types in programs. There are many specific properties that can be proven as a consequence of parametricity, but we believe the following two are fairly representative of what a programmer desires.

Corollary 4.3 (Type independence). *If* $\alpha{:}\star \mid \cdot \vdash e : \mathsf{int}$ *then for any* $\cdot \vdash \tau_1 : \star$ *and* $\cdot \vdash \tau_2 : \star$ *if* $e[\tau_1/\alpha]$ *and* $e[\tau_2/\alpha]$ *both terminate, they will produce the same value.*

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.

Corollary 4.4 (No forgery). *If* $\alpha{:}\star \mid \cdot \vdash e : \alpha$ *then* e *must diverge.*

This second corollary states that there is no way for a program to invent values of an abstract type.

However, in the presence of run-time type analysis it is clear that these corollaries do not hold. Consider the term (eliding labels).

$$\textbf{typecase}\ [\gamma.\mathsf{int}]\ \alpha\ 1\ (\Lambda\alpha{:}\star.\Lambda\beta{:}\star.2)\ (\Lambda\alpha{:}\star.\Lambda\beta{:}\star.3)$$

This term violates Corollary 4.3 as the result of substituting in int and $\mathsf{int} \times \mathsf{int}$ for α will produce unrelated terms. Instantiating with the first type will result in a term that evaluates to 1 versus one that produces 3 with the second type.

We would like to state similar properties about λ_{iSEC}, but we cannot because the parametricity theorem does not hold. In the next section we show how to generalize this theorem so that we can derive similar corollaries.

Looking at the parametricity proof, we cannot add a case for **typecase**, because we would have to show that the two terms would produce related results, even though they analyze on arbitrary constructors. Furthermore, λ_{iSEC} presents another problem: the normal forms of types also include (for example) $\mathsf{Typerec}$ when its argument is variable. Therefore, we must also extend our definition of the logical relation to these sorts of types.

We solve these two problems as follows: For **typecase**, we much change the definition of the relation for polymorphic types and require that the type constructor arguments be related to each other, as defined in the next section. The labels on the types and

$$\frac{\ell_2 \not\sqsubseteq \ell_1}{v_1 \approx_{\ell_1} v_2 : \star^{\ell_2}}\ \textsf{tslr:type-opaq}$$

$$\frac{\ell_2 \sqsubseteq \ell_1}{\mathsf{int} \approx_{\ell_1} \mathsf{int} : \star^{\ell_2}}\ \textsf{tslr:type-int}$$

$$\frac{\begin{array}{cc}\ell_2 \sqcup \ell_3 \sqsubseteq \ell_4 & \ell_4 \sqsubseteq \ell_1\\ \tau_1 \approx_{\ell_1} \tau_3 : \star^{\ell_2} & \tau_2 \approx_{\ell_1} \tau_4 : \star^{\ell_3}\end{array}}{\tau_1 \to \tau_2 \approx_{\ell_1} \tau_3 \to \tau_4 : \star^{\ell_4}}\ \textsf{tslr:type-arr}$$

$$\frac{\begin{array}{cc}\ell_2 \sqcup \ell_3 \sqsubseteq \ell_4 & \ell_4 \sqsubseteq \ell_1\\ \tau_1 \approx_{\ell_1} \tau_3 : \star^{\ell_2} & \tau_2 \approx_{\ell_1} \tau_4 : \star^{\ell_3}\end{array}}{\tau_1 \times \tau_2 \approx_{\ell_1} \tau_3 \times \tau_4 : \star^{\ell_4}}\ \textsf{tslr:type-prod}$$

$$\frac{\forall(\tau_1' \approx_{\ell_1} \tau_2' : \kappa_1).\tau_1\tau_1' \approx_{\ell_1} \tau_2\tau_2' : \kappa_2 \sqcup \ell_2}{\tau_1 \approx_{\ell_1} \tau_2 : \kappa_1 \xrightarrow{\ell_2} \kappa_2}\ \textsf{tslr:arr}$$

$$\frac{\tau_1 \rightsquigarrow^* v_1 \qquad \tau_2 \rightsquigarrow^* v_2 \qquad v_1 \approx_\ell v_2 : \star^\ell}{\tau_1 \approx_\ell \tau_2 : \star^\ell}\ \textsf{tsclr:base}$$

Figure 8: Logical relation over type constructors

$$\frac{\alpha \mapsto R \in \eta \qquad \ell_2 \sqsubseteq \ell_1 \implies v_1 R_\rho^{\ell_2} v_2}{\eta \vdash v_1 \approx_\ell v_2 : (\rho\{\alpha\})^{\ell_2}}\ \textsf{slr:con}$$

$$\frac{\ell_2 \sqsubseteq \ell_1 \implies i_1 = i_2}{\eta \vdash i_1 \approx_{\ell_1} i_2 : (\mathsf{int})^{\ell_2}}\ \textsf{slr:int}$$

$$\frac{\begin{array}{l}\forall(\eta \vdash e_1 \approx_{\ell_1} e_2 : \sigma_1).\\ \quad \eta \vdash v_1 e_1 \approx_{\ell_1} v_2 e_2 : \sigma_2 \sqcup \ell_2\end{array}}{\eta \vdash v_1 \approx_{\ell_1} v_2 : \sigma_1 \xrightarrow{\ell_2} \sigma_2}\ \textsf{slr:arr}$$

$$\frac{\begin{array}{c}\eta \vdash \mathbf{fst}\, v_1 \approx_{\ell_1} \mathbf{fst}\, v_2 : \sigma_1 \sqcup \ell_2\\ \eta \vdash \mathbf{snd}\, v_1 \approx_{\ell_1} \mathbf{snd}\, v_2 : \sigma_2 \sqcup \ell_2\end{array}}{\eta \vdash v_1 \approx_{\ell_1} v_2 : \sigma_1 \times^{\ell_2} \sigma_2}\ \textsf{slr:prod}$$

$$\frac{\begin{array}{l}\forall(\tau_1 \approx_{\ell_1} \tau_2 : \star^{\ell_3}).\forall(R_\rho^\ell \in \rho\{\tau_1\}^\ell \leftrightarrow \rho\{\tau_2\}^\ell).\\ \quad \eta, \alpha \mapsto R \vdash v_1[\tau_1] \approx_{\ell_1} v_2[\tau_2] : \sigma \sqcup \ell_2\end{array}}{\eta \vdash v_1 \approx_{\ell_1} v_2 : \forall^{\ell_2} \alpha{:}\star^{\ell_3}.\sigma}\ \textsf{slr:all}$$

$$\frac{\begin{array}{l}e_1 \rightsquigarrow^* v_1, e_2 \rightsquigarrow^* v_2, \sigma \rightsquigarrow^* \sigma' \implies\\ \quad \eta \vdash v_1 \approx_\ell v_2 : \sigma\end{array}}{\eta \vdash e_1 \approx_\ell e_2 : \sigma}\ \textsf{sclr:term}$$

$$\frac{(e_1 \Uparrow) \vee (e_2 \Uparrow)}{\eta \vdash e_1 \approx_\ell e_2 : \sigma}\ \textsf{sclr:divr}$$

Figure 9: Logical relation over terms

kinds allow us to have some leeway in this definition, so that it does not need to be an identity relation when types are used parametrically. For Typerec we expand on the trick of quantifying over all relations between terms of the correct type, to quantifying over all families of relations between terms of the correct type under any enclosing type context.

In the next subsections we discuss these solutions in more detail.

4.3 Related constructors

The first step towards proving our generalized parametricity theorem is to define what it means for two type constructors to be related, as we do in Figure 4.3. We write $\tau_1 \approx_\ell \tau_2 : \kappa$ to mean two closed constructors are equivalent at kind κ with respect to an observer at level ℓ in the label lattice. If ℓ is greater than the security level of the kind κ, then the two constructors must be identical. Otherwise, the rule tslr:type-opaq relates any two type constructors of the appropriate kind. Intuitively, if the constructors carry information more restrictive than the level of the observer, the observer shouldn't be able to tell them apart. For example, $\cdot \vdash \mathsf{int} : \star^{\mathsf{H}}$ and $\cdot \vdash \mathsf{int} \times \mathsf{int} : \star^{\mathsf{H}}$ which carry "high-security" information H, will be indistinguishable to an observer at a "low-security" level L.

As a sanity check we can show that if two constructors are related at kind κ, they are related at all superkinds of κ.

Lemma 4.5 (Constructor relation closed under subsumption). *If* $\tau_1 \approx_\ell \tau_2 : \kappa_1$ *and* $\kappa_1 \leq \kappa_2$ *then* $\tau_1 \approx_\ell \tau_2 : \kappa_2$.

Proof. By induction over $\tau_1 \approx_\ell \tau_2 : \kappa_1$. □

The other three rules for kind $\star$ state that if two primitive constructors are related at $\star^{\ell_2}$ where ℓ_2 is a label that is less than the observer ℓ_1, their components must appear related to the observer.

For constructors of base kind that are not in normal form, we use the rule tcslr:base. This rule says that two constructors are related if and only if their weak-head normal forms are related.

The rule for type functions, tslr:arr, is standard for logical relations. Two functions are considered related if for any two related constructors of the correct type, their application is related.

$$\textit{(type contexts)}\quad \rho \ ::= \ \bullet \mid \mathsf{Typerec}\ \rho\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} \mid \rho\,\tau$$

Figure 10: The grammar of type contexts

4.4 Related terms

We use the notation $\eta \vdash e_1 \approx_\ell e_2 : \sigma$ to indicate that terms e_1 and e_2 are equivalent to an observer at level ℓ at type σ, with the relation mapping η. This relation is defined in Figure 9.

Like the relation for type constructors, if the security level of the observer is not greater than or equal that of the type σ, then e_1 and e_2 may be arbitrarily related. However, instead of adding a rule analogous to tslr:type-opaque, we show that this property holds of our relation as a metatheorem. (The definition of related type substitutions $\eta \vdash \delta_1 \approx_\ell \delta_2 : \Delta$ appears at the beginning of the next section.)

Lemma 4.6 (Indistiguishability of secure objects). *If* $\Delta \mid \cdot \vdash e_1 : \sigma$ *and* $\Delta \mid \cdot \vdash e_2 : \sigma$ *and* $\mathcal{L}(\sigma) \not\sqsubseteq \ell$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \Delta$ *then* $\eta \vdash \delta_1(e_1) \approx_\ell \delta_2(e_2) : \sigma$.

Proof. By induction over σ. □

We can also show that the term relation is closed under subsumption.

Lemma 4.7 (Term relation closed under subsumption). *If* $\eta \vdash e_1 \approx_\ell e_2 : \sigma_1$ *and* $\Delta \vdash \sigma_1 \leq \sigma_2$ *then* $\eta \vdash e_1 \approx_\ell e_2 : \sigma_2$.

Proof. By induction over $\eta \vdash e_1 \approx_\ell e_2 : \sigma_1$. □

We solve the problem with Typerec by changing the mapping η. In this definition, the mapping η takes type variables to *families* of binary relations between values. These families are indexed by a type context ρ and a security level. The shape of type contexts is described in Figure 10. Contexts are either holes $\bullet$, Typerecs of a context, or a context applied to any arbitrary type constructor. A context does not contain any free type variables. We write $\rho\{\tau\}$ for filling a context's hole with τ. Before, where we wrote $R : \tau_1 \leftrightarrow \tau_2$, we now require that each relation, R^{l}_{ρ}, is between values of type $(\rho\{\tau_1\})^{\ell}$ and $(\rho\{\tau_2\})^{\ell}$.

4.5 Generalized Parametricity

Before we can state the generalized parametricity theorem, we must describe related type and term substitutions. The type substitutions δ_1 and δ_2 are intimately connected with the relation mapping η. The type variable context Δ describes the domain of all three of these maps.

$$\frac{\begin{array}{c}\forall \alpha{:}\star^{\ell_2} \in \Delta.(\eta(\alpha)^{\ell_3}_{\rho} \in \delta_1(\rho\{\alpha\})^{\ell_3} \leftrightarrow \delta_2(\rho\{\alpha\})^{\ell_3}) \\ \forall \alpha{:}\star^{\ell_2} \in \Delta.(\delta_1(\alpha) \approx_{\ell_1} \delta_2(\alpha) : \star^{\ell_2})\end{array}}{\eta \vdash \delta_1 \approx_{\ell_1} \delta_2 : \Delta}$$

Not only must η be a family of relations for the types that δ_1 and δ_2 map, but the types that they map must be related according to our definition in Figure 4.3.

Two term substitutions γ_1 and γ_2 are considered related if they map variables to related terms, defined with the following inference rule:

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx_\ell \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx_\ell \gamma_2 : \Gamma}$$

Now we may state our generalized parametricity theorem. This time the theorem comes in two parts: stating that both type constructors and terms are related to themselves, after any related substitutions.

Theorem 4.8 (Generalized parametricity).

1. *If* $\Delta \vdash \tau : \kappa$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_\ell \delta_2(\tau) : \kappa$.
2. *If* $\Delta \mid \Gamma \vdash e : \sigma$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \Delta$ *and* $\eta \vdash \gamma_1 \approx_\ell \gamma_2 : \Gamma$ *then* $\eta \vdash \delta_1(\gamma_1(e)) \approx_\ell \delta_2(\gamma_2(e)) : \sigma$.

After giving more details of the proof of this theorem below, we discuss applications of this theorem in the next section.

The proof for the first part proceeds by straightforward structural induction upon $\Delta \vdash \tau : \kappa$. The only difficulty is the primitive recursive nature of Typerec. As a consequence the case for wfc:trec requires the use of the following Lemma 4.9.

Lemma 4.9 (Relation closed under Typerec).
If $\tau \approx_{\ell_1} \tau' : \star^{\ell_2}$
and $\tau_{\mathsf{int}} \approx_{\ell_1} \tau'_{\mathsf{int}} : \star^{\ell} \xrightarrow{\ell_2} \kappa \xrightarrow{\ell_2} \kappa$
and $\tau_{\rightarrow} \approx_{\ell_1} \tau'_{\rightarrow} : \star^{\ell} \xrightarrow{\ell_2} \star^{\ell} \xrightarrow{\ell_2} \kappa \xrightarrow{\ell_2} \kappa \xrightarrow{\ell_2} \kappa$
and $\tau_{\times} \approx_{\ell_1} \tau'_{\times} : \star^{\ell} \xrightarrow{\ell_2} \star^{\ell} \xrightarrow{\ell_2} \kappa \xrightarrow{\ell_2} \kappa \xrightarrow{\ell_2} \kappa$
then $\mathsf{Typerec}\ \tau\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} \approx_{\ell_1} \mathsf{Typerec}\ \tau'\ \tau'_{\mathsf{int}}\ \tau'_{\rightarrow}\ \tau'_{\times} : \kappa$.

Proof. If $\ell_2 \not\sqsubseteq \ell_1$ then trivial by Lemma 4.6, otherwise by induction on $\tau \approx_{\ell_1} \tau' : \star^{\ell_2}$. □

The second part of the proof of the generalized parametricity theorem also proceeds by structural induction, this time on $\Delta \mid \Gamma \vdash e : \sigma$. The proof is a bit more complicated, though most cases are still straightforward. As in the standard proof of parametricity, we need a type substitution for case wft:tapp. The following lemma is a generalization of Lemma 4.2.

Lemma 4.10 (Type substitution). $\eta, \alpha \mapsto (\equiv) \vdash e_1 \approx_\ell e_2 : \sigma$ *if and only if* $\eta \vdash e_1 \approx_\ell e_2 : \sigma[\tau/\alpha]$ *where* $\equiv_\rho^{\ell'}$ *is the relation* $\eta \vdash \bullet \approx_\ell \bullet : (\rho\{\tau\})^{\ell'}$.

Proof. By induction on $\eta, \alpha \mapsto (\equiv) \vdash e_1 \approx_\ell e_2 : \sigma$. □

The most complicated part of the generalized parametricity proof is the case for wft:tcase. It first requires making use of the first part of Theorem 4.8, and then a nested induction on the resulting relation, making use of the type substitution lemma above.

4.6 Applications of Generalized Parametricity

A typical corollary of Theorem 4.8 is normally called noninterference – that is it is possible to substitute values indistinguishable to the present observer and get indistinguishable results follows as an immediate consequence.

Corollary 4.11 (Noninterference for terms). *If* $\cdot \mid \cdot, x{:}\sigma_1 \vdash e : \sigma_2$ *and* $\cdot \vdash v_1 \approx_\ell v_2 : \sigma_1$ *where* $\ell \sqsubseteq \mathcal{L}(\sigma_1)$ *then* $\cdot \vdash e[v_1/x] \approx_\ell e[v_2/x] : \sigma_2$.

Proof. Follows directly from Theorem 4.8. □

However, more importantly it is also possible to restate the corollaries of we showed for standard parametricity. Here we consider them in the context of a two point lattice with low (L) and high (H) security labels.

Corollary 4.12 (Representation independence). *If* $\alpha{:}\star^{\mathsf{H}} \mid \cdot \vdash e : \mathsf{int}^{\mathsf{L}}$ *then for any* $\cdot \vdash \tau_1 : \star^{\mathsf{L}}$ *and* $\cdot \vdash \tau_2 : \star^{\mathsf{L}}$ *if* $e[\tau_1/\alpha]$ *and* $e[\tau_2/\alpha]$ *both terminate, they will produce the same value.*

Corollary 4.13 (No forgery). *If* $\alpha{:}\star^{\mathsf{H}} \mid \cdot \vdash e : \alpha^{\mathsf{L}}$ *then* e *must diverge.*

In fact, these lemmas follow because Theorem 4.8 is a generalization of Theorem 4.1. By choosing the labels of kinds and types, and the label of the observer correctly, we can make the relation in Figure 9 be the same as the relation in Figure 6.

5 Label polymorphism

Implementations of information-flow languages, such as FlowCaml [22] and Jif [17], require many more language features than are present in λ_{iSEC} to make them suitable for practical use. To keep the language and proofs simple in the previous sections, we have not included features such as bounded label polymorphism. However, bounded label polymorphism is critical to a realistic information flow type system because it allows the type system to track information flows through the program, without knowing (or restricting) ahead of time the information contexts in which functions will be called.

labels	ℓ	$::=$	$\ldots \mid \iota$
types	σ	$::=$	$\ldots \mid \forall^{\ell_1} \iota \sqsubseteq \ell_2.\sigma$
terms	e	$::=$	$\ldots \mid \Lambda\iota \sqsubseteq l.e \mid e[\ell]$
label context	Ω	$::=$	$\cdot \mid \Omega, \iota \sqsubseteq \ell$

Figure 11: Bounded label polymorphism extension

Below, we show how this feature may be added to λ_{iSEC}, and the associated modifications that must be made to the generalized parametricity theorem.

The additions to the calculus for label polymorphism (in Figure 11 include extending the syntax of labels to include label variables, ι, extending types to include the types of label polymorphic terms, $\forall^{\ell_1} \iota \sqsubseteq \ell_2.\sigma$, and adding terms for label abstraction and application. Furthermore the static semantics of this calculus must include a context Ω that records the label variables in scope and their bounds. This context is necessary to check whether types and terms are well-formed (by ensuring that all label variables are bound). It also is necessary to determine the relative positions of labels in the lattice (i.e. $\ell_1 \subseteq \ell_2$) because one or both of these labels may be a variable.

To update the generalized parametricity theorem to include label polymorphism, we must define when two label-polymorphic values are related. Two values are related if they always produce related results, no matter what label they are instantiated with.

$$\frac{\forall \ell \sqsubseteq \ell_3.(\eta \vdash v_1[\ell] \approx_{\ell_1} v_2[\ell] : \sigma[\ell/\iota] \sqcup \ell_2)}{\eta \vdash v_1 \approx_{\ell_1} v_2 : \forall^{\ell_2} \iota \sqsubseteq \ell_3.\sigma}$$

We can restate the generalized parametricity theorem (4.8) as follows: Suppose ω is a substitution from label variables in the domain of Ω to closed labels, such that their bounds are satisfied. Then we apply this substitution uniformly to all contexts, kinds, constructors, types and terms that may contain free label variables. More formally:

Theorem 5.1 (Generalized parametricity for label polymorphism).

1. *If* $\Omega \mid \Delta \vdash \tau : \kappa$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \omega(\Delta)$ *then* $\delta_1(\omega(\tau)) \approx_\ell \delta_2(\omega(\tau)) : \omega(\kappa)$.
2. *If* $\Omega \mid \Delta \mid \Gamma \vdash e : \sigma$ *and* $\eta \vdash \delta_1 \approx_\ell \delta_2 : \omega(\Delta)$ *and* $\eta \vdash \gamma_1 \approx_\ell \gamma_2 : \omega(\Gamma)$ *then* $\eta \vdash \delta_1(\gamma_1(\omega(e))) \approx_\ell \delta_2(\gamma_2(\omega(e))) : \omega(\sigma)$.

6 Related work

Our design of λ_{iSEC} draws heavily upon previous work on type analysis, parametricity, information flow, and data integrity analysis.

Most information flow models use a lattice model that originates from work by Bell and LaPadula [4] and Denning [5]. The earliest work on static information flow dates back to Denning and Denning [6]. Palsberg and Ørbæk examined trust and integrity with a simple extension of the λ-calculus [21]. Heintze and Riecke's formalized information-flow and integrity in a typed λ-calculus with references, the SLam calculus [10], and proved a number of soundness and noninterference results. Volpano et al. [28] showed how to formulate Denning's work as type system and proved its soundness with respect to noninterference. Volpano and Smith later developed a type inference system for information flow with noninterference [27].

Recognizing that strict noninterference is too strong for practical programming, Myers and Liskov build upon previous work in information flow with a decentralized declassification model [18, 19, 16, 20] that allows principals to leak information in a controlled fashion. Out of this research extensions to the Java [7] language and type system were developed, first in JFlow [15], and then Jif [17]. Research on extending Jif with full integrity constraints is in progress [13].

Other researchers have noticed the connection between parametricity and non-interference. Essentially a non-interference result shows that high-security terms are parametric with respect to low security observers. With this observation, Tse and Zdancewic [26] use parametricity to *encode* to prove non-interference, by encoding Abadi, et al.'s [1] dependency core calculus and System $\mathbb{F}$.

There is considerable research into run-time type analysis in programming languages. Abadi, et al. introduced a type "dynamic" to which types could be coerced, and later via case analysis, extracted [3]. The core semantics of **typecase** in λ_{iSEC} is adopted directly from the intensional polymorphism of Harper and Morrisett [9]. Trifonov, Saha, and Shao extended Harper and Morrisett's language to be fully-reflexive [25] by adding kind polymorphism. Weirich [30] extended run-time analysis to higher-order type constructors [12].

The fact that run-time type analysis (as well as other forms of ad-hoc polymorphism) breaks parametricity has been long understood, but little has been done to reconcile the two. Leifer et al. [**?**] design a system that preserves type abstraction through marshalling, which is implemented internally through type analysis. However, they do not allow users to define their own marshalling routines, or any other type-indexed operations. Rossberg [8] uses generative types to hide type information in the presence of run-time analysis, relying on colored-brackets [8] to provide easy access. However, he does not discuss the formal abstraction properties that his system provides.

7 Conclusion

With λ_{iSEC}, we address the conflict between intensional type analysis and enforceable abstractions. Software developers can specify with a fine degree of control who may inspect their abstractions, and what they can do with the information they learn by doing so.

However, this fine degree of control comes at with the penalty of having to write many annotations in order for a program to type check. We have not investigated how tedious these annotations will prove in practice. Although existing large scale languages, such as Jif [17] and FlowCaml [22], implement some form of information flow inference, they can be difficult to use. On the other hand, languages based on λ_{iSEC} have the advantage that if the only goal is to secure type level abstractions and no type analysis is performed, then no information flow-annotations should be necessary. However, isolated uses of type analysis could result in far reaching information flow interactions. Therefore it will be imperative when developing a high level language based around λ_{iSEC} that the cost of maintaining the necessary annotations does not outweigh the benefits conferred by type directed programming.

Acknowledgements

This paper benefitted greatly from conversations with Steve Zdancewic and Stephen Tse.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, Jan. 1999.

[2] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1–2):9–58, 6 Dec. 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[4] 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. Available as NTIS AD-A023 588.

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

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

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Aug. 1996. ISBN 0-201-63451-1.

[8] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, Nov. 2000.

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[10] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, 1998.

[11] M. Hicks, S. Weirich, and K. 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.

[12] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue.

[13] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, Sept. 2003.

[14] G. Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, Dec. 1995.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, Jan. 1999.

[16] A. C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1999. Ph.D. thesis.

[17] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif.

[18] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[19] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

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

[21] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, Sept. 1995.

[22] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[23] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[24] A. Rossberg. Generativity and dynamic opacity for abstract types. In D. Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, Aug. 2003. ACM Press. Extended version available from `http://www.ps.uni-sb.de/Papers/generativity-extended.html`.

[25] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, Sept. 2000.

[26] S. Tse and S. Zdancewic. Translating dependency into parametricity. Technical report, University of Pennsylvania, Jan. 2004.

[27] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT '97, Colloquium on Formal Approaches to Software Engineering*, Lille, France, Apr. 1997.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[29] P. Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.

[30] S. Weirich. Higher-order intensional type analysis. In D. L. Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, 2002.

[31] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

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

A λ_{iSEC} rules

A.1 Static semantics

A.1.1 Sub-kinding

$$\frac{}{\kappa \leq \kappa}\ \textsf{sbk:refl} \qquad \frac{\kappa_1 \leq \kappa_2 \qquad \kappa_2 \leq \kappa_3}{\kappa_1 \leq \kappa_3}\ \textsf{sbk:trans}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\star^{\ell_1} \leq \star^{\ell_2}}\ \textsf{sbk:type}$$

$$\frac{\kappa_3 \leq \kappa_1 \qquad \kappa_2 \leq \kappa_4 \qquad \ell_1 \sqsubseteq \ell_2}{\kappa_1 \xrightarrow{\ell_1} \kappa_2 \leq \kappa_3 \xrightarrow{\ell_2} \kappa_4}\ \textsf{sbk:arr}$$

A.1.2 Constructor well-formedness

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{wfc:var} \qquad \frac{}{\Delta \vdash \textsf{int} : \star^{\perp}}\ \textsf{wfc:int}$$

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

$$\frac{\Delta \vdash \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_1 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 : \star^{\ell_1 \sqcup \ell_2}}\ \textsf{wfc:prod}$$

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

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

$$\frac{\begin{array}{c}\Delta \vdash \tau : \star^{\ell} \qquad \ell \sqsubseteq \ell' \qquad \Delta \vdash \tau_{\textsf{int}} : \kappa \\ \Delta \vdash \tau_{\to} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \Delta \vdash \tau_{\times} : \star^{\ell} \xrightarrow{\ell'} \star^{\ell} \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \\ \text{where } \ell' = \mathcal{L}(\kappa)\end{array}}{\Delta \vdash \textsf{Typerec}\ \tau\ \tau_{\textsf{int}}\ \tau_{\to}\ \tau_{\times} : \kappa}\ \textsf{wfc:trec}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \qquad \kappa_1 \leq \kappa_2}{\Delta \vdash \tau : \kappa_2}\ \textsf{wfc:sub}$$

A.1.3 Constructor equivalence

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau = \tau : \kappa}\ \textsf{eqc:refl}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa}{\Delta \vdash \tau_1 = \tau_3 : \kappa}\ \textsf{eqc:trans}$$

$$\frac{\Delta \vdash \tau_2 = \tau_1 : \kappa}{\Delta \vdash \tau_1 = \tau_2 : \kappa}\ \textsf{eqc:sym}$$

$$\frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \to \tau_2 = \tau_3 \to \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \textsf{eqc:arr}$$

$$\frac{\Delta \vdash \tau_3 = \tau_1 : \star^{\ell_1} \qquad \Delta \vdash \tau_2 = \tau_4 : \star^{\ell_2}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_3 \times \tau_4 : \star^{\ell_1 \sqcup \ell_2}}\ \textsf{eqc:prod}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 = \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \xrightarrow{\perp} \kappa_2}\ \textsf{eqc:abs-con}$$

$$\frac{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 : \kappa_2}{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2}\ \textsf{eqc:abs-beta}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 = \tau_3 : \kappa_1 \xrightarrow{\ell} \kappa_2 \\ \Delta \vdash \tau_2 = \tau_3 : \kappa_1\end{array}}{\Delta \vdash \tau_1\tau_2 = \tau_3\tau_4 : \kappa_2 \sqcup \ell}\ \textsf{eqc:app}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 = \tau_2 : \star^{\ell} \qquad \Delta \vdash \tau_{\mathsf{int}} = \tau'_{\mathsf{int}} : \kappa \\ \Delta \vdash \tau_{\rightarrow} = \tau'_{\rightarrow} : \star^{\ell} \xrightarrow{\mathcal{L}(\kappa)} \star^{\ell} \xrightarrow{\mathcal{L}(\kappa)} \kappa \xrightarrow{\mathcal{L}(\kappa)} \kappa \xrightarrow{\mathcal{L}(\kappa)} \kappa \\ \Delta \vdash \tau_{\times} = \tau'_{\times} : \star^{\ell} \xrightarrow{\mathcal{L}(\kappa)} \star^{\ell} \xrightarrow{\mathcal{L}(\kappa)} \kappa \xrightarrow{\mathcal{L}(\kappa)} \kappa \xrightarrow{\mathcal{L}(\kappa)} \kappa \\ \ell \sqsubseteq \mathcal{L}(\kappa)\end{array}}{\Delta \vdash \begin{array}{l}\mathsf{Typerec}\ \tau_1 \\ \quad \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times}\end{array} = \begin{array}{l}\mathsf{Typerec}\ \tau_2 \\ \quad \tau'_{\mathsf{int}}\ \tau'_{\rightarrow}\ \tau'_{\times}\end{array} : \kappa}\ \textsf{eqc:trec-con}$$

$$\frac{\Delta \vdash \mathsf{Typerec}\ \mathsf{int}\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} : \kappa}{\Delta \vdash \mathsf{Typerec}\ \mathsf{int}\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} = \tau_{\mathsf{int}} : \kappa}\ \textsf{eqc:trec-int}$$

$$\frac{\Delta \vdash \mathsf{Typerec}\ (\tau_1 \rightarrow \tau_2)\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} : \kappa}{\Delta \vdash \begin{array}{l}\mathsf{Typerec} \\ (\tau_1 \rightarrow \tau_2) \\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times}\end{array} = \begin{array}{l}\tau_{\rightarrow}\tau_1\tau_2 \\ (\mathsf{Typerec}\ \tau_1 \\ \quad \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times}) \\ (\mathsf{Typerec}\ \tau_2 \\ \quad \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times})\end{array} : \kappa}\ \textsf{eqc:trec-arr}$$

$$\frac{\Delta \vdash \mathsf{Typerec}\ (\tau_1 \times \tau_2)\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times} : \kappa}{\Delta \vdash \begin{array}{l}\mathsf{Typerec} \\ (\tau_1 \times \tau_2) \\ \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times}\end{array} = \begin{array}{l}\tau_{\times}\tau_1\tau_2 \\ (\mathsf{Typerec}\ \tau_1 \\ \quad \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times}) \\ (\mathsf{Typerec}\ \tau_2 \\ \quad \tau_{\mathsf{int}}\ \tau_{\rightarrow}\ \tau_{\times})\end{array} : \kappa}\ \textsf{eqc:trec-prod}$$

A.1.4 Type well-formedness

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

$$\frac{\Delta \vdash \sigma_1 \qquad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \times^{\ell} \sigma_2}\ \textsf{wftp:prod}$$

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

A.1.5 Subtyping

$$\frac{\Delta \vdash \sigma}{\Delta \vdash \sigma \le \sigma}\ \textsf{sbt:refl}$$

$$\frac{\Delta \vdash \sigma_1 \le \sigma_2 \qquad \Delta \vdash \sigma_2 \le \sigma_3}{\Delta \vdash \sigma_1 \le \sigma_3}\ \textsf{sbt:trans}$$

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

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

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

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

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

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

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

$$\frac{\Delta, \alpha{:}\star^{\ell_4} \vdash \sigma_1 \le \sigma_2 \qquad \ell_4 \sqsubseteq \ell_2 \qquad \ell_1 \sqsubseteq \ell_3}{\Delta \vdash \forall^{\ell_1} \alpha{:}\star^{\ell_2}.\sigma_1 \le \forall^{l_3} \alpha{:}\star^{\ell_4}.\sigma_2}\ \textsf{sbt:all}$$

A.1.6 Term well-formedness

$$\frac{}{\Delta \mid \Gamma \vdash i : \mathsf{int}^{\perp}}\ \textsf{wft:int} \qquad \frac{x : \sigma \in \Gamma}{\Delta \mid \Gamma \vdash x : \sigma}\ \textsf{wft:var}$$

$$\frac{\Delta \mid \Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \qquad \Delta \vdash \sigma_1}{\Delta \mid \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \xrightarrow{\perp} \sigma_2}\ \textsf{wft:abs}$$

$$\frac{\Delta \mid \Gamma \vdash e_1 : \sigma_1 \xrightarrow{\ell} \sigma_2 \qquad \Delta \mid \Gamma \vdash e_2 : \sigma_1}{\Delta \mid \Gamma \vdash e_1 e_2 : \sigma_2 \sqcup \ell}\ \textsf{wft:app}$$

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

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

$$\frac{\Delta \mid \Gamma \vdash e_1 : \sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : \sigma_2}{\Delta \mid \Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times^{\perp} \sigma_2}\ \textsf{wft:pair}$$

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta \mid \Gamma \vdash \mathbf{fst}\, e : \sigma_1 \sqcup \ell}\ \textsf{wft:fst}$$

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \times^{\ell} \sigma_2}{\Delta \mid \Gamma \vdash \mathbf{snd}\, e : \sigma_2 \sqcup \ell}\ \textsf{wft:snd}$$

$$\frac{\Delta \mid \Gamma, x{:}\sigma \vdash e : \sigma \qquad \Delta \vdash \sigma}{\Delta \mid \Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma}\ \textsf{wft:fix}$$

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

$$\frac{\Delta \mid \Gamma \vdash e : \sigma_1 \qquad \Delta \vdash \sigma_1 \leq \sigma_2}{\Delta \mid \Gamma \vdash e : \sigma_2}\ \textsf{wft:sub}$$

A.2 Dynamic semantics

A.2.1 Constructor reduction

$$\frac{\tau_1 \leadsto \tau_1'}{\tau_1\tau_2 \leadsto \tau_1'\tau_2}\ \textsf{whr:app-con}$$

$$\frac{}{(\lambda\alpha{:}\kappa.\tau_1)\tau_2 \leadsto \tau_1[\tau_2/\alpha]}\ \textsf{whr:app}$$

$$\frac{\tau \leadsto \tau'}{\mathsf{Typerec}\ \tau\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times} \leadsto \mathsf{Typerec}\ \tau'\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times}}\ \textsf{whr:trec-con}$$

$$\frac{}{\mathsf{Typerec}\ (\mathsf{int})\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times} \leadsto \tau_{\mathsf{int}}}\ \textsf{whr:trec-int}$$

$$\frac{}{\begin{array}{l}\mathsf{Typerec}\ (\tau_1 \to \tau_2)\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times} \leadsto \\ \quad \tau_{\to}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times}) \\ \qquad\qquad (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times})\end{array}}\ \textsf{whr:trec-arr}$$

$$\frac{}{\begin{array}{l}\mathsf{Typerec}\ (\tau_1 \times \tau_2)\tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times} \leadsto \\ \quad \tau_{\times}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times}) \\ \qquad\qquad (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{int}}\ \tau_{\to}\ \tau_{\times})\end{array}}\ \textsf{whr:trec-prod}$$

A.2.2 Term computation rules

$$\frac{}{(\lambda x{:}\sigma.e)v \leadsto e[v/x]}\ \textsf{ev:app}$$

$$\frac{}{(\Lambda\alpha{:}\kappa.e)[\tau] \leadsto e[\tau/\alpha]}\ \textsf{ev:tapp}$$

$$\frac{}{\mathbf{fst}\ \langle v_1, v_2\rangle \leadsto v_1}\ \textsf{ev:fst} \qquad \frac{}{\mathbf{snd}\ \langle v_1, v_2\rangle \leadsto v_2}\ \textsf{ev:snd}$$

$$\frac{}{\mathbf{fix}\ x{:}\sigma.e \leadsto e[\mathbf{fix}\ x{:}\sigma.e/x]}\ \textsf{ev:fix}$$

$$\frac{\tau \leadsto^* \mathsf{int}}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\to}\ e_{\times} \leadsto e_{\mathsf{int}}}\ \textsf{ev:tcase-int}$$

$$\frac{\tau \leadsto^* \tau_1 \to \tau_2}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\to}\ e_{\times} \leadsto e_{\to}[\tau_1][\tau_2]}\ \textsf{ev:tcase-arr}$$

$$\frac{\tau \leadsto^* \tau_1 \times \tau_2}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{int}}\ e_{\to}\ e_{\times} \leadsto e_{\times}[\tau_1][\tau_2]}\ \textsf{ev:tcase-prod}$$

A.2.3 Term congruence rules

$$\frac{e_1 \leadsto e_1'}{e_1 e_2 \leadsto e_1' e_2}\ \textsf{ev:app1} \qquad \frac{e_2 \leadsto e_2'}{v_1 e_2 \leadsto v_1 e_2'}\ \textsf{ev:app2}$$

$$\frac{e_1 \leadsto e_1'}{\langle e_1, e_2\rangle \leadsto \langle e_1', e_2\rangle}\ \textsf{ev:pair1}$$

$$\frac{e_2 \leadsto e_2'}{\langle v_1, e_2\rangle \leadsto \langle v_1, e_2'\rangle}\ \textsf{ev:pair2}$$

$$\frac{e \leadsto e'}{\mathbf{fst}\, e \leadsto \mathbf{fst}\, e'}\ \textsf{ev:fst-con}$$

$$\frac{e \leadsto e'}{\mathbf{snd}\, e \leadsto \mathbf{snd}\, e'}\ \textsf{ev:snd-con}$$

$$\frac{e \leadsto e'}{e[\tau] \leadsto e'[\tau]}\ \textsf{ev:tapp-con}$$