From Type Aliases to Abstract Types: A Gradual Approach

Raimil Cruz *[1] and Éric Tanter[1]

[1] *Computer Science Department (DCC), University of Chile, Chile*

Abstract

Type aliases allow the creation of synonyms between types, which is useful in creating an implicit documentation in programs, but they do not enforce abstraction since the alias and the base type are treated without any distinction in the whole system. Abstract types allow the building of applications in a modular way providing only the type interface to create an abstraction barrier. We study the multi-agent calculus presented by Grossman et al. [**?**], which proves type abstraction properties using syntactic techniques. The calculus presented by Grossman et al. relies on the concept of principals. Examples of principals are modules of a system, such as a host or a client. In their calculus it is possible to express the knowledge that each principal has about abstract types.

In this paper we present a gradual approach to relax the static principals of the multi-agent calculus to support interaction with a calculus that has type aliases. Before developing the gradual approach, we reformulate the multi-agent calculus such that it better separates the notion of principals from program expressions. While Grossman et al. introduce separate syntactic categories for expressions (one per principal), our calculus relies on a unique syntactic category for expressions, and principals are part of the typing and evaluation contexts. This separation is closer to an actual implementation and simplifies the gradual checking approach.

1 Introduction

In the construction of software systems, developers often separate their applications in modules. Modules can serve as a mechanism to publish abstract types. Abstract types are a way of providing data abstraction [**?**]. The abstract type interface is exposed, but its implementation details are hidden. Using this abstraction, clients of a module base their implementation on the abstract type interfaces that the module provides.

*Funded by grant CONICYT, CONICYT-PCHA/Doctorado Nacional/2014-63140148

In a simple scenario, outside the module, the implementation details of abstract types are unknown. However, in a richer scenario we would define that more than one module knows the implementation details of an abstract data type. In both cases the desired property is that type abstraction is not violated. Grossman et al. [**?**] present a calculus where it is possible to express the knowledge that principals have about abstract types in the system, via a global type mapping. In their work the concept of modules is associated to the notion of principals used in security.

In a language with no explicit constructions for creating modules and abstract types, type aliases can be used to express the intention of the programmer of separating a concept from its implementation. We are interested in studying the interaction of a calculus with type aliases (without a notion of type abstraction) and a calculus with type abstraction. We relax the notion of static principals of Grossman et al. to provide a flexible way to convert type aliases to abstract types while ensuring type abstraction properties. We appeal to the ideas of gradual typing [**?**, **?**] and its applications to other typing disciplines [**?**, **?**, **?**, **?**, **?**] to get a gradual calculus for type abstraction.

Our contributions in this paper are the following:

- We reformulate of the multi-agent calculus of Grossman et al. [**?**] without having multiple copies of the same language for different principals, removing principal annotations from the language terms. We recover principal information in the typing and evaluation contexts. This calculus is presented in Section **??**.
- We provide an implementation of the calculus in Redex [**?**] (`https://bitbucket.org/raimilcruz/sta`).
- We present a gradual principal calculus to enforce type abstraction in the presence of type aliases. We follow the framework of Abstracting Gradual Typing (AGT) [**?**] to develop the gradual type system in Section **??**. We use the ideas of Siek and Taha [**?**] to develop the dynamic semantics of our gradual principal calculus through a translation to an intermediate language with cast (Section **??**)

2 Background

In this section we explain the concepts of type aliases and abstract typees and why they are useful. We introduce the approach of Grossman et al. to ensure type abstraction over a variation of the simply typed lambda calculus with a notion of principals [**?**]. We only present the most important concepts that are necessary to understand the reformulation we present in Section **??**.

2.1 Type alias

A type alias is a type synonyms for an existing type in the system. For example, it is possible to create a type alias in Haskell using the `newtype` expression.

```
newtype Id = String
```

A type alias expresses the programmer intention of making a distinction between the alias type (here, `Id`) and the base type (here, `String`) in the system. The distinction is useful for documentation purposes. It allows the use of the type synonym in any context where the original type is expected and the other way around. For example, in the code below, the `concactHello` function receives an argument of type `Id`, but we can apply this function indistinctly with `Id` or `String` because they are technically the same type.

```
concatHello ::  Id → Id
concatHello s = "hello " ++ s
```

The `concatHello` function uses Haskell's concatantetion function (`++`) over strings. The `++` function has type `String → String → String`, so the argument `s` is treated as what it really is: a `String`.

2.2 Abstract types

An abstract type is a type whose exact identity (i.e realization) is unknown to its clients. For example, suppose we have a file handler interface like the one in the Figure **??**. The file handle abstract type `fh` provides an interface with two functions: `open` and `read`. The only way for a client to get a file handler instance is to use the `open` method. However, the one that provides the implementation, let us say a host principal, knows what the realization of `fh` is. For the rest of this section we assume that host implements `fh` with an `int`.

```
abstype  fh
open : string → fh
read : fh → char
```

Figure 1: Abstract interface for file handlers

A type system that supports abstract types must ensure *type abstraction*. For the file handler example, this means to prevent a client from using the fact that `fh` is implemented as `int`. The common way of achieving type abstraction with abstract types is to use *existential types* [**?**]. However, we are interested in exploring the syntactic approach of Grossman et al. [**?**], which appeals to the notion of principals (e.g., client and host) to ensure type abstraction properties.

2.3 Syntactic Type Abstraction

Grossman et al. present an approach to ensure type abstraction properties in a calculus with the notion of principals [**?**]. The calculus establishes a syntactic distinction between code of different principals annotating each term with the principal to which it belongs. For example the `open` function in the host is encoded as follows: $\lambda s_h : \texttt{string} \rightarrow \texttt{int}.\mathit{hopen}\ s_h$. The subscript h indicates that the function belongs to the host and the `hopen` (for host open) function is the internal system implementation for opening a file given a path.

The abstraction barrier is created in the calculus with a syntactic embedding for the abstracted term. For example the host could export the `open` function as follows: $[\lambda s_h : \texttt{string} \rightarrow \texttt{int}.\mathit{hopen}\ s_h]_h^{\texttt{string}\rightarrow\texttt{fh}}$. The embedding superscript indicates the abstract type `fh` and the embedding subscript the principal that creates the abstraction, in this case the `host`. The embedding is propagated during the function application to keep the abstraction barrier. For example the following function application $[\lambda s_h : \texttt{string} \rightarrow \texttt{int}.\mathit{hopen}\ s_h]_h^{\texttt{string}\rightarrow\texttt{fh}}$ "file"$_c$ in the client code evaluates to an embedding containing the internal `int` file handle, $[3]_h^{\texttt{fh}}$.

Attempts to violate the abstraction are detected by the type system. For example the client expression $1_c + [3]_h^{\texttt{fh}}$ is not well-typed, because the client does not have permission to see what the internal term of the embedding is. However in the host code, the expression $1_h + [3]_h^{\texttt{fh}}$ reduces to $1_h + 3_h$ (because the host knows `fh = int`) and finally reduces to `4`.

These examples show that the result of an expression depends on which principal is executing the code. In those simple examples there is only an abstract type (`fh`), where the host knows `fh=int` and for the client `fh` is abstract. In a scenario with multiple principals and abstract types, the knowledge of each principal p about each abstract type is defined using a partial mapping function δ_p. For the file handle example we have only the mapping $\delta_h(\texttt{fh}) = \texttt{int}$ because the client does not know the file handle implementation ($\texttt{fh} \notin \mathit{dom}(\delta_c)$).

Figure **??** resumes the interaction between a client function and a host function. The code is executed in the client context. A client obtains a file handle using the `open` function of the host. Step 1 shows how the host function is adapted to be used in client code, uncovering the function term but keeping the embedding inside its body to protect the file handle abstraction. After some evaluation steps the client obtains an embedding and it is not able to see what is inside.

The calculus of Grossman et al. is presented as multiple copies of a simply typed lambda calculus. Each principal has an instance of the calculus. This approach is convenient to prove type abstraction properties syntactically, but it is not handy to implement in a real language.

$$\delta_h(\texttt{fh}) = \texttt{int}$$

$(\lambda open_c : \texttt{string} \to \texttt{fh}.\ (open_c\ \text{“}file\text{”}_c))\,[\lambda s_h : \texttt{string}.\ hopen\ s_h]_h^{\texttt{string}\to\texttt{fh}}$

1 $(\lambda open_c : \texttt{string} \to \texttt{fh}.\ (open_c\ \text{“}file\text{”}_c))(\lambda s_c : \texttt{string}.\ [hopen\,[s_c]_c^{\texttt{string}}]_h^{\texttt{fh}})$

2 $(\lambda s_c : \texttt{string}.\ [hopen\,[s_c]_c^{\texttt{string}}]_h^{\texttt{fh}})\text{“}file\text{”}_c$

.

4 $[3_h]_h^{\texttt{fh}}$

Figure 2: Client and server interaction

3 Unified Principal Calculus

In the this section we modify the syntax of the multi-agent calculus of Grossman et al. to provide a unique syntactic category for expressions. We remove principal annotations for terms, except for embeddings and a new let term. Instead, we include principals as part of the typing and evaluation contexts. These modifications make it easier both to implement the language and formulate a gradual type abstraction checking approach.

3.1 Model

The syntax of the new calculus is mostly the same as that of the multi-agent calculus but removing principal annotations from terms. We call it the unified principal calculus (UPC) and we denote it $\lambda^p_{[\cdot]}$. This is a simply typed lambda calculus augmented with term embedding and a new form of `let` expression to introduce principals. Terms include variables x, constants b, functions $\lambda x : \tau.\, e$, function application $e\ e$ and embeddings. Types are base types `B` (e.g `int`, `string`), abstract types `t` and function types. The new expression $let_p\ x : \tau = e_1\ in\ e_2$ allows principal p to export term e_1 at type τ through the x variable in the expression e_2. In other words, it is just a syntactic sugar for $(\lambda x : \tau.e_2)\,[e_1]_p^\tau$. Embeddings have the same role as in the multi-agent calculus. Each embedding is annotated with the list of agents that participate in the creation of the abstraction barrier. In the file handle example in Section **??** for simplicity we avoided using lists in embedding. We explain what the role of list of principals is in Section **??**.

There are two kinds of values: primitive values $\hat{v}$ and extended values. Primitive values are ground values (e.g. `1`, “foo”) and functions. Extended values include primitive values and embeddings. An embedding $[\hat{v}]_l^\tau$ of a primitive value is also a value if the type τ is abstract for the principal in context. Principal contexts appear in static and dynamic semantics, so we cannot talk about embedding values at the syntax level.

Principals have different level of knowledge of each type, for example in the file handle example the `host` knows $\texttt{fh} = \texttt{int}$. This knowledge is expressed with the partial mapping $\delta_{host}(\texttt{fh}) = \texttt{int}$. Grossman et al. [**?**] define the notion of compatible mappings. We write this definition below because we appeal to it in

$$p \in \textsc{Principal} \quad \hat{v} \in \textsc{Value} \quad e \in \textsc{Term} \quad v \in \textsc{XValue}$$
$$\tau \in \textsc{Type} \quad b \in \textsc{GroundType} \quad l \in \textsc{PrincipalList} \quad \mathtt{t} \in \textsc{AbstractType}$$

$$\begin{array}{llll}
\text{(terms)} & e & ::= & x \mid b \mid \lambda x : \tau.\, e \mid e\, e \mid let_p\, x : \tau = e\, in\, e \mid [\![e]\!]^{\tau}_{l} \\
\text{(values)} & \hat{v} & ::= & b \mid \lambda x : \tau.\, e \\
\text{(xvalues)} & v & ::= & \hat{v} \mid [\![\hat{v}]\!]^{\mathtt{t}}_{l} \\
\text{(types)} & \tau & ::= & \mathtt{B} \mid \tau \to \tau \mid \mathtt{t} \\
 & & & \\
\text{(principal lists)} & l & ::= & p \mid pl
\end{array}$$

Figure 3: Unified model for multi-agent calculus

the Section **??**.

Definition 3.1. *(Well-formed type mapping) A set $\{\delta_1, ..., \delta_n\}$ of finite partial maps from type variables to types is compatible if:*

1. *$\forall i, j \in \{1, ..., n\}$ if $t \in dom(\delta_i) \cap dom(\delta_j)$, then $\delta_i(t) = \delta_j(t)$.*
2. *The collection of type variables must be totally ordered such that for every agent i and type variable t, all variables in $\delta_i(t)$ precede t.*

Note that second condition denotes that type variables and types form a directed acyclic graph, which means that is not possible to have loops in a mappings. The total version of δ_p is Δ_p and it defined as follows:

$$\begin{array}{rcl}
\Delta_p(\mathtt{B}) & = & \mathtt{B} \\
\Delta_p(\mathtt{t}) & = & \begin{cases} \mathtt{t} & \mathtt{t} \notin dom(\delta_p) \\ \delta_p(\mathtt{t}) & \mathtt{t} \in dom(\delta_p) \end{cases} \\
\Delta_p(\tau_1 \to \tau_2) & = & \Delta_p(\tau_1) \to \Delta_p(\tau_2)
\end{array}$$

The most concrete view that a principal p has for a type τ is obtained with the function $\bar{\Delta}_p$. This function is the result of applying Δ_p iteratively until having $\Delta_p(\tau) = \tau$. For example, given the mapping $\delta_h(\mathtt{fh}_2) = \mathtt{fh}_1, \delta_h(\mathtt{fh}_1) = \mathtt{fh}$, the most concrete view that h has for fh_2 is $\mathtt{fh}$, i.e. $\bar{\Delta}_h(fh_2) = \mathtt{fh}$.

3.2 Static semantics

In this section we define the static semantics for the unified calculus (Figure **??**). The type judgment $\Gamma, p \vdash e : T$ makes explicit which principal is in context. It can be read as the expression e has type τ for the principal p in lexical context Γ. Some typing rules use $\bar{\Delta}_p$ which is defined over the δ_p mapping as explained before. This mapping is specified globally and does not change during the type checking process. The type judgment in Figure **??** shows that all expressions are typed in a principal context. We should specify the principal for the top

level program. We call that principal world. The world principal has its own δ mappings like any other principal.

Before describing the typing rules, we present an example to show how type-checking is done. In the code, the world principal uses the `open` function of the `host` principal.

$$\delta_h(\texttt{fh}) = \texttt{int}, \texttt{fh} \notin dom(\delta_{\mathsf{world}})$$

$$\begin{array}{lll} let_h\ open\ :\ \texttt{string} \to \texttt{fh} & = \lambda s : \texttt{string}.\ hopen\ s \\ & in \\ & (open\ \text{“file”}) + 1 \end{array}$$

This program is not well-typed because of the expression `(open "file") + 1`. The expression `(open "file")` has type `fh`, but the world principal does not know that `fh` is an integer ($\texttt{fh} \notin dom(\delta_{\mathsf{world}})$). However if we add the mapping $\delta_{\mathsf{world}}(\texttt{fh}) = \texttt{int}$ the code type checks.

The Figure **??** show that we add explicit constraints for each typing rule without using the same meta-variable. This follows the methodology proposed for the Abstracting Gradual Typing (AGT) [**?**] to systematically derive the gradual version of the type system (Section **??**). The most important constraint is type equality. For $\lambda^p_{[.]}$, type equality involves the knowledge of the principal about both types. We define type equality as follows:

Definition 3.1. *(Type equality) Two types τ_1 and τ_2 are equal for the principal p, notation $\tau_1 =_p \tau_2$, iff $\bar{\Delta}_p(\tau_1) = \bar{\Delta}_p(\tau_2)$*

$\boxed{\Gamma, p \vdash e : T}$

$$\frac{}{\Gamma, p \vdash x : \bar{\Delta}_p(\Gamma(x))}\ [\text{T-Var}] \qquad \frac{}{\Gamma, p \vdash b : \texttt{B}}\ [\text{T-Cons}]$$

$$\frac{\Gamma, p \vdash e_1 : \tau_{11} \to \tau \quad \Gamma, p \vdash e_2 : \tau_{12} \quad \tau_{11} =_p \tau_{12}}{\Gamma, p \vdash e_1\ e_2 : \tau}\ [\text{T-App}]$$

$$\frac{\Gamma[x : \tau_1], p \vdash e : \tau_2}{\Gamma, p \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ [\text{T-Abs}]$$

$$\frac{\Gamma, p_1 \vdash e_1 : \tau_1 \quad \Gamma[x : \tau], p \vdash e_2 : \tau_2 \quad \tau =_p \tau_1}{\Gamma, p \vdash let_{p_1}\ x : \tau =\ e_1\ in\ e_2 : \tau_2}\ [\text{T-Let}]$$

$$\frac{\Gamma, p' \vdash e : \tau_1 \quad \tau_1 \hookleftarrow_{p'\overline{p}} \tau}{\Gamma, p \vdash [e]^{\tau}_{p'\overline{p}} : \bar{\Delta}_p(\tau)}\ [\text{T-Emb}]$$

Figure 4: Static semantics for unified calculus

All rules except [T-Let] and [T-Emb] are standard. Rule [T-Var] gives the most concrete type of a variable from the environment for the principal in context. Rule [T-App] explicitly indicates that the formal function argument and the actual argument must have the same type for the principal p. Rule

$\boxed{\tau \hookleftarrow_l \tau'}$

$$\frac{\bar{\Delta}_p(\tau) = \bar{\Delta}_p(\tau')}{\tau \hookleftarrow_p \tau'}\ \text{[T-eq]} \qquad \frac{\tau \hookleftarrow_l \bar{\Delta}_p(\tau')}{\tau \hookleftarrow_{lp} \tau'}\ \text{[T-trans]}$$

Figure 5: Collaborative principals knowledge relation $\tau \hookleftarrow_l \tau'$

[T-Abs] in the original multi-agent calculus had a constraint in function parameters $\bar{\Delta}_p(\tau) = \tau$ that forces an agent to use the most concrete view for function parameters. We remove this restriction since we get the most concrete view for variables in rule [T-Var]. This change is relevant to encode type aliases in the gradual version of $\lambda^p_{[.]}$ and support functions like `concat` with type $\texttt{Id} \to \texttt{Id} \to \texttt{Id}$.

The new rule [T-Let] expresses the decision of abstracting a type. There is a requirement that the principal (that exports the type) has the knowledge that the type of the named expression is equal to the annotated type of the identifier ($\tau =_p \tau_1$). In this rule a change in the principal context takes place for type-checking expression e_1. This reflects that the expression e_1 belongs to principal p_1. The same happens in the [T-Emb] rule for type-checking the expression inside the embedding. The [T-Emb] rule uses the typing relation $\tau \hookleftarrow_l \tau'$ to check that the list of annotated principals can prove collaboratively that the annotated type can be reduced to the type of expression inside the embedding. We make a little change to the typing relation $\tau \hookleftarrow_l \tau'$ of the multi-agent calculus so that is deterministic. We show how embedding with multiple principals appears in the dynamic semantic (in Section **??**).

3.3 Dynamic semantics

In this section we present the dynamic semantics of the $\lambda^p_{[.]}$. Before that we define a predicate $value_p$ to characterize whether a term is a value or not for a principal p. This predicate is used in the evaluation rules

Definition 3.2. *(Value) A term e is a value for a principal p if the predicate* $value_p(e)$ *holds:*

$$\frac{}{value_p(\hat{v})} \qquad \frac{t \notin dom(\delta_p)}{value_p(\llbracket \hat{v} \rrbracket^t_l)}$$

We also define the predicate $reducible_p$ to know when a principal can reduce an abstract type and we define the predicate $abtract_p$ to know when a principal p does not know nothing about a type.

Definition 3.3. *(Reducible) A type τ is reducible for a principal p if the predicate* $reducible_p(\tau)$ *holds:*

$$reducible_p(\tau) \text{ iff } \Delta_p(\tau) \neq \tau$$

Definition 3.4. *(Abstract) A type τ is fully abstract for a principal p if the predicate $fully\text{-}abstract_p$ holds:*

$$abstract_p(\tau) \textit{ iff } \tau \notin dom(\delta_p)$$

Figure **??** shows the small-step operational semantics of $\lambda^p_{[.]}$. In essence, the dynamic semantics are the same as that of the multi-agent calculus. The main difference is that we explicitly know which principal is executing each an expression. A program configuration is a pair of a principal and an expression. Reduction rules take a program configuration to the next. Rules [EApp1] and [EApp2] represent reductions for function application. First we reduce the function expression and then the argument expression. Once both expressions in an application are values for the current principal, the [EApp] rules applies. This is a typical application reduction rule. Expression $[x \to v]\, e$ substitutes v for x in e.

$$\frac{\langle p, e_1\rangle \longmapsto \langle p, e_1'\rangle}{\langle p, e_1 e_2\rangle \longmapsto \langle p, e_1' e_2\rangle}\ [\text{EApp1}] \qquad \frac{\langle p, e_2\rangle \longmapsto \langle p, e_2'\rangle \quad value_p(v_1)}{\langle p, v_1 e_2\rangle \longmapsto \langle p, v_1 e_2'\rangle}\ [\text{EApp2}]$$

$$\frac{value_p(v)}{\langle p, (\lambda x : \tau.\, e) v\rangle \longmapsto \langle p, [x \to v]\, e\rangle}\ [\text{EApp}]$$

$$\frac{\langle p_1, e\rangle \longmapsto \langle p_1, e''\rangle}{\langle p, let_{p_1}\ x : \tau = \ e\ in\ e'\rangle \longmapsto \langle p, let_{p_1}\ x : \tau = \ e''\ in\ e'\rangle}\ [\text{ELet1}]$$

$$\frac{value_{p_1}(v)}{\langle p, let_{p_1}\ x : \tau = \ v\ in\ e'\rangle \longmapsto \left\langle p, (\lambda x : \tau.\, e')\, \llbracket v \rrbracket^{\tau}_{p_1}\right\rangle}\ [\text{ELet2}]$$

$$\frac{\langle p_1, e\rangle \longmapsto \langle p_1, e'\rangle}{\left\langle p, \llbracket e \rrbracket^{\tau}_{p_1 \overline{p}}\right\rangle \longmapsto \left\langle p, \llbracket e' \rrbracket^{\tau}_{p_1 \overline{p}}\right\rangle}\ [\text{EEmbr1}] \qquad \frac{reducible_p(\tau) \quad value_k(v)}{\left\langle p, \llbracket v \rrbracket^{\tau}_{k \overline{p}}\right\rangle \longmapsto \left\langle p, \llbracket v \rrbracket^{\bar{\Delta}_p(\tau)}_{k \overline{p}}\right\rangle}\ [\text{EEmbr2}]$$

$$\frac{}{\left\langle p, \llbracket b \rrbracket^{\mathtt{B}}_{k \overline{p}}\right\rangle \longmapsto \langle p, v\rangle}\ [\text{EEmbO}] \qquad \frac{!reducible_p(\tau) \quad full\text{-}abstract_k(\tau_1)}{\left\langle p, \llbracket \llbracket \hat{v} \rrbracket^{\tau_1}_{l_1} \rrbracket^{\tau}_{kl}\right\rangle \longmapsto \left\langle p, \llbracket \hat{v} \rrbracket^{\tau}_{l_1 kl}\right\rangle}\ [\text{EEmbC}]$$

$$\frac{!reducible_p(\tau_1 \to \tau_2) \quad x' fresh}{\left\langle p, \llbracket \lambda x : \tau.\, e \rrbracket^{\tau_1 \to \tau_2}_{l_1}\right\rangle \longmapsto \left\langle p, \lambda x' : \tau_1.\, \llbracket \left\{ \llbracket x' \rrbracket^{\tau}_{\mathtt{prev}(l_1)} / x \right\} e \rrbracket^{\tau_2}_{l_1}\right\rangle}\ [\text{EEmbF}]$$

Figure 6: Dynamic semantics for the unified calculus

The reduction rules for `let` are interesting because they change the principal context. The [ELetr1] rule reduces the expression e_1 but using the declared

principal p_1 as the principal in context. [ELet] rule evaluates the `let` expression as a lambda application, embedding the argument.

Once we have an embedding, the current principal obtains the most concrete view that it knows for the annotated type through [EEmbr2]. With the embedding annotated type reduced, one of the following rules would apply: [EEmbO], [EEmbC] or [EEmbF]. [EEmbO] applies when the embedding annotated type is a base type and in this case it is safe to open the embedding. [EEmbC] applies when the embedding annotated type remains abstract for the principal and the expression inside the embedding is another embedding. In this case embeddings are compacted and the list of principals of the external embedding is appended to the list of principals of the internal embedding. This is the place where an embedding list grows.

The [EEmbF] rule exports a function from one principal to another. A new function is created for the principal in the evaluation context expecting an argument of type τ_1, which is the one that the principal `p` expects. The body of the function belongs to the first principal in the list of principals l_1 and for this reason the embedding for the body is created. The x argument of the new function is also embedded in $[x]^{\tau}_{p\texttt{rev(l1)}}$ to protect the possible abstract types of the principal p in the other principal contexts. The helper function rev inverts the agent list, because agents in list $\texttt{prev}(l_1)$ should be able to prove that τ is related to the type of x, τ_1.

Figure **??** illustrates the runtime semantics with an example that consists of the interaction between three principals: p_1, p_2 and world. The example describes reduction step by step. Principal p_1 knows $\tau_1 = \tau_2$, p_2 knows $\tau_2 = \texttt{int}$ and principal `world` does not have any knowledge about τ_1 or τ_2. We have a function `f` that uses a built-in function `toStr` of type $\texttt{int} \to \texttt{str}$ to get the literal representation of an integer. Function `f` has type $\texttt{int} \to \texttt{str}$. On the left we have annotated the name of the reduction rules used for each evaluation step.

$$\delta_{p1}(\tau_1) = \tau_2, \delta_{p2}(\tau_2) = \texttt{int}$$
$$f \doteq \lambda x : \texttt{int}.\ toStr\ x$$

$$\begin{array}{rl}
 & \langle w, let_{p_1}\ f_1 : \tau_1 \to \texttt{str} = \ (let_{p_2}\ f_2 : \tau_2 \to \texttt{str} = \ f\ in\ f_2)\ in\ f_1 \rangle \\
\text{[ELet]} & \langle w, let_{p_1}\ f_1 : \tau_1 \to \texttt{str} = \ ((\lambda f_2 : \tau_2 \to \texttt{str}.\ f_2)[f]^{\tau_2 \to \texttt{str}}_{p_2})\ in\ f_1 \rangle \\
\text{[EApp]} & \langle w, let_{p_1}\ f_1 : \tau_1 \to \texttt{str} = \ [f]^{\tau_2 \to \texttt{str}}_{p_2}\ in\ f_1 \rangle \\
\text{[ELet]} & \langle w, (\lambda f_1 : \tau_1 \to \texttt{str}.\ f_1)[[f]^{\tau_2 \to \texttt{str}}_{p_2}]^{\tau_1 \to \texttt{str}}_{p_1} \rangle \\
\text{[EEmbC]} & \langle w, (\lambda f_1 : \tau_1 \to \texttt{str}.\ f_1)[f]^{\tau_1 \to \texttt{str}}_{p_2 p_1} \rangle \\
\text{[EApp]} & \langle w, [f]^{\tau_1 \to \texttt{str}}_{p_2 p_1} \rangle
\end{array}$$

Figure 7: Reduction rules in action. Example 1

The initial expression has two `let` expressions. Principal p_2 uses the inner `let` expression to export the function `f` with abstract type $\tau_2 \to \texttt{str}$. Principal p_1 uses the outer `let` expression to abstract the resulting inner function from

type $\tau_2 \to \texttt{str}$ to type $\tau_1 \to \texttt{str}$. The first evaluation uses the [TLet] rule for converting the inner `let` expression to a function application, with the embedding $[\![f]\!]_{p_2}^{\tau_2 \to \texttt{str}}$. The second evaluation step uses the [EApp] to reduce the function application. Third step uses the [TLet] rule for reducing the outer `let` expression. This produces nesting of embeddings, so the [EEmbC] is used for combining the two embeddings in a single embedding $[\![f]\!]_{p_2 p_1}^{\tau_1 \to \texttt{str}}$. Recall that the process of combining embeddings combines lists of principals for remembering which principals participate in creating the abstraction barrier. Finally, the last evaluation step uses the [EApp] rule to apply the function and we get an embedding as result. The principal world is not able to reduce the embedding ($[\![f]\!]_{p_2 p_1}^{\tau_1 \to \texttt{str}}$), because it does not have knowledge about the type τ_1 (so the embedding is a value for world)

4 Gradual Language

In this section we present a gradual calculus[1] $\lambda_{[.]}^{?}$ that includes the notion of an unknown principal. We give an intuition of what is the meaning of the unknown principal before providing the formalization of the gradual calculus. This calculus is obtained from $\lambda_{[.]}^{p}$ using the Abstracting Gradual Typing (AGT) approach of Garcia et al. [**?**].

4.1 Dynamic principals

Suppose we have some legacy code that uses type aliases but does not have a notion of type abstraction. We may want to strengthen some type aliases to be abstract types. This implies that the programmers must be explicit about what parts of the code are on the implementation side and what parts are on the client side. Following the syntactic type abstraction approach we have to specify a static principal for all parts of the code that use a type alias.

In order to be flexible with this principal assignments, we present a system that allows to associate principals in a gradual way, detecting inconsistencies statically where possible or dynamically when there is no principal declarations. In particular our approach allows code without an associated principal to be used "on behalf of" multiples principals.

Following the gradual typing approach we introduce the notion of an *unknown principal* `?` to deal with code that does not have a principal associated. The following example shows the unknown principal in action. We have a host function `g` defined in a scope accessible by the `let` expression. Function `g` has type `fh` $\to$ `str`. The `let` expression introduces the identifier `f` to denote a function with unknown principal. The body of the function applies the function `g` to a `string`. This invocation takes place in a code with an unknown principal associated. This could violate the type abstraction in $\lambda_{[.]}^{p}$ because `g` receives a `fh`. However, we want to be flexible when some code belongs to the

[1] see associated survey for a review of gradual typing and related approaches

unknown principal. The intuition is that if there exists a principal that knows the abstract type implementation, then code with the unknown principal can statically use this knowledge. This gives the possibility that if the function `f` is applied by a principal with the required knowledge, then there is no type abstraction violation. For this concrete example, the host principal knows that `fh = int`, so the type checker accepts the program and the execution of `f 1` goes well because the function `f` is executed by the host.

$$\begin{array}{l}\delta_h(\texttt{fh}) = \texttt{int} \\ g \doteq [\lambda x : \texttt{int}.\, toStr\ x]_h^{\texttt{fh}\to\texttt{str}} \\ let_? \ f : \texttt{int} \to \texttt{str} = \ \lambda x : \texttt{int}.g\ x\ in\ f\ 1\end{array}$$

Figure 8: Function of the unknown principal executed by the host

The next example is similar to the previous one, but the function `f` is now executed by the client. The program is accepted by the type checker (because of $\delta_h(\texttt{fh}) = \texttt{int}$), but the evaluation fails at runtime because the client does not know the implementation of `fh`.

$$let_? \ f : \texttt{int} \to \texttt{str} = \ \lambda x : \texttt{int}.g\ x\ in\ f\ 1$$

Figure 9: Function of the unknown principal executed by the client

In the following program the unknown principal is exporting a function with type concrete `int` $\to$ `int` with the abstract type `int` $\to$ `fh`. This time, for all principal p we have $dom(\delta_p) = \emptyset$, so no principal has the knowledge that `fh = int`. In this case the type checker statically rejects the program.

$$let_? \ intToFh : \texttt{int} \to \texttt{fh} = \ \lambda x : \texttt{int}.x\ in\ intToFh\ 1$$

Figure 10: Unknown principal exports function with abstract type

4.2 Syntax

To get the static semantics of the gradual calculus, we follow the AGT methodology [**?**]. First, we define gradual principals to include the *unknown principal* `?`.

$$\begin{array}{lcl}\widetilde{p} & \in & \text{GPrincipal} \\ \widetilde{p} & ::= & p \mid ?\end{array}$$

We also define gradual terms that are terms with gradual principals and gradual list of embeddings to include gradual principal in the list of principals of an embeddings.

$$\begin{array}{lcl}\widetilde{e} & \in & \text{GTerm} \\ \widetilde{e} & ::= & \ldots \mid let_{\widetilde{p}}\, x : \tau = \widetilde{e}\, in\, \widetilde{e} \mid \ [\widetilde{e}]_{\widetilde{l}}^{\tau} \\ \widetilde{l} & \in & \text{GPrincipalList} \\ \widetilde{l} & ::= & \widetilde{p} \mid \widetilde{l}\widetilde{p}\end{array}$$

A key aspect of using the AGT methodology is to define the meaning of the unknown type information through a concretization function γ. Our concretization function goes from gradual principals to sets of static principals (Definition **??**). The concretization function γ_p expresses that the unknown principal represents any principal. In particular in our gradual calculus the unknown principal represents the first known principal in a parent scope. This is explained in the Section **??**.

Definition 4.1. *(Principal Concretization). Let* $\gamma_p : \text{GPrincipal} \to \mathcal{P}(\text{Principal})$ *be defined as follow:*

$$\gamma_p(p) = \{p\}$$
$$\gamma_p(?) = \text{Principal}$$

4.3 Lifting predicates and functions

Once we have the concretization function, we lift predicates and functions on static principals to their consistent predicates and gradual functions on gradual principals. We have a single predicate in our type system, which is the type equality for a principal ($\tau_1 =_p \tau_2$). We lift the type equality predicate to get the consistent type equality on gradual principals.

Definition 4.2. *(Consistent type equality).* $\tau_1 \mathrel{\widetilde{=}}_{\widetilde{p}} \tau_2$ *iff* $\tau_1 =_p \tau_2$ *for some* $p \in \gamma_p(\widetilde{p})$

The consistent type equality relation informally says that two types are equal for the unknown principal if there exists a principal in the system for which the type equality relation holds.

Once we have lifted predicates, we lift functions on set of static principals to gradual principals. In $\lambda^{p}_{[.]}$ we have two functions on static principals: Δ_p and $\bar{\Delta}_p$. Both functions take a static principal and a static type and return a static type ($\text{Principal} \times \text{Type} \to \text{Type}$). In our gradual calculus we are only interested in supporting gradual principals, but not gradual types. This means that the lifted versions of Δ_p and $\bar{\Delta}_p$, which are $\widetilde{\Delta}_p$ and $\widetilde{\bar{\Delta}}_p$ should take a gradual principal and a static type and return a static type ($\text{GPrincipal} \times \text{Type} \to \text{Type}$). Let us see a first attempt to define $\widetilde{\Delta}_p$ using the AGT approach for lifting functions:

$$\widetilde{\Delta}(\widetilde{p}, \tau_1) = \{\tau_2 \mid \Delta_p(\tau_1) = \tau_2 \quad p \in \gamma_p(\widetilde{p})\}$$

In this definition we write $\widetilde{\Delta}(\widetilde{p}, \tau_1)$ instead $\widetilde{\Delta}_{\widetilde{p}}(\tau_1)$ to make explicit that the function takes a gradual principal as argument. The definition of $\widetilde{\Delta}(\widetilde{p}, \tau_1)$ theoretically gets an arbitrary set. We can then use the AGT approach and abstract the resulting set of static types to some kind of gradual type, but we do not want gradual types in our system. We do not need to appeal to gradual type because the resulting set in our system must have a single static type. We formalize this property in the following lemma:

Lemma 1. *(Singleton result for* $\widetilde{\Delta}$*).* $\forall \widetilde{p}, \tau \quad \widetilde{\Delta}(\widetilde{p}, \tau) = \{\tau'\}$ *for some* τ'

Proof. Follows from the δ mapping constraints that if $\tau \in Dom(\delta_i) \cap Dom(\delta_j)$ then $\delta_i(\tau) = \delta_j(\tau)$. □

Now we define the right version of $\widetilde{\Delta}$ for our gradual calculus. We use a partial helper function *single* that is defined only for a set of one element, an in this case returns the element.

Definition 4.3. *(Gradual version of Δ_p). Let* $\widetilde{\Delta}$: GPrincipal × Type → Type *be defined as follows:*
$\widetilde{\Delta}(\widetilde{p}, \tau_1) = single(\{\tau_2 \mid \Delta_p(\tau_1) = \tau_2 \quad p \in \gamma_p(\widetilde{p})\})$

The definition of $\widetilde{\bar{\Delta}}$ following AGT has to deal with a similar issue. Let us see a first attempt:

$$\widetilde{\bar{\Delta}}(\widetilde{p}, \tau_1) = \{\tau_2 \mid \bar{\Delta}_p(\tau_1) = \tau_2 \quad p \in \gamma_p(\widetilde{p})\}$$

In this case the resulting set could have many inhabitants. For example if we have $\delta_{p_1}(\texttt{fh}_2) = \texttt{fh}$, $\delta_{p_1}(\texttt{fh}) = \texttt{int}$, $\delta_{p_2}(\texttt{fh}_2) = \texttt{fh}$, then we have $\bar{\Delta}_{p_1}(\texttt{fh}_2) = \texttt{int}$ and $\bar{\Delta}_{p_2}(\texttt{fh}_2) = \texttt{fh}$. This means that $\widetilde{\bar{\Delta}}(?, \texttt{fh}_2) = \{\texttt{fh}, \texttt{int}\}$. We could provide a kind of bounded gradual type, that is not used explicitly by programmers, but are used internally in gradual functions and predicates. In this work we avoid that and provide a direct version of $\widetilde{\bar{\Delta}}$, which follows our intuition about the meaning of the unknown in the function $\widetilde{\bar{\Delta}}$. We have that the function $\bar{\Delta}_p(\tau)$ gives the most concrete view of τ for p, so the function $\widetilde{\bar{\Delta}}_?(\tau)$ gives the most concrete view of τ of all principals.

The second constraint in the definition of the δ mappings (Definition **??**), tells us that there exists a partial order for type variables and types, which is type precision. For example, if we have for some p that $\Delta_p(\texttt{fh}) = \texttt{int}$, then we can say that `int` is more precise than `fh` (written $\texttt{int} \sqsubseteq \texttt{fh}$). Beside, from the constraints of δ_p mappings, we can say that there exists a unique finite sequence of type variables from a type variable t to the most concrete view τ that any principal has. This sequence has a type precision order, and hence a minimum.

The final definition of $\widetilde{\bar{\Delta}}_p$ is the following:

Definition 4.4. *(Gradual version of $\bar{\Delta}_p$). Let* $\widetilde{\bar{\Delta}}$: GPrincipal × Type → Type *be defined as follows*
$\widetilde{\bar{\Delta}}(\widetilde{p}, \tau_1) = min(\{\tau_2 \mid \bar{\Delta}_p(\tau_1) = \tau_2 \quad p \in \gamma_p(\widetilde{p})\})$

4.4 Gradual Type System

We obtain the gradual type system by replacing static principals with gradual principals, lifting predicates on static principals to gradual principals and lifting functions on static principals to gradual principals following AGT. Figure **??** shows the resulting gradual type system.

$\boxed{\Gamma, \widetilde{p} \vdash e : T}$

$$\frac{}{\Gamma, \widetilde{p} \vdash x : \widetilde{\widetilde{\Delta}}_{\widetilde{p}}(\Gamma(x))}\ [\text{GT-Var}] \qquad \frac{}{\Gamma, \widetilde{p} \vdash b : \mathtt{B}}\ [\text{GT-Const}]$$

$$\frac{\Gamma, \widetilde{p} \vdash \widetilde{e}_1 : \tau_{11} \to \tau \quad \Gamma, \widetilde{p} \vdash \widetilde{e}_2 : \tau_{12} \quad \tau_{11} \widetilde{=}_{\widetilde{p}} \tau_{12}}{\Gamma, \widetilde{p} \vdash \widetilde{e}_1\ \widetilde{e}_2 : \tau}\ [\text{GT-App}]$$

$$\frac{\Gamma[x : \tau_1], \widetilde{p} \vdash \widetilde{e} : \tau_2}{\Gamma, \widetilde{p} \vdash \lambda x : \tau_1.\, \widetilde{e} : \tau_1 \to \tau_2}\ [\text{GT-Abs}]$$

$$\frac{\Gamma, \widetilde{p}_1 \vdash \widetilde{e}_1 : \tau_1 \quad \Gamma[x : \tau], \widetilde{p} \vdash \widetilde{e}_2 : \tau_2 \quad \tau \widetilde{=}_{\widetilde{p}} \tau_1}{\Gamma, \widetilde{p} \vdash \mathit{let}_{\widetilde{p}_1}\ x : \tau = \ \widetilde{e_1}\ \mathit{in}\ \widetilde{e_2} : \tau_2}\ [\text{GT-Let}]$$

$$\frac{\Gamma, \widetilde{p}_1 \vdash \widetilde{e} : \tau_1 \quad \vdash \tau_1 \hookrightarrow_{\widetilde{p}_1 \widetilde{\overline{p}}} \tau}{\Gamma, \widetilde{p} \vdash \ [\widetilde{e}]^{\tau}_{\widetilde{p}_1 \widetilde{\overline{p}}} : \widetilde{\widetilde{\Delta}}_{\widetilde{p}}(\tau)}\ [\text{GT-Emb}]$$

Figure 11: Gradual Type System

$\boxed{\tau \hookleftarrow_{\widetilde{l}} \tau'}$

$$\frac{\widetilde{\widetilde{\Delta}}_{\widetilde{p}}(\tau) = \widetilde{\widetilde{\Delta}}_{\widetilde{p}}(\tau')}{\tau \hookleftarrow_{\widetilde{p}} \tau'}\ [\text{T-eq}] \qquad \frac{\tau \hookleftarrow_{\widetilde{l}} \widetilde{\widetilde{\Delta}}_{\widetilde{p}}(\tau')}{\tau \hookleftarrow_{\widetilde{l}\widetilde{p}} \tau'}\ [\text{T-trans}]$$

Figure 12: Gradual collaborative principals knowledge relation $\tau \hookleftarrow_l \tau'$

We have some changes in the gradual type system respect to the static type system. We have a gradual principal in the typing context. We also have gradual terms instead of static terms. We also define $\tau \hookleftarrow_{\widetilde{l}} \tau'$, the gradual version of the collaborative principals knowledge relation $\tau \hookleftarrow_{l} \tau'$.

Now we explain how the gradual type system works. We typecheck the examples of Figures **??** and **??**. In the example of Figure **??** the important part is to type check the expression $\lambda x : \texttt{int}.\ g\ x.$. We show the typing derivations for the expression $\lambda x : \texttt{int}.\ [\lambda x : \texttt{int}.\ toStr\ x]_h^{\texttt{fh}\to\texttt{str}}\ x$, where g is substituted by its definition.

$$
\begin{array}{ll}
\text{[GT-Abs]} & \Gamma, ? \vdash \lambda x : \texttt{int}.\ [\lambda x : \texttt{int}.\ toStr\ x]_h^{\texttt{fh}\to\texttt{str}}\ x : \texttt{int} \to \texttt{str} \\
\text{[GT-App]} & \quad \Gamma\,[x : \texttt{int}], ? \vdash [\lambda x : \texttt{int}.\ toStr\ x]_h^{\texttt{fh}\to\texttt{str}}\ x : \texttt{str} \\
\text{[GT-Emb]} & \quad\quad \Gamma\,[x : \texttt{int}], ? \vdash [\lambda x : \texttt{int}.\ toStr\ x]_h^{\texttt{fh}\to\texttt{str}} : \texttt{int} \to \texttt{str} \\
\ldots & \quad\quad\quad \emptyset, h \vdash \lambda x : \texttt{int}.\ toStr\ x : \texttt{int} \to \texttt{str} \\
\ldots & \quad\quad\quad \ldots \\
\text{[GT-Var]} & \quad\quad \Gamma\,[x : \texttt{int}], ? \vdash x : \texttt{int} \\
 & \quad\quad \texttt{int} =_? \texttt{int}
\end{array}
$$

The above example is well typed and the key point is that the unknown principal refines that the type of the embedding from $\texttt{fh} \to \texttt{str}$ to $\texttt{int} \to \texttt{str}$. This is possible because there exists a principal, the host, that knows that $\texttt{fh} = \texttt{int}$. We do not show all the derivation tree, but the most important parts.

In the case of the example in Figure **??** the program is not well-typed because of the rule [GT-Let]:

$$\Gamma, \mathsf{world} \vdash \mathit{let}_?\ f : \texttt{int} \to \texttt{str} = \ \lambda x : \texttt{int}.g\ x\ \mathit{in}\ f\ 1$$

Basically, the side condition of this rule does not hold for the types of the identifier and the named expression, $\texttt{int} \to \texttt{fh} \mathrel{\widetilde{=}}_? \texttt{int} \to \texttt{int}$. The problem is that there is no principal that knows $\texttt{fh} = \texttt{int}$.

5 Intermediate language

In this section we present an intermediate cast calculus to execute our gradual principal calculus. We reuse the ideas of Siek and Taha [**?**] to present a type directed translation from the gradual calculus to the intermediate cast calculus. The dynamic semantics of the intermediate calculus gives a mean to execute a program with gradual principals and the inserted casts ensure dynamically that type abstraction is not violated.

5.1 Syntax

Figure **??** shows the intermediate language syntax. The syntax does not include the `let` expression because it disappears after the translation. The syntax is basically the same as $\lambda^p_{[.]}$. We add a cast expression $\langle \tau_1 \Leftarrow \tau_2 \rangle\, e$ for checking that the principal in context knows that both types are equal. We add the error `AbsViolation` to expressions. This is the error one gets when a cast fails.

In the intermediate language, embeddings can be annotated with the unknown principal. We explain how to reduce an embedding of the unknown principal in Section **??**, but intuitively the expression inside the embedding is reduced using the principal of the context in which the embedding is.

$$\text{(terms)} \quad e \quad ::= \quad ... \mid \langle \tau_1 \Leftarrow \tau_2 \rangle \mid [e]^{\tau}_{\widetilde{l}} \mid \texttt{AbsViolation}$$

Figure 13: Syntax of the intermediate language

5.2 Type system for the intermediate language

We present the type system for the intermediate language in Figure **??**. We only write the rules for the new expression, because the other rules remains equal as those of the static semantics of the gradual calculus. The [IT-Cast] rule expresses that a cast expression is always well-typed. The [IT-Error] gives a non-determinist type to an error, which expresses that an error can have any expected type.

$\boxed{\Gamma, p \vdash e : T}$

$$\frac{}{\Gamma, p \vdash \texttt{AbsViolation} : \tau} \text{ [IT-Error]}$$

$$\frac{\Gamma, p \vdash e : \tau_2}{\Gamma, p \vdash \langle \tau_1 \Leftarrow \tau_2 \rangle \ e : \tau_1} \text{ [IT-Cast]}$$

Figure 14: Type system for the intermediate language

5.3 Translation to intermediate language

Following the ideas of Siek and Taha [**?**] we provide a translation from the gradual calculus to the intermediate cast calculus. The cast expression appears when the unknown principal is in context and when a `let` expression is annotated with the unknown principal. We define a helper meta function *insert-cast*? for inserting cast. This function is similar to the function *insert-has*? used by Bañados et al. [**?**]. We avoid to insert a cast if the unknown principal is not assuming the representation knowledge about an abstract type:

$$\textit{insert-cast?}(p, \tau_1, \tau_2, e) = \begin{cases} \langle \tau_1 \Leftarrow \tau_2 \rangle \ e & p =? \wedge \tau_1 \neq \tau_2 \\ e & \textit{otherwise} \end{cases}$$

Figure **??** shows the translation judgment $\Gamma, \widetilde{p} \vdash \widetilde{e} \Rightarrow e' : \tau$, which can be read as: given the expression $\widetilde{e}$ in the context of principal $\widetilde{p}$ we translate $\widetilde{e}$ to e' maintaining the type τ. The most important rules are [CI-App] and [CI-Let?]. The rule [CI-App] uses the meta function *insert-cast*? that inserts a cast

expression in the case that the principal in context is the unknown principal and $\tau_1 1$ and $\tau_1 2$ are not equal. The rule [CI-Let?] only applies when the annotated principal of the `let` expression is the unknown principal. This rule possibly inserts the cast expression to check that the annotated type in the let expression and the type of the named expression must be equal. This rule produces the expression an embedding with the unknown principal to indicate the expression e belongs to the unknown principal.

$\boxed{\Gamma, \widetilde{p} \vdash \widetilde{e} \Rightarrow e' : \tau}$

$$\frac{}{\Gamma, \widetilde{p} \vdash x \Rightarrow x : \widetilde{\bar{\Delta}}_{\widetilde{p}}(\Gamma(x))}\ [\text{CI-Var}] \qquad \frac{}{\Gamma, \widetilde{p} \vdash b \Rightarrow b : B}\ [\text{CI-Const}]$$

$$\frac{\Gamma, \widetilde{p} \vdash \widetilde{e}_1 \Rightarrow e_1' : \tau_{11} \to \tau \quad \Gamma, \widetilde{p} \vdash \widetilde{e}_2 \Rightarrow e_2' : \tau_{12} \quad \tau_{11} \widetilde{=}_p \tau_{12}}{\Gamma, \widetilde{p} \vdash \widetilde{e}_1 \widetilde{e}_2 \Rightarrow e_1' \boxed{\mathit{insert\text{-}cast?}(p, \tau_{11}, \tau_{12}, e_2')} : \tau}\ [\text{CI-App}]$$

$$\frac{\Gamma[x : \tau'], \widetilde{p} \vdash \widetilde{e} \Rightarrow e' : \tau}{\Gamma, \widetilde{p} \vdash \lambda x : \tau'.\, \widetilde{e} \Rightarrow \lambda x : \tau'.\, e' : \tau' \to \tau}\ [\text{CI-Abs}]$$

$$\frac{\Gamma, ? \vdash \widetilde{e_1} \Rightarrow e_1' : \tau_1 \quad \tau \widetilde{=}_? \tau_1 \quad \Gamma[x : \tau], \widetilde{p} \vdash \widetilde{e_2} \Rightarrow e_2' : \tau_2}{\Gamma, \widetilde{p} \vdash \mathit{let}_? \; x : \tau = \; \widetilde{e_1} \; \mathit{in} \; \widetilde{e_2} \Rightarrow (\lambda x : \tau.\, e_2')(\boxed{\mathit{insert\text{-}cast?}(?, \tau, \tau_1, \llbracket e_1' \rrbracket_?^{\tau}}\,)) : \tau_2}\ [\text{CI-Let?}]$$

$$\frac{p \neq ? \quad \Gamma, p \vdash \widetilde{e_1} \Rightarrow e_1' : \tau_1 \quad \bar{\Delta}_p(\tau) = \tau_1 \quad \Gamma[x : \tau], \widetilde{p} \vdash \widetilde{e_2} \Rightarrow e_2' : \tau_2}{\Gamma, \widetilde{p} \vdash \mathit{let}_p \; x : \tau = \; \widetilde{e_1} \; \mathit{in} \; \widetilde{e_2} \Rightarrow (\lambda x : \tau.\, e_2')(\llbracket e_1' \rrbracket_p^{\tau}) : \tau_2}\ [\text{CI-LetS}]$$

Figure 15: Type-directed translation to the intermediate language

5.4 Dynamic Semantics for cast calculus

In this section we present the runtime semantics for the intermediate language. The runtime semantics give a meaning to execute a gradual program. Figure **??** shows the most important rules for the dynamic semantics of the intermediate cast calculus respects to that of the $\lambda^p_{[\cdot]}$. Equivalents rules to [EApp1], [EApp], [EApp2], [ELet1], [EEmbr1], [EEmbr2], [EEmbrO], [EEmbrC], [EEmbrF] are omitted because are very similar. We do not have equivalent rules for the [ELet1] and [ELet2] because the intermediate cast calculus does not have a `let` expression.

The unknown principal never executes an expression. The reason is than when we execute an embedding, which is annotated with the unknown principal, we replace the unknown principal for the principal in context (rule [IE-Emb?]). The fact that the top level principal is world and it is always known makes no possible to have the unknown principal in the evaluation context. An alternative dynamic semantics can be formulated considering to have a top level unknown principal, but that is left for future work.

$$\frac{}{\langle p, [e]^{\tau}_{?}\rangle \longmapsto \langle p, [e]^{\tau}_{p}\rangle}\ \text{[IE-Emb?]}$$

$$\frac{\langle p, e\rangle \longmapsto \langle p, e'\rangle}{\langle p, \langle \tau_2 \Leftarrow \tau_1\rangle\, e\rangle \longmapsto \langle p, \langle \tau_2 \Leftarrow \tau_1\rangle\, e'\rangle}\ \text{[IE-CastR]}$$

$$\frac{\tau_1 =_p \tau_2}{\langle p, \langle \tau_2 \Leftarrow \tau_1\rangle\, v\rangle \longmapsto \langle p, v\rangle}\ \text{[IE-Cast]}$$

$$\frac{\tau_1 \neq_p \tau_2}{\langle p, \langle \tau_2 \Leftarrow \tau_1\rangle\, v\rangle \longmapsto \texttt{AbsViolation}}\ \text{[IE-CastE]}$$

Figure 16: Dynamic Semantics for the internal language

The rule [IE-CastR] reduces the expression protected by the cast. Once we have a value with a cast, the [IE-Cast] or [IE-CastE] rules could apply. The rule [IE-Cast] applies when the principal in context knows that both types are equal and in this case we obtain the value. The rule [IE-CastE] applies when the principal does not know the abstraction between both types and in this case an abstraction violation error is raised.

5.5 Examples

In this section we show two examples to clarify the process for evaluating a gradual program. This process involves the type directed translation of the gradual programs to the intermediate calculus and then the execution of the translated program using the runtime semantics of the intermediate calculus. We reuse the example of Figure **??**. We only show the most important translation steps to convert the gradual program to a program in the internal language. We omit the type environment Γ in the notation because is not important to see where the cast insertion happens. We start with the expression:

$$h \vdash \mathit{let}_{?}\ f : \texttt{int} \to \texttt{str} = \ \lambda x : \texttt{int}.g\ x\ \mathit{in}\ \mathrm{f}\ 1$$

The next step is to transform the named expression. For the translation of the named expression the unknown principal is in context, which means that a cast could be inserted:

$$? \vdash \lambda x : \texttt{int}.g\ x$$

In fact a cast is inserted because function g has type $\texttt{fh} \to \texttt{str}$ and it is applied with `int`, so the rule [CI-App?]applies and a cast is inserted:

$$? \vdash \lambda x : \texttt{int}.g\ \ \langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x$$

Once we have the translation for the named expression, we use the rule [CI-Let?] over the expression:

$$h \vdash \mathit{let}_{?}\ f : \texttt{int} \to \texttt{str} = \ \lambda x : \texttt{int}.g\ \ \langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x\ \mathit{in}\ \mathrm{f}\ 1$$

to get the final program in the intermediate language:

$$(\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)\,[\lambda x : \texttt{int}.g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x]^{\texttt{str}}_{?}$$

Now we execute the above program with the runtime semantics of the intermediate language. In this example we assume that the execution takes place in the host context.

		$\langle \mathsf{h}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)\,[\lambda x : \texttt{int}.g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x]^{\texttt{str}}_{?}\rangle$
1	[IE-Emb?]	$\langle \mathsf{h}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)\,[\lambda x : \texttt{int}.g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x]^{\texttt{str}}_{h}\rangle$
2	[IE-EmbF]	$\langle \mathsf{h}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)(\lambda x_1 : \texttt{int}.[g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [x_1]^{\texttt{int}}_{h}]^{\texttt{str}}_{h})\rangle$
3	[IE-App]	$\langle \mathsf{h}, (\lambda x_1 : \texttt{int}.[g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [x_1]^{\texttt{int}}_{h}]^{str}_{h})\ 1\rangle$
4	[IE-App]	$\langle \mathsf{h}, [g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [1]^{\texttt{int}}_{h}]^{str}_{h}\rangle$
5	[IE-EmbO]	$\langle \mathsf{h}, [g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\ 1]^{str}_{h}\rangle$
6	[IE-Cast]	$\langle \mathsf{h}, [g\ 1]^{str}_{h}\rangle$
	...	...
	...	$[\text{“1”}]^{str}_{h}$
	[IE-EmbO]	“1”

The most important steps are step 1 and 6. In step 1, we replace the unknown principal of the embedding for the principal in context (host). This means that the expression inside the embedding now belongs to the principal host and the host is the one that must execute the cast. This happens in the step 6, and the cast succeeds because the host knows that `fh`=`int`. The other evaluation steps are not interesting and finally the host gets a `string`, which is the literal representation of 1.

Let us see the same program but executed in the client context. We omit the translation process because, for this example, it is identical to the one we do for the host. The reduction rules are presented below:

		$\langle \mathsf{c}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)\,[\lambda x : \texttt{int}.g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x]^{\texttt{str}}_{?}\rangle$
1	[IE-Emb?]	$\langle \mathsf{c}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)\,[\lambda x : \texttt{int}.g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, x]^{\texttt{str}}_{c}\rangle$
2	[IE-EmbF]	$\langle \mathsf{c}, (\lambda f : \texttt{int} \to \texttt{str}.\ \mathrm{f}\ 1)(\lambda x_1 : \texttt{int}.[g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [x_1]^{\texttt{int}}_{c}]^{\texttt{str}}_{c})\rangle$
3	[IE-App]	$\langle \mathsf{c}, (\lambda x_1 : \texttt{int}.[g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [x_1]^{\texttt{int}}_{c}]^{str}_{c})\ 1\rangle$
4	[IE-App]	$\langle \mathsf{c}, [g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\, [1]^{\texttt{int}}_{c}]^{str}_{c}\rangle$
5	[IE-EmbO]	$\langle \mathsf{c}, [g\,\langle \texttt{fh} \Leftarrow \texttt{int}\rangle\ 1]^{str}_{c}\rangle$
6	[IE-Cast]	`AbsViolation`

The evaluation steps from 1 to 5 are the same, but now we are in the client context. That is the reason we replace the unknown principal by the client principal in the step 1. The difference here is in the step 6. The client does not know that `fh` = `int` and that is the reason we get the `AbsViolation` error.

6 Conclusion

We have developed a gradual calculus that makes it possible to strengthen type aliases to abstract types in a gradual way. We developed this gradual calculus

over a variation of the calculus of Grossman et al. [**?**].

We use the ideas of AGT to derive the static semantics, but it remains to study if AGT can help design the runtime semantics. In this work we provide the dynamic semantics for the gradual language in the style of Siek and Taha [**?**], which is to compile the gradual language to an internal language with casts and to define the runtime semantics for the internal language.

The gradual approach presented in this paper is not complete, as we have not proven the expected formal results of a gradual language, such as a) the gradual language is type safe, b) the gradual language is a conservative extension of the static language, and c) the language satisfies the gradual guarantee of Siek et al. [**?**]. These formal results are the next step for future work.