Dynamic Opacity for Abstract Types

Andreas Rossberg*
Universität des Saarlandes
rossberg@ps.uni-sb.de

ABSTRACT

Existential types are the standard formalisation of abstract types. While this formulation is sufficient in entirely statically typed languages, it proves to be too weak for languages enriched with forms of dynamic typing: in the presence of operations performing type analysis, the abstraction barrier erected by the static typing rules for existential types is no longer impassable, because parametricity is violated. We present a light-weight calculus for polymorphic languages with abstract types that addresses this shortcoming. It features a variation of existential types that retains most of the simplicity of standard existentials. It relies on modified scoping rules and explicit coercions between the quantified variable and its witness type.

1. INTRODUCTION

Type abstraction is an important tool for structuring programs and is a fundamental feature of most module systems. Languages like ML [20, 16], Modula [36, 4], CLU [19] and Ada [13] provide features for specifying abstract types, either directly or by means of their module systems. Generally speaking, an abstract type is declared in two parts: its *signature* and an *implementation*. The former usually allows to declare a name for the abstract type and specifies the operations available on values of that type, while the latter fixes a *representation type* for those values and implements the signature's operations accordingly. The key property is that the representation type remains private: the sole way to create or access values of the abstract type from the outside is by going through the operations listed in the signature.

For illustration purposes we will use (a subset of) the Standard ML module language. In SML, an abstract type's signature can be specified by a signature declaration. Consider the common example of a complex number type:

```
signature COMPLEX =
sig
   type complex
   val mk : real * real -> complex
   val real : complex -> real
   val imag : complex -> real
   val mul : complex * complex -> complex
end
```

An implementation of a corresponding abstract type is provided by a structure declaration:

```
structure C :> COMPLEX =
struct
   type complex = real * real (* polar *)
   fun mk(x,y) = (sqrt(x*x+y*y), atan2(y,x)+pi)
   fun real(a,th) = a * cos th
   fun imag(a,th) = a * sin th
   fun mul((a1,th1), (a2,th2)) =
       (a1*a2, rem(th1+th2, 2*pi))
end
```

An alternative implementation might use cartesian representation for complex numbers. In any case, the abstraction operator `:>` hides the representation type `real * real` in the sense that to the outside, type `complex` is different from `real * real` — or any other type, for that matter. The operations exported through the signature are the only means to create and compose complex numbers.

The advantage of the encapsulation idea implemented by type abstraction is twofold. First, the use of type abstraction enforces loose coupling between an abstract's type implementation and any client code: client code is compelled not to rely on internals of the representation. The implementation may thus be modified freely without breaking any existing client code, as long as the signature (and the semantics of its operations) remains the same. Even more important is a second advantage: the type system guarantees that values of abstract type cannot be forged by clients. Such a guarantee is an essential prerequisite for enabling implementations to maintain invariants on their representations and their internal state. For example, our complex implementation preserves the invariant that the argument θ (`th`) of the complex number is always normalized to $\theta \in [0; 2\pi[$. The number of cases that must be handled in the implementation of some operations is reduced. Other examples of abstract types might be impossible to implement correctly

*This research is funded by the Deutsche Forschungsgemeinschaft (DFG) as part of SFB 378: Ressourcenadaptive kognitive Prozesse, Project NEP: Statisch getypte Programmierumgebung für nebenläufige Constraints

at all without the ability to ensure invariants. Consider an abstract time stamp type.

In their classic paper, Mitchell and Plotkin showed that abstract types could be formalised naturally as existential types [23], using the standard typing rules for existential types as found in constructive logic (e.g. System F [9]). Recent theories for ML-style module systems [29, 6] also capture type abstraction by existential quantification. We will review the basic rules of existential types and their relationship to abstract types in section 2.

1.1 Dynamic type analysis

Constructs for (dynamic) type analysis have been formulated in different flavours. Examples are dynamics [1, 17], intensional type analysis [12] and extensional polymorphism [8]. They have in common that there is some form of typecase expression that allows branching dependent on a type that is determined dynamically.

Let us consider an extension of SML with typecase. In order to simplify the presentation we use a very simple variant throughout this paper. Our typecase does not bind any type variables, but merely allows the type of an expression to be compared to a second type:

`typecase` exp_1 `:` τ_1 `of` x `:` τ_2 `then` exp_2 `else` exp_3

The intuitive semantics of this expression form is that it evaluates to $exp_2[x := exp_1]$ iff $\tau_1 = \tau_2$ dynamically, to exp_3 otherwise. That semantics will be made more precise in section 3. A simple example for using a typecase might be a simplistic polymorphic string conversion function:

```
fun 'a toString (x : 'a) =
    typecase x : 'a of x' : int
      then Int.toString x'
    else typecase x : 'a of x' : real
      then Real.toString x'
    else typecase x : 'a of x' : bool
      then Bool.toString x'
    else "_"
```

By applying this function to some arbitrary value v the polymorhic type variable `'a` will be instantiated to a concrete (dynamic) type τ, the type of v. The function will properly dispatch on that type and delegate the conversion task to a suitable library function, if available.

Now, the interesting question is, how does typecase interact with type abstraction? What happens, if we try to evaluate the following expression:

```
typecase C.mk(0.0, 1.0) : C.complex
  of p : real * real
  then print("theta = " ^ Real.toString(#2 p))
  else raise CouldntAccessRepresentation
```

Or even worse:

```
typecase (1.0, 1001.0*pi) : real * real
  of z : C.complex
  then z
  else raise CouldntAccessRepresentation
```

It is obvious that in both cases the `else` branch should be chosen. Or is it? Unfortunately, this is not the answer the standard model of abstract types using existential quantification will give! The reasons will become appearent in section 3.1. In fact, it is well-known that existential abstraction can be broken in the presence of primitives for type analysis, because the presence of the latter causes loss of the parametricity property [33, 28, 2] its encapsulation power relies on. Weirich demonstrated that in a non-parametric setting arbitrary values of existential type can be cast back and forth to and from their actual representation type [35]. While such a cast is still type-safe in the sense of not violating soundness, it clearly undermines any of the previously mentioned additional guarantees the type system should make about abstract types — the first expression above is coupled to internals of the complex representation, while the second even breaks its θ-invariant. Because type abstraction is no longer sufficient to ensure encapsulation, it is practically rendered useless.

1.2 Requirements

How can the conflict be solved? A simple possibility is to forbid analysis of abstract types altogether. That approach has been suggested by Harper and Morrisett, who curtly propose using the kind system to distinguish between analyzable and non-analyzable types [12]. However, this not only represents a considerable complication of the type theory (besides subkinding, kind polymorphism is probably required to regain a useful level of flexibility for higher-order types), it also is too restrictive. For example, it would disallow us to extend the string conversion function to handle complex numbers, by rendering the following code illegal:

```
fun 'a toString (x : 'a) =
    typecase x : 'a of x' : C.complex then
       Real.toString(real x') ^
       (if imag x' >= 0.0 then "+" else "-") ^
       "i" ^ Real.toString(abs(imag x'))
    else ...
```

Similarly, in a language with type `dynamic`, it became impossible to inject values with abstract type into `dynamic` — or more precisely, to project them out again. Hence, such a solution might seriously impair the usefulness of type analysis as well as the applicability of type abstraction.

This paper thus aims to define a simple formal semantics for type abstraction that is fully compatible with type analysis. In short, we seek a semantics in which the interplay between both features has the following characteristics:

1. *Dynamic Opacity:* an abstract type cannot be identified with any other type through dynamic analysis.
2. *Full Reflexivity:* every type can be inspected dynamically.

Dynamic opacity basically says that the key property of type abstraction ought to carry over from the static type system to dynamic typing — abstract types need to be unaccessible and unforgeable even by dynamic type analysis. The second property effectively means that any type must be comparable (dynamically) to any other type. We borrow the term *full reflexivity* from Trifonov et.al. [34], wo introduced it in a slightly different context to express the absence of any restriction on the *syntactic form* of types that are available for

analysis (no such restricition is necessary in the weak form of typecase we use in this paper).

Taken together, both requirements imply that an abstract type must be different from any other type in the language's universe of types. It clearly follows that type abstraction must have some sort of *generative* semantics: introduction of an abstract type dynamically generates a new type. Without generativity, type abstraction has no dynamic interpretation! In this paper we present the light-weight λ_{up}-calculus that features a variation of existential types that can model type abstraction with the desired properties. Abstraction in that calculus relies on explicit coercions between an abstract type and its representation and incorporates a simple syntactic treatment of generativity.

With respect to the stated requirements it should be noted that we solely discuss dynamic typing intended for programmatic use, i.e. as a language feature available to the programmer in the external language. There are different application domains for type analysis, especially in language implementations for dealing with specialised data representations in the compilation of polymorphic functions (which was the motivation of Harper and Morrisett's work). Such internal use demands for different, incompatible properties. In particular, dynamic opacity is specifically not wanted under such circumstances. We consider external and internal use of dynamic typing as largely independent issues, so that the latter will not be considered in this paper.

1.3 Plan

In section 2 we give a short overview of the polymorphic λ-calculus with existential types and review how it can encode type abstraction. In section 3 we add type analysis and investigate how it interferes with this encoding. In section 4 we introduce the λ_{up}-calculus as an alternative, state its basic properties, and discuss how it achieves dynamic opacity and models generativity. In section 5 we look at the expressiveness of that calculus in comparison to the standard model for existential types. We review some related work in section 6 and conclude in section 7.

2. EXISTENTIAL TYPES

Figure 1 specifies syntax, typing rules and basic reduction rules of $\lambda_\exists$, the polymorphic lambda calculus with existential types. The typing and type well-formedness rules of the calculus use an environment Γ which is a set of type variables $\alpha, \beta, \gamma, \ldots$ and type assignments $x : \tau$ such that all x are disjoint (i.e. it contains a partial function from identifiers to types). We write Γ, α and $\Gamma, x : \tau$ for disjoint extension of environments, i.e. these notations are only considered well-formed if α or x do not occur in Γ, respectively. We always assume well-formedness of given environments and contained types.

The one-step reduction relation $\rightarrow$ is the compatible [3] closure of the primitive rules given in figure 1. The many-step reduction relation $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. We will write $\equiv$ for syntactic equivalence of $\lambda_\exists$-terms and types (modulo α-conversion), and $=$ for convertibility with respect to the corresponding congruence relation generated from $\rightarrow$.

In addition to the standard syntax and rules for plain polymorphic lambda calculus, the $\lambda_\exists$-calculus features an introduction and an elimination form for existential types. An existential type is introduced by an expression of the form $\langle \alpha = \tau_1, e : \tau_2 \rangle$.[1] It encapsulates a type τ_1 and a value of some type τ_2' (which usually contains occurences of τ_1) into an existential type $\exists \alpha.\tau_2$. The type τ_2, which we will call the *signature type*, is obtained from τ_2' (the *implementation type*) by replacing some or all occurences of the *representation type* (or *witness*) τ_1 with the type variable α. Note that the signature type is not determined uniquely by the implementation and representation types alone, thus it has to be specified explicitly. We will speak of a value of existential type as a *package* and call the encapsulated value e its *implementation*, respectively.

In order to do anything interesting with a package, i.e. access the encapsulated implementation, the existential quantifier has to be eliminated. In the expression form (open $\langle \alpha, x \rangle = e_1$ in e_2) the subexpression e_1 denotes a package, whose representation type and implementation can be referred to by the variables α and x within e_2, respectively.[2]

The package expression $\langle \alpha = \tau_1, e : \tau_2 \rangle$ is a binder for the type variable α (which can occur within τ_2). The expression (open $\langle \alpha, x \rangle = e_1$ in e_2) binds x and α (within e_2). Both expression forms are subject to standard α-conversion rules for term and type variables. Rule (OPEN) demands that the type variable bound by the existential quantifier and the one bound by open are the same. This can always be achieved via appropriate α-renaming. Likewise, the respective reduction rule may safely assume that the type variables bound by open and by the corresponding package are the same.

For clarity, we will sometimes use the notation (let $x = e_1$ in e_2) as abbreviation for the expression $(\lambda x : \tau_1.e_2)\ e_1$, where τ_1 is the type of e_1. Moreover, we sometimes use _ for *don't care* variables.

2.1 Encoding Abstract Types

An abstract type declaration introduces a new type bundled with a set of operations available on values of that type. A direct encoding of abstract types via existential types is relatively straight-forward. Let us assume that $\lambda_\exists$ has been enriched with product types and real numbers. Then the signature `COMPLEX` from the introduction can be represented by the type

$$COMPLEX \quad \equiv \quad \exists \gamma.COMPLEX'(\gamma)$$

where

$$COMPLEX'(\gamma) \quad \equiv \quad (real \times real \rightarrow \gamma) \times (\gamma \rightarrow real) \times (\gamma \rightarrow real) \times (\gamma \times \gamma \rightarrow \gamma)$$

That is, the set of operations is mapped to a tuple of appropriate type, and this type is existentially quantified over the type to be hidden by the abstraction. The structure `C` can be modelled as (taking the freedom to use tuple patterns as

[1] Another common syntax for existential introduction is (pack τ_1, e as $\exists \alpha.\tau_2$) and variants thereof.

[2] Often this expression form is written as (unpack e_1 as α, x in e_2).

$$\begin{array}{lrcl} \text{(types)} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \\ \text{(terms)} & e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau \mid \\ & & & \langle \alpha{=}\tau_1, e{:}\tau_2\rangle \mid \text{open } \langle\alpha, x\rangle = e_1 \text{ in } e_2 \end{array}$$

Syntax

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha : \diamond} \qquad \frac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma \vdash \tau_2 : \diamond}{\Gamma \vdash \tau_1 \to \tau_2 : \diamond}$$

$$\frac{\Gamma, \alpha \vdash \tau : \diamond}{\Gamma \vdash \forall\alpha.\tau : \diamond} \qquad \frac{\Gamma, \alpha \vdash \tau : \diamond}{\Gamma \vdash \exists\alpha.\tau : \diamond}$$

$$(\textsc{Id})\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\textsc{Abs})\ \frac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.e) : \tau_1 \to \tau_2}$$

$$(\textsc{App})\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$(\textsc{Gen})\ \frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad (\textsc{Inst})\ \frac{\Gamma \vdash e : \forall\alpha.\tau \qquad \Gamma \vdash \tau' : \diamond}{\Gamma \vdash e\ \tau' : \tau[\alpha := \tau']}$$

$$(\textsc{Seal})\ \frac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma \vdash e : \tau_2[\alpha := \tau_1]}{\Gamma \vdash \langle \alpha{=}\tau_1, e{:}\tau_2\rangle : \exists\alpha.\tau_2}$$

$$(\textsc{Open})\ \frac{\Gamma \vdash e_1 : \exists\alpha.\tau' \qquad \Gamma, \alpha, x{:}\tau' \vdash e_2 : \tau}{\Gamma \vdash (\text{open } \langle\alpha, x\rangle = e_1 \text{ in } e_2) : \tau}(\alpha \notin \text{FV}(\tau))$$

Typing

$$\begin{array}{rcl} (\lambda x{:}\tau.e_1)\ e_2 & \to & e_1[x := e_2] \\ (\Lambda\alpha.e)\ \tau & \to & e[\alpha := \tau] \\ \text{open } \langle\alpha, x\rangle = \langle \alpha{=}\tau_1, e_1{:}\tau_2\rangle \text{ in } e_2 & \to & e_2[\alpha := \tau_1][x := e_1] \end{array}$$

Reduction

Figure 1: The $\lambda_\exists$ calculus

λ-parameters):

$$\begin{aligned} C \quad \equiv \quad & \langle \gamma{=}\mathit{real} \times \mathit{real}, \\ & (\lambda(x,y) : \mathit{real} \times \mathit{real}\ .\ (\sqrt{x^2+y^2}, \arctan(y/x) + \pi), \\ & \lambda(a,\theta) : \mathit{real} \times \mathit{real}\ .\ a \cdot \cos\theta, \\ & \lambda(a,\theta) : \mathit{real} \times \mathit{real}\ .\ a \cdot \sin\theta, \\ & \lambda((a_1,\theta_1),(a_2,\theta_2)) : (\mathit{real} \times \mathit{real}) \times (\mathit{real} \times \mathit{real})\ . \\ & \qquad (a_1 \cdot a_2,\ \text{rem}(\theta_1 + \theta_2, 2\pi)) \\ &) : \mathit{COMPLEX}'(\gamma)\rangle \end{aligned}$$

It is easy to see that the type of C will be *COMPLEX* according to the typing rules of $\lambda_\exists$.

ML style module access does not map as directly to $\lambda_\exists$, because structure components are accessed using the dot notation, while the package has to be opened explicitly to make its content available. For example, in SML one just writes

```
val a = C.real(C.mk(0.0, 1.0))
```

while the $\lambda_\exists$-encoding requires an open expression, so that one can access the contained type and value and to have the abstract type properly shared:

$$a \quad \equiv \quad \text{open } \langle \gamma, (\mathit{mk}, \mathit{real}, \_, \_)\rangle = C \text{ in } \mathit{real}\ (\mathit{mk}\ (0,1))$$

However, Cardelli and Leroy have shown that there exists a systematic translation from dot notation into plain existential types [5]. The basic idea is to insert an open expression around the innermost expression containing all occurences of C and replace access by references to the variables bound by that open expression. When applied to ML structures it is always possible to open the package directly after its introduction. Consequently, we may refine the encoding of ML structures such that for `C` it yields

$$\begin{aligned} & \text{open } \langle \gamma, (\mathit{mk}, \mathit{real}, \mathit{imag}, \mathit{mul})\rangle = \\ & \qquad \langle \gamma{=}\mathit{real} \times \mathit{real}, (\ldots) : \mathit{COMPLEX}'(\gamma)\rangle \\ & \text{in } p[C.\mathit{complex} := \gamma][C. := \epsilon] \end{aligned}$$

where p is the encoding of the remainder of the program. Dot access is deleted as indicated by the substitution notation (boldly ignoring namespace issues). The declaration of `a` can now simply be represented as

$$\mathit{real}\ (\mathit{mk}\ (0,1))$$

The typing rules for open plus the standard hygiene conventions for bound variables ensure that γ is distinct from any other type variable in the same scope and thus behaves like a "fresh" type. Moreover, they behave as fully abstract types because in the standard λ-calculi every expression is *parametric* [27, 25] in any type variable in scope, meaning that reduction can proceed uniformingly for all possible instantiations. That is particularly true for the body e_2 of an open with respect to the variable α bound by open — its evaluation will never depend on the actual representation type τ of the package being opened, although α is substituted by τ during reduction. That key observation establishes the close relation between existential types and abstract types.

A polymorphic λ-calculus with existential types can encode most aspects of the SML module system, as has been shown by Russo in a slightly different way [30]. In general one will need to extend it to higher-order types, though. But $\lambda_\exists$ is more expressive, since it allows passing around packages, and thus abstract types (as opposed to values of abstract types), in a first-class manner. Several distinct types can be 'generated' from the same package, because elimination need not be coupled with introduction:

$$\begin{aligned} & \text{let } M = \langle \alpha{=}\mathit{int}, 0\rangle \text{ in} \\ & \text{open } \langle \alpha, x\rangle = M \text{ in} \\ & \text{open } \langle \beta, y\rangle = M \text{ in} \\ & \text{open } \langle \gamma, z\rangle = M \text{ in} \\ & \ldots \end{aligned}$$

All types α, β, γ are distinct.

3. DYNAMIC TYPE ANALYSIS

As mentioned in the introduction, we consider only a simple form of type analysis throughout this paper. It still is general enough to demonstrate the fundamental problem.

$$e \quad ::= \quad \ldots \mid \text{typecase } e_1 : \tau_1 \text{ of } x : \tau_2 \text{ then } e_2 \text{ else } e_3$$

$$(\textsc{Typecase}) \quad \frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash \tau_2 : \diamond \\ \Gamma, x : \tau_2 \vdash e_2 : \tau & \Gamma \vdash e_3 : \tau \end{array}}{\Gamma \vdash (\text{typecase } e_1 : \tau_1 \text{ of } x : \tau_2 \text{ then } e_2 \text{ else } e_3) : \tau}$$

$$\begin{array}{rcl} \text{typecase } e_1 : \tau \text{ of } x : \tau \text{ then } e_2 \text{ else } e_3 & \rightarrow & e_2[x := e_1] \\ \text{typecase } e_1 : \tau_1 \text{ of } x : \tau_2 \text{ then } e_2 \text{ else } e_3 & \rightarrow & e_3 \quad (\tau_1 \not\equiv \tau_2) \end{array}$$

Figure 2: A typecase extension

Figure 2 specifies the semantics of our typecase, as an extension to the $\lambda_\exists$-calculus.[3]

Note that simply adding typecase to $\lambda_\exists$ without modifying the reduction relation breaks confluence. Consider the following term for example:

$$(\Lambda\alpha.\lambda x : \alpha.\text{typecase } x{:}\alpha \text{ of } y : \mathit{int} \text{ then } 1 \text{ else } 0)\ \mathit{int}\ 9$$

Depending on whether the Λ or the typecase redex gets reduced first, reduction will terminate in 1 or 0, respectively. Clearly, the types τ_1 and τ_2 in a typecase expression must be "evaluated" in a strict manner to avoid this, i.e. all potential type substitutions stemming from surrounding redexes must have been performed before applying the typecase reduction rules. For simplicity, we hence assume a deterministic call-by-value reduction strategy in the remainder of this section. We omit the straight-forward details of the reduction relation's refinement.

3.1 Interaction with existentials

We have seen that the encoding of abstract types via existentials crucially relies on the parametricity property which holds in standard λ-calculi. However, that property breaks down in the face of operations for type analysis: if a polymorphic function is able to analyse its type argument it obviously is not the case that it will evaluate independently of any conrete instantiation. Similarly, a function that is passed an argument of existentially quantified type might inspect the type encapsulated by the quantifier — the computation can be dependent on the actual representation type. Consider for example the following function:

$$\begin{array}{rcl} \mathit{sneak} & \equiv & \lambda x{:}(\exists\alpha.\alpha)\ .\ \text{open } \langle\alpha, y\rangle = x \text{ in} \\ & & \qquad (\text{typecase } y : \alpha \text{ of } y' : \mathit{int} \text{ then } 1 \text{ else } 0) \end{array}$$

Let $x_1 \equiv \langle\alpha = \mathit{int}, 5 : \alpha\rangle$ and $x_2 \equiv \langle\alpha = \mathit{string}, \text{"Moo"} : \alpha\rangle$ be two values of appropriate existential type $\exists\alpha.\alpha$. Although f does not even depend on its argument's implementation, application yields $\mathit{sneak}\ x_1 = 1$ and $\mathit{sneak}\ x_2 = 0$, respectively. In general, $\mathit{sneak}\ e$ will evaluate to 1 if and only if the package e uses int as its representation type. That of course violates the requirement for dynamic opacity formulated in the introduction.

[3]The displayed rules do not preclude any redundant (i.e. statically determined) applications of typecase. Ruling those out is not particularly difficult but orthogonal to the issues discussed here.

For a more realistic example recall the offending typecase expression from section 1.1 that was incriminated to violate the invariant of the complex representation. Expressed in the $\lambda_\exists$-calculus it may look like follows (assuming existence of a $\perp$ value):

$$\begin{array}{l} \text{open } \langle\gamma, (\mathit{mk}, \mathit{real}, \mathit{imag}, \mathit{mul})\rangle = \langle\gamma = \mathit{real} \times \mathit{real}, \ldots\rangle \text{ in} \\ \text{typecase } (1, 1001\pi) : \mathit{real} \times \mathit{real} \text{ of } z : \gamma \\ \quad \text{then } z \text{ else } \perp \end{array}$$

But upon reduction of open the type variable γ naming the abstract type will be substituted and reveal the actual implementation type:

$$\begin{array}{rl} \rightarrow & \text{typecase } (1, 1001\pi) : \mathit{real} \times \mathit{real} \text{ of } z : \mathit{real} \times \mathit{real} \\ & \quad \text{then } z \text{ else } \perp \end{array}$$

Both types in the typecase are now equal and the construct returns $z = (1, 1001\pi)$ from its left branch having the same static type γ as values properly generated by the complex implementation.

The source of the loss of dynamic abstraction clearly is the reduction rule for open, which substitutes the type variable representing the abstract types with its representation. Obviously, the correspondence between type abstraction as a feature for encapsulation on one side and existential quantification on the other is of purely static nature. Dynamically, there is no encapsulation at all performed by mere introduction of an existential type.

The interference between existential types and type analysis has received only little attention in prior work. Besides the proposal by Harper and Morrisett mentioned in the introduction, Abadi, Cardelli et.al. [1] suggested that dynamic opacity can be achieved by simply replacing the type variable bound by open with a "fresh" type constant during evaluation. This amounts to changing the corresponding reduction rule to:

$$\text{open } \langle\alpha, x\rangle = \langle\alpha = \tau_1, e_1 : \tau_2\rangle \text{ in } e_2 \quad \rightarrow \quad e_2[\alpha := t][x := e_1]$$

where t is a fresh type constant. Obviously, the representation type could no longer be analysed transparently. Unfortunately however, this modification destroys type preservation, as can easily be seen from the following example, which is a simple η-expansion of the expression a given in section 2.1:

$$a' \equiv \text{open } \langle\gamma, (\mathit{mk}, \mathit{real}, \_, \_)\rangle = C \text{ in } (\lambda z{:}\gamma.\mathit{real}\ z)\ (\mathit{mk}\ (0, 1))$$

This term is well-typed, but after applying the above reduction rule it becomes:

$$\rightarrow (\lambda z{:}t.R\ z)\ (M\ (0, 1))$$

with

$$\begin{array}{rcl} R & \equiv & \lambda(a, \theta){:}\mathit{real} \times \mathit{real}\ .\ a \cdot \cos\theta \\ M & \equiv & \lambda(x, y){:}\mathit{real} \times \mathit{real}\ .\ (\sqrt{x^2 + y^2},\ \arctan(y/x) + \pi) \end{array}$$

which is no longer typable — there is a clash between the abstract type t and its respective representation type $\mathit{real} \times \mathit{real}$, which is the argument type of function R. Another problem with their suggestion is that a proper formalisation of the meta-concept of dynamic "freshness" requires introducing a notion of state.

4. THE λ_{up}-CALCULUS

We will take the suggestion by Abadi, Cardelli et al. as a starting point for developing a sound variation of existential types that provides dynamic opacity by implementing generativity. The calculus we present does not incorporate type analysis by itself but is merely aimed at being "compatible" with it. We will discuss the addition of typecase and dynamic opacity later on. In particular, we put no restrictions on evaluation strategies yet.

4.1 Ideas

Looking at the approach taken by Abadi, Cardelli, et al. one question is: how can the type clash witnessed be avoided? Abstract type and representation type cannot be made equal during reduction when we want to guarantee dynamic opacity. The only alternative then is to insert appropriate coercions at the boundaries of the abstraction that allow going from the abstract type to its representation and vice versa. For example, if a package of the form $\langle \alpha = \mathit{int}, \ldots \rangle$ is meant to export an integer i under abstract type α, it must do so using an explicit upward coercion, which we write as $(\text{up}_\alpha\ i)$. Likewise, if some function of type $\alpha \to \tau$ is part of the package's signature, the implementation of this function must perform a downward coercion $(\text{dn}_\alpha\ x)$ on its parameter x to get back its concrete value at type *int*. These coercions shall only be available within the package's implementation, a restriction that can be enforced trivially by means of scoping. Using this approach the substitution of an abstract type variable by the package's representation type can be avoided in the reduction rule for open without losing type preservation.

However, since implementations now contain coercions and those coercions refer to an abstract type variable α, how can we avoid α going out of scope upon elimination of the existential? There needs to be a binder, thus we also have to modify the scoping rules for existentials. The second idea of the λ_{up}-calculus thus is to extrude the scope of a package upon opening it, instead of elimination. Doing so we essentially have modelled generativity, because the type variable introduced by the package will never disappear (except if the package is discarded altogether).

4.2 The calculus

Figure 3 shows the rules of the calculus we will refer to as λ_{up}, which implements the aforementioned ideas. Modifications relative to $\lambda_\exists$ have been high-lighted for easier reference. The calculus uses slightly generalised environments Γ, recording pairs $\alpha \triangleright \tau$ instead of plain type variables (with pairwise disjoint α), which we call *type assertions*. They represent a function mapping abstract type variables to their corresponding representation type. Type assertions are used to type coercions. Coercions are the special function symbols $\text{up}_\alpha : \tau \to \alpha$ and $\text{dn}_\alpha : \alpha \to \tau$, where α is an abstract type and τ the corresponding representation type. Since the typing rule (Seal) for packages is the only place that extends the environment with non-trivial type assertions, coercions are only available within the implementation of the corresponding package (note that unlike $\lambda_\exists$, the λ_{up}-calculus allows an package's type variable α to occur free in its implementation e). All other type variables, in particular those bound by open, are mapped to themselves in the environment. The side conditions in the typing rules (Up) and (Dn) thus ensure that they are not available for coercions.

Coercion functions allow an implementation to perform appropriate type conversions for any value of abstract type α that crosses the abstraction boundaries in either direction. Upward coercions can be considered markers, or constructors, for values of abstract type. They are eliminated only by downward coercions, the corresponding destructors. Consequently, the only evaluation possible with coercions is cancellation: in an expression of the form $\text{dn}_\alpha(\text{up}_\alpha e)$ they cancel out each other and the expression can be reduced to e. The standard hygiene convention for type variables — i.e. all bound variables are pairwise disjoint — guarantees that only coercions belonging to the same abstract type can cancel out each other.

The use of coercions will also uniquely determine the package's signature, so that the explicit annotation necessary in $\lambda_\exists$ can be dropped. The typing rule (Seal) for packages just adds the quantifier, no substitution is taking place. We will see in section 5.1 that an implementation expression can always be constructed such that it possesses the desired signature type.

Recall the complex example. Rewritten in λ_{up} (augmented with product types and reals) it looks as follows:

$$
\begin{aligned}
C' \quad \equiv \quad & \langle \gamma = \mathit{real} \times \mathit{real}, \\
& (\lambda(x,y) : \mathit{real} \times \mathit{real}\ . \\
& \quad \text{up}_\gamma(\sqrt{x^2+y^2}, \arctan(y/x) + \pi), \\
& \lambda z : \gamma\ .\ \text{let}\ (a,\theta) = \text{dn}_\gamma\ z\ \text{in}\ a \cdot \cos\theta, \\
& \lambda z : \gamma\ .\ \text{let}\ (a,\theta) = \text{dn}_\gamma\ z\ \text{in}\ a \cdot \sin\theta, \\
& \lambda(z_1, z_2) : \gamma \times \gamma\ . \\
& \quad \text{let}\ (a_1,\theta_1) = \text{dn}_\gamma\ z_1\ \text{in} \\
& \quad \text{let}\ (a_2,\theta_2) = \text{dn}_\gamma\ z_2\ \text{in} \\
& \quad \text{up}_\gamma\ (a_1 \cdot a_2,\ \text{rem}(\theta_1 + \theta_2, 2\pi)) \\
&)\ \rangle
\end{aligned}
$$

Instead of explicitely annotating the desired signature type $\mathit{COMPLEX}'(\gamma)$, it is uniquely determined by the uses of γ, up_γ and dn_γ in the implementation.

The crucial change that will make analysis of a package's representation type impossible, is that the substitution of the bound type variable α that was present in $\lambda_\exists$'s reduction rule for open expressions has simply been removed in λ_{up}.[4] To obtain a valid binding structure, the reduction rule for open incorporates the second aforementioned change: the reduct of an open expression is again a package, retaining the binder from the eliminated one. The same type variable is bound, so the reduct is closed with respect to that variable and the attached coercions. This is also reflected in the typing rule (Open). We will discuss the implications of this semantics in section 4.5.

4.3 Basic properties

[4]Effectively, type variables naming abstract types are no longer true variables since they never get substituted. For simplicity and closer correspondence to $\lambda_\exists$ we refrained from introducing a separate notion of type names, nevertheless.

$$\begin{array}{lrcl} \text{(types)} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \\ \text{(terms)} & e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau \mid \langle \alpha{=}\tau,\ e\rangle \mid \\ & & & \text{open } \langle\alpha, x\rangle = e_1 \text{ in } e_2 \mid \mathrm{up}_\alpha\ e \mid \mathrm{dn}_\alpha\ e \end{array}$$

Syntax

$$\frac{\alpha \rhd \tau \in \Gamma}{\Gamma \vdash \alpha : \diamond} \qquad \frac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma \vdash \tau_2 : \diamond}{\Gamma \vdash \tau_1 \to \tau_2 : \diamond}$$

$$\frac{\Gamma, \alpha \rhd \alpha \vdash \tau : \diamond}{\Gamma \vdash \forall\alpha.\tau : \diamond} \qquad \frac{\Gamma, \alpha \rhd \alpha \vdash \tau : \diamond}{\Gamma \vdash \exists\alpha.\tau : \diamond}$$

$$(\textsc{Id})\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\textsc{Abs})\ \frac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.e) : \tau_1 \to \tau_2}$$

$$(\textsc{App})\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$(\textsc{Gen})\ \frac{\Gamma, \alpha \rhd \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad (\textsc{Inst})\ \frac{\Gamma \vdash e : \forall\alpha.\tau \qquad \Gamma \vdash \tau' : \diamond}{\Gamma \vdash e\ \tau' : \tau[\alpha := \tau']}$$

$$(\textsc{Seal})\ \frac{\Gamma \vdash \tau' : \diamond \qquad \Gamma, \alpha \rhd \tau' \vdash e : \tau}{\Gamma \vdash \langle \alpha{=}\tau', e\rangle : \exists\alpha.\tau}$$

$$(\textsc{Open})\ \frac{\Gamma \vdash e_1 : \exists\alpha.\tau' \qquad \Gamma, \alpha \rhd \alpha, x{:}\tau' \vdash e_2 : \tau}{\Gamma \vdash (\text{open } \langle\alpha, x\rangle = e_1 \text{ in } e_2) : \exists\alpha.\ \tau}$$

$$(\textsc{Up})\ \frac{\Gamma \vdash e : \tau \quad \alpha \rhd \tau \in \Gamma}{\Gamma \vdash \mathrm{up}_\alpha\ e : \alpha}(\tau \not\equiv \alpha)$$

$$(\textsc{Dn})\ \frac{\Gamma \vdash e : \alpha \quad \alpha \rhd \tau \in \Gamma}{\Gamma \vdash \mathrm{dn}_\alpha\ e : \tau}(\tau \not\equiv \alpha)$$

Typing

$$\begin{array}{rcl} (\lambda x{:}\tau.e_1)\ e_2 & \to & e_1[x := e_2] \\ (\Lambda\alpha.e)\ \tau & \to & e[\alpha := \tau] \\ \text{open } \langle\alpha, x\rangle = \langle\alpha{=}\tau, e_1\rangle \text{ in } e_2 & \to & \langle\alpha{=}\tau, e_2[x := e_1]\rangle \\ \mathrm{dn}_\alpha(\mathrm{up}_\alpha e) & \to & e \end{array}$$

Reduction

Figure 3: The λ_{up} calculus

It is fairly standard to show that the following properties about typing hold for λ_{up}:

Proposition (Unique Types) *Whenever* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau \equiv \tau'$.

Proposition (Preservation) *If* $\Gamma \vdash e : \tau$ *and* $e \to^* e'$, *then* $\Gamma \vdash e' : \tau$.

We define λ_{up}-values as follows:

$$v \quad ::= \quad \lambda x{:}\tau.e \mid \Lambda\alpha.e \mid \langle\alpha{=}\tau, v\rangle \mid \mathrm{up}_\alpha v$$

Using that definition we can state progress:

Proposition (Progress) *For every expression* e *such that* $\emptyset \vdash e : \tau$ *for some* τ *(i.e.* e *is closed and well-typed), then either* $e \equiv v$ *for some value* v, *or* $e \to e'$ *for some expression* e'.

Note that if v is closed w.r.t. type variables, then $v \not\equiv \mathrm{up}_\alpha v'$.

Taken together, Preservation and Progress establish type soundness for λ_{up}. Reduction properties are not difficult to show either:

Proposition (Termination) *There are no infinite reduction sequences.*

Proposition (Confluence) *Whenever there are expressions* e, e_1, e_2 *such that* $e \to^* e_1$ *and* $e \to^* e_2$, *then their exists an expression* e' *with* $e_1 \to^* e'$ *and* $e_2 \to^* e'$.

Proofs of all propositions can be found in the appendix. As an example, here is how evaluation can proceed for the example a' manifesting a type clash in section 3.1:

$$\begin{array}{l} \underline{\text{open } \langle\gamma, (mk, real, \_, \_)\rangle = C \text{ in } (\lambda z{:}\gamma.real\ z)\ (mk\ (0,1))} \\ = \langle\gamma{=}real \times real,\ \underline{(\lambda z{:}\gamma.R'\ z)\ (M'\ (0,1))}\rangle \\ = \langle\gamma{=}real \times real,\ \underline{(\lambda z{:}\gamma.R'\ z)\ \overline{(\mathrm{up}_\gamma(1, \pi/2))}}\rangle \\ = \langle\gamma{=}real \times real,\ \overline{R'\ (\mathrm{up}_\gamma(1, \pi/2))}\rangle \\ = \langle\gamma{=}real \times real,\ \overline{\text{let } (a, \theta) = \mathrm{dn}_\gamma(\mathrm{up}_\gamma(1, \pi/2)) \text{ in } a \cdot \cos\theta}\rangle \\ = \langle\gamma{=}real \times real,\ \overline{1 \cdot \cos(\pi/2)}\rangle \\ = \langle\gamma{=}real \times real,\ 0\rangle \end{array}$$

with

$$\begin{array}{rcl} R' & \equiv & \lambda z{:}\gamma.\text{let } (a, \theta) = \mathrm{dn}_\gamma\ z \text{ in } a \cdot \cos\theta \\ M' & \equiv & \lambda(x, y){:}real \times real.\mathrm{up}_\gamma(\sqrt{x^2 + y^2}, \arctan(y/x) + \pi) \end{array}$$

Obviously, every intermediate expression is well-typed.

4.4 Opacity

The motivation for the λ_{up}-calculus was to achieve dynamic opacity for abstract types. To make opacity observable it is necessary to again leave the parametric setting by considering the addition of a type analsyis construct like the one from figure 2. As before, we have to stick to a more rigid evaluation strategy such as call-by-value to preserve confluence in the presence of typecase.

To see how opacity is still preserved in a non-parametric modification of λ_{up} we take a look at the evaluation of function *sneak* from section 3.1. Consider:

$$\begin{array}{ll} & sneak\ \langle\alpha{=}int, \mathrm{up}_\alpha 5\rangle \\ = & \text{open } \langle\alpha, y\rangle = \langle\alpha{=}int, \mathrm{up}_\alpha 5\rangle \text{ in} \\ & \text{typecase } y : \alpha \text{ of } y' : int \text{ then 1 else 0} \\ = & \langle\alpha{=}int,\ (\text{typecase } \mathrm{up}_\alpha 5 : \alpha \text{ of } y' : int \text{ then 1 else 0})\rangle \\ = & \langle\alpha{=}int,\ 0\rangle \end{array}$$

Although the argument uses representation type *int*, evaluation delivers a result of 0.

Likewise, the λ_{up}-encoding of complex numbers is safe with respect to dynamic typing, as the reduction of the expression representing the second offending example from section 1.1

shows:

$$
\begin{array}{ll}
 & \text{open } \langle \gamma, \ldots \rangle = \langle \gamma = \mathit{real} \times \mathit{real}, \ldots \rangle \text{ in} \\
 & \text{typecase } (1, 1001\pi) : \mathit{real} \times \mathit{real} \text{ of } z : \gamma \text{ then } z \text{ else } \bot \\
\to & \langle \gamma = \mathit{real} \times \mathit{real}, \\
 & \quad \text{typecase } (1, 1001\pi) : \mathit{real} \times \mathit{real} \text{ of } z : \gamma \text{ then } z \text{ else } \bot \rangle \\
\to & \langle \gamma = \mathit{real} \times \mathit{real}, \bot \rangle
\end{array}
$$

The variable z keeps its abstract type even after opening the package and the attempt to violate the abstraction via typecase remains unsuccessful.

Types can be abstracted multiple times, introducing a distinct type with every abstraction. Consider:

$$
\begin{array}{l}
\text{open } \langle \alpha, x \rangle = \langle \alpha = \mathit{int}, \lambda x : \mathit{int}.\mathrm{up}_\alpha x \rangle \text{ in} \\
\text{open } \langle \beta, y \rangle = \langle \beta = \alpha, \lambda x : \alpha.\mathrm{up}_\beta x \rangle \text{ in} \\
\text{open } \langle \gamma, z \rangle = \langle \gamma = \alpha, \lambda x : \alpha.\mathrm{up}_\gamma x \rangle \text{ in} \\
\ldots
\end{array}
$$

Values of types $\mathit{int}, \alpha, \beta, \gamma$ will have the following shapes, respectively:

$$
\begin{array}{rcl}
i & : & \mathit{int} \\
\mathrm{up}_\alpha i & : & \alpha \\
\mathrm{up}_\beta(\mathrm{up}_\alpha i) & : & \beta \\
\mathrm{up}_\gamma(\mathrm{up}_\alpha i) & : & \gamma
\end{array}
$$

Abstractions can only be cancelled in the opposite order they have been constructed — there is no way of going from a value of type β back to int without coercing to α first. Both coercions can only be performed by the corresponding implementations. Neither can β be coerced to γ without either abstraction providing suitable operations that allow going through α. So as one would expect, abstractions nest properly on the type *and* term level.

4.5 Generativity

The reduction rule for open expressions may appear a bit odd. Strictly speaking, open is no longer an elimination form in the modified version. It should rather be interpreted as providing a manual form of scope extrusion, shifting existential binders outwards. Each evaluated open expression generates a new package, which can never be eliminated if the expression is not discarded altogether (i.e. not relevant to the result of the computation). All packages actually accessed to compute the result accumulate in it. Consider the following expression, for example:

$$
\begin{array}{l}
\text{let } p = \langle \alpha = \tau_1, e_1 \rangle \text{ in} \\
\text{let } q = \langle \alpha = \tau_2, e_2 \rangle \text{ in} \\
\text{open } \langle \alpha, x \rangle = p \text{ in} \\
\text{open } \langle \beta, y \rangle = q \text{ in} \\
\text{open } \langle \gamma, z \rangle = p \text{ in} \\
e
\end{array}
$$

Its normal form is the value

$$\langle \alpha = \tau_1, \langle \beta = \tau_2, \langle \gamma = \tau_1, v \rangle \rangle \rangle$$

where v is the normal form of e. Every open expression "generates" a new type name. In particular, opening the same package multiple times generates multiple types, which also shows up in the result type:

$$\exists\alpha.\exists\beta.\exists\gamma.\tau$$

with τ being the respective type of e (and v). Thus the calculus captures pretty nicely the generative aspect of type abstraction: abstraction dynamically creates new types, and this generativity is modelled syntactically. However, there are some serious implications we will discuss in the next section.

5. EXPRESSIVENESS

The λ_{up}-calculus features modifications like the requirement for coercions and the change in typing of open expressions that make it non-obvious how its expressiveness relates to the original $\lambda_\exists$-calculus. In this section we show that the former does not pose a restriction, while the latter obviously does.

5.1 A-posteriori abstraction

At first it may look as if the modified form of existential formation featured in λ_{up} is less flexible than the original one, because the injected expression e is required to have the right type τ *a priori*, instead of being sealed *a posteriori*. However, we can systematically construct suitable expressions $e : \tau$ from any given expression $e' : \tau[\tau'/\alpha]$. This is achieved by the transformation rules shown in figure 4.

The transformation is defined inductively over the signature type τ. It recursively constructs an η/open-expansion of the original term e', replacing all subterms e'' of type α with $(\mathrm{up}_\alpha e'')$. Function types require an inverse treatment of the argument, where $e'' : \alpha$ is replaced by $(\mathrm{dn}_\alpha e'')$ instead. The following lemma captures the central invariants of the translation algorithm:

Lemma (A-posteriori Abstraction Invariants) *Let e be a λ_{up}-expression and $\Gamma \equiv \Gamma', \alpha \triangleright \tau'$ an evironment with $\tau' \not\equiv \alpha$.*

1. *If $\Gamma \vdash e : \tau[\alpha := \tau']$, then $\Gamma \vdash \lceil e : \tau \rceil_{\alpha \triangleright \tau'} : \tau$.*
2. *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \lfloor e : \tau \rfloor_{\alpha \triangleright \tau'} : \tau[\alpha := \tau']$.*

Using the transformation we can encode any abstraction in λ_{up} that can be written in the original $\lambda_\exists$-calculus. We can hence view the $\lambda_\exists$ form of existential formation as syntactic sugar in λ_{up}. As an example, it is straight-forward to verify that the $\lambda_\exists$-encoding C of structure `C` exactly transforms to the λ_{up}-expression C' as given in section 4.2, when adding adequate transformation rules for tuples:

$$
\begin{array}{rcl}
\lceil e : \tau_1 \times \tau_2 \rceil_{\alpha \triangleright \tau'} & = & (\lceil e.1 : \tau_1 \rceil_{\alpha \triangleright \tau'}, \lceil e.2 : \tau_2 \rceil_{\alpha \triangleright \tau'}) \\
\lfloor e : \tau_1 \times \tau_2 \rfloor_{\alpha \triangleright \tau'} & = & (\lfloor e.1 : \tau_1 \rfloor_{\alpha \triangleright \tau'}, \lfloor e.2 : \tau_2 \rfloor_{\alpha \triangleright \tau'})
\end{array}
$$

The transformation procedure is similar in spirit to Leroy's boxing/unboxing transformations [14]. The similarity may suggest that it also imposes potential operational overhead similar to the one observed for unboxing operations [21]. In particular, it looks like it implies (at least partial) copying of all data structures that cross abstraction boundaries in either direction. However, it is important to note that coercions are required solely for typing reasons and are identity functions operationally. A type erasing translation of type analysis [7] could reasonably erase all abstractions and their

$$\langle \alpha{=}\tau', e{:}\tau\rangle \quad\rightsquigarrow\quad \langle \alpha{=}\tau', \lceil e : \tau\rceil_{\alpha\triangleright\tau'}\rangle \quad \text{where}$$

$$\begin{array}{lcll}
\lceil e:\alpha\rceil_{\alpha\triangleright\tau'} & = & \mathrm{up}_\alpha e & \\
\lceil e:\alpha'\rceil_{\alpha\triangleright\tau'} & = & e & (\alpha'\not\equiv\alpha)\\
\lceil e:\tau_1\to\tau_2\rceil_{\alpha\triangleright\tau'} & = & \lambda x{:}\tau_1.\lceil e(\lfloor x:\tau_1\rfloor_{\alpha\triangleright\tau'}):\tau_2\rceil_{\alpha\triangleright\tau'} & \\
\lceil e:\forall\alpha'.\tau\rceil_{\alpha\triangleright\tau'} & = & \Lambda\alpha'.\lceil e\ \alpha':\tau\rceil_{\alpha\triangleright\tau'} & \\
\lceil e:\exists\alpha'.\tau\rceil_{\alpha\triangleright\tau'} & = & \text{open } \langle\alpha',x\rangle = e \text{ in } \lceil x:\tau\rceil_{\alpha\triangleright\tau'} &
\end{array}$$

$$\begin{array}{lcll}
\lfloor e:\alpha\rfloor_{\alpha\triangleright\tau'} & = & \mathrm{dn}_\alpha e & \\
\lfloor e:\alpha'\rfloor_{\alpha\triangleright\tau'} & = & e & (\alpha'\not\equiv\alpha)\\
\lfloor e:\tau_1\to\tau_2\rfloor_{\alpha\triangleright\tau'} & = & \lambda x{:}\tau_1[\alpha{:=}\tau'].\lfloor e(\lceil x:\tau_1\rceil_{\alpha\triangleright\tau'}):\tau_2\rfloor_{\alpha\triangleright\tau'} & \\
\lfloor e:\forall\alpha'.\tau\rfloor_{\alpha\triangleright\tau'} & = & \Lambda\alpha'.\lfloor e\ \alpha':\tau\rfloor_{\alpha\triangleright\tau'} & \\
\lfloor e:\exists\alpha'.\tau\rfloor_{\alpha\triangleright\tau'} & = & \text{open } \langle\alpha',x\rangle = e \text{ in } \lfloor x:\tau\rfloor_{\alpha\triangleright\tau'} &
\end{array}$$

Figure 4: Transforming a-posteriori abstraction

corresponding coercions. In the erased form an expression generated by the abstraction transformation could thus be collapsed back into its original shape (by a specialised process of partial evaluation). Or, if erasure is performed directly on the sugared version, then the transformation need not be applied at all. In both cases any runtime overhead for coercions is completely avoided.

5.2 Incompleteness

The possibility of a-posteriori abstraction shows that any package expression of $\lambda_\exists$ can be represented in λ_{up} — as long as its original implementation is representable. A valid question to ask now is whether λ_{up} is *complete* in the sense that every $\lambda_\exists$-term has a representation in it. It is easy to see that this is not the case, due to the modified typing of open expressions. For example, the following terms have the same type $\tau = (\exists\alpha.\alpha\times \mathit{int})\to \mathit{int}$ in $\lambda_\exists$, but not so in λ_{up}:

$$\begin{array}{lcl} f_1 & \equiv & \lambda x : (\exists\alpha.\alpha\times \mathit{int})\ .\ \text{open } \langle\alpha,(x_1,x_2)\rangle = x \text{ in } x_2\\ f_2 & \equiv & \lambda x : (\exists\alpha.\alpha\times \mathit{int})\ .\ 3 \end{array}$$

Consequently, λ_{up} does not allow us to formulate a function that can be applied to both these terms. Put more generally, we find that the λ_{up}-calculus is less expressive with respect to λ-abstraction than $\lambda_\exists$.

The expressiveness gap becomes even more obvious when we add a fixpoint operator. With recursion, $\lambda_\exists$ enables us to perform an arbitrary, dynamic number of abstractions:

$$\begin{array}{lcl} f_3 & \equiv & \mu f{:}\tau.\lambda x : (\exists\alpha.\alpha\times \mathit{int})\ .\ \text{open } \langle\alpha,(x_1,x_2)\rangle = x \text{ in}\\ & & \quad \text{if } x_2 = 0 \text{ then } 0 \text{ else } f\ \langle\alpha'{=}\alpha,(x_1,x_2-1){:}\alpha'\times \mathit{int}\rangle \end{array}$$

Obviously, there is no way at all to express such a function in λ_{up} with recursion, because every used abstract type shows up as an existential quantifier in the result type und hence has to be statically determined.

5.3 A complete fragment

But not all is lost. By closer inspection of both calculi we find that the typing of a term in λ_{up} will differ from the typing of a similar term in $\lambda_\exists$ only by the potential presence of additional existential quantifiers, introduced by the modified open rule. As a consequence, we can specify inductive translation rules that are able to translate a significant subset of the language. Moreover, we can isolate an interesting enough sublanguage of $\lambda_\exists$ that is fully representable in λ_{up}.

Figure 5 defines the respective translation. It recursively inserts appropriate open expressions whenever it is necessary to shift additional existential quantifiers generated by inner λ_{up}-open terms. Packages are translated using the a-posteriori abstraction transformation. Generated quantifiers are marked like $\hat\exists$ to distinguish them from other quantifiers in the rules. We write sequences of such quantifiers as $\hat\exists\vec\alpha$. If the translation succeeds, it does so with a well-typed λ_{up}-term, and it succeeds only for well-typed $\lambda_\exists$-terms:

Proposition (Correctness of Translation) *Let e be a $\lambda_\exists$-term. If $\Gamma \vdash e \rightsquigarrow e' : \tau'$ for some well-formed λ_{up}-environment Γ and λ_{up}-type τ', then*

1. $\overline{\Gamma} \vdash_{\lambda_\exists} e : \overline{\tau'}$
2. $\Gamma \vdash_{\lambda_{\mathrm{up}}} e' : \tau'$

Here, $\overline{\tau}$ is obtained from τ by erasing all generated quantifiers $\hat\exists\alpha$ and $\overline{\Gamma}$ is obtained from Γ by simplifying all type assertions $\alpha\triangleright\alpha$ to plainly α and replacing assumptions $x:\tau$ by $x:\overline{\tau}$.

There are two points where the translation may fail: at function applications, if the type of the argument's translation contains inner quantifiers at negative positions (as is the case with the examples from the previous section), or at package formation, when the implementation type contains negative quantifiers. Based on this observation, we can define a fragment of $\lambda_\exists$ where both these cases cannot occur, by stratification of the language into *small* types and terms, that may not introduce existentials, and *large* ones, where use of existentials is allowed, but which cannot be used as function arguments.

More precisely, define the syntax of small and large types as follows:

$$\begin{array}{lcl} \tau^S & ::= & \alpha \mid \tau_1^S \to \tau_2^S \mid \forall\alpha.\tau^S\\ \tau^L & ::= & \alpha \mid \tau_1^S \to \tau_1^L \mid \forall\alpha.\tau^L \mid \exists\alpha.\tau^L \end{array}$$

Small and large terms are defined correspondingly:

$$\begin{array}{lcl} e^S & ::= & x^S \mid \lambda x^S{:}\tau^S.e^S \mid e_1^S\ e_2^S \mid \Lambda\alpha.e^S \mid e^S\ \tau^S\\ e^L & ::= & x^p \mid \lambda x^S{:}\tau^S.e^L \mid e_1^L\ e_2^S \mid \Lambda\alpha.e^L \mid e^L\ \tau^S \mid\\ & & \quad \langle\alpha{=}\tau_1^S, e^L{:}\tau_2^L\rangle \mid \text{open } \langle\alpha,x^p\rangle = e_1^L \text{ in } e_2^L \end{array}$$

The metavariable p stands for either S or L. As an additional restriction, we need a side condition to rule (Open) enforcing that size is respected (we define $S<L$, $L\not<S$):

$$\frac{\Gamma\vdash e_1 : \exists\alpha.\tau'^{p_1} \qquad \Gamma,\alpha,x^{p_2}{:}\tau'^{p_1}\vdash e_2:\tau}{\Gamma\vdash(\text{open } \langle\alpha,x^{p_2}\rangle = e_1 \text{ in } e_2):\tau}\ \begin{pmatrix}\alpha\notin \mathrm{FV}(\tau)\\ p_1\le p_2\end{pmatrix}$$

Note that the language of large terms is restricted to first order in the sense that the domain of the function space is small. Similarly, type variables can only be instantiated with small types. The small types are contained in the large, likewise for terms. We say an environment Γ is small (large) if it only contains small (large) types.

The restriction to small arguments precludes any of the examples from the previous section: the generativity of a function's body cannot depend on its argument. Thus the number of abstract types generated during reduction is statically determined for all terms. As a consequence, we can state the following result about the translation with respect to the $\lambda_\exists$-fragment of large terms:

Proposition (Completeness of Large Translation) *Let e be a $\lambda_\exists$-term with $\Gamma \vdash_{\lambda_\exists} e : \tau$. If e and Γ are large, then $\Gamma' \vdash e \rightsquigarrow e' : \tau'$ for some Γ', τ', e' such that $\Gamma = \overline{\Gamma'}$ and $\tau = \overline{\tau'}$.*

In the large sublanguage of $\lambda_\exists$, existentials, and thus abstract types, are degraded to second-class citizens. The stratification between small and large is very similar to the design of the ML language: small types and terms correspond to the core language, while large types and terms mirror signatures and modules. Like the fragment, ML does not support first-class abstract types. Because universal polymorphism still is first-class in the fragment, it should be possible to encode the module and type abstraction machinery of Standard ML (which is first-order), along the lines of Russo [30], provided a generalisation to higher-order types.

Developing a theory that can cope with higher-order modules or even first-class use of existential packages is an interesting direction for future research.

6. RELATED WORK

Although being simple in spirit, to our knowledge there is no previous work that isolates the aspect of type generativity for abstraction and formalises it in a calculus. While module theories usually account for generativity as well, they do so solely on the static level of typing rules. In fact, all of the most influential theories for ML modules [15, 18, 29, 6] are not full calculi, but merely type systems, that side-step the issue of reduction. The presence of ad-hoc typing rules encompassing type abstraction precludes a type-preserving evaluation strategy — for similar reasons as the problematic suggestion by Abadi et.al. discussed in section 3.1.

One notable exception is Sewell, who uses generativity for modelling certain aspects of type abstraction [32]. However, in his system generated abstract types are recorded as manifestly equal to their representation in the global environment, so that opacity is not properly maintained dynamically. Moreoever, his system requires a lot of additional type annotations on terms.

Glew presented a calculus for generating new tagged types at runtime and dispatching on them [10]. His system is more complex than ours in order to allow hierarchical types being generated, but it is not fully reflexive since untagged types cannot be analysed.

The work most relevant to ours is by Grossman et al. on proof techniques for abstraction [11]. They present a calculus that uses annotated brackets as special syntax for marking abstraction boundaries during reduction. These are somewhat similar to coercions in λ_{up}, but are not directed and not restricted to individual values of abstract type. Moreoever, nested brackets may be collapsed, although that does not seem to be essential to their system. However, their calculus requires identifying a fixed set of abstractions statically, since technically, the reduction system has to be extended for each occuring abstraction. Furthermore, the definition of type equivalence is complicated by its dependence on an additional type environment. This also complicates the operational semantics, because the environment has to be maintained dynamically to cope with abstraction scoping. Their calculus hence seems less suited as a simple operational model for type abstraction.

$$\frac{\Gamma(x) = \tau'}{\Gamma \vdash x \rightsquigarrow x : \tau'}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \rightsquigarrow e' : \tau_2'}{\Gamma \vdash \lambda x{:}\tau_1.e \rightsquigarrow \lambda x{:}\tau_1.e' : \tau_1 \to \tau_2'}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \hat\exists\vec\alpha_1.\tau_2 \to \tau_1' \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \hat\exists\vec\alpha_2.\tau_2}{\begin{array}{l}\Gamma \vdash e_1\ e_2 \rightsquigarrow (\text{open } \langle\vec\alpha_1, x_1\rangle = e_1' \text{ in} \\ \qquad \text{open } \langle\vec\alpha_2, x_2\rangle = e_2' \text{ in } x_1\ x_2) : \hat\exists\vec\alpha_1.\hat\exists\vec\alpha_2.\tau_1'\end{array}}$$

$$\frac{\Gamma, \alpha \triangleright \alpha \vdash e \rightsquigarrow e' : \tau'}{\Gamma \vdash \Lambda\alpha.e \rightsquigarrow \Lambda\alpha.e' : \forall\alpha.\tau'}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \hat\exists\vec\alpha.\forall\alpha.\tau_2' \qquad \Gamma \vdash \tau_1 : \diamond}{\Gamma \vdash e\ \tau_1 \rightsquigarrow (\text{open } \langle\vec\alpha, x\rangle = e' \text{ in } x\ \tau_1) : \hat\exists\vec\alpha.\tau_2'[\alpha{:=}\tau_1]}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau_2'[\alpha{:=}\tau_1] \qquad \Gamma \vdash \tau_1 : \diamond}{\Gamma \vdash \langle\alpha{=}\tau_1, e{:}\overline{\tau_2'}\rangle \rightsquigarrow \langle\alpha{=}\tau_1, \lceil e' : \tau_2'\rceil_{\alpha\triangleright\tau_1}\rangle : \exists\alpha.\tau_2'}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \hat\exists\vec\alpha.\exists\alpha.\tau_1' \qquad \Gamma, \alpha \triangleright \alpha, x{:}\tau_1' \vdash e_2 \rightsquigarrow e_2' : \tau_2'}{\begin{array}{l}\Gamma \vdash (\text{open } \langle\vec\alpha, x\rangle = e_1 \text{ in } e_2) \rightsquigarrow \\ \quad (\text{open } \langle\vec\alpha, x'\rangle = e_1' \text{ in open } \langle\alpha, x\rangle = x' \text{ in } e_2') : \hat\exists\vec\alpha.\hat\exists\alpha.\tau_2'\end{array}}$$

where for $\vec\alpha = \alpha_1, \ldots, \alpha_n$:

$$\begin{array}{rcl}\text{open } \langle\vec\alpha, x\rangle = e_1 \text{ in } e_2 & :\equiv & \text{open } \langle\alpha_1, x_1\rangle = e_1 \text{ in} \\ & & \text{open } \langle\alpha_2, x_2\rangle = x_1 \text{ in} \\ & & \ldots \\ & & \text{open } \langle\alpha_n, x\rangle = x_{n-1} \text{ in } e_2\end{array}$$

Figure 5: Translation of $\lambda_\exists$ into λ_{up}

7. CONCLUSION

The standard encoding of abstract types via existential types relies on parametricity of polymorphism. If parametricity is not given, due to constructs for type analysis, the encoding is inappropriate because it cannot warrant encapsulation. We might go as far as saying that the similarities between type abstraction in the strong sense of encapsulation on one hand, and existential abstraction on the other are purely superficial. In non-parametric settings it is necessary to capture generativity to achieve dynamic opacity and thereby encapsulation.

We presented a calculus for abstract types with syntactic treatment of dynamic generativity. It uses a variation of

standard existentials that builds on coercions and seems expressive enough to encode SML's first-order modules. One obstacle for adoption of dynamic opacity along the lines of λ_{up} into a module theory is the calculus' reliance on explicit existential elimination. Module languages and calculi usually do not require this, but keep existential elimination implicit [18]. For a module calculus it is desirable to have projection built in instead of going through a translation to insert the necessary scope extrusion constructs. It is compelling to automize extrusion and use a different approach to model generativity by adopting an expression form like the ν-binder present in π-calculus [31] or the ν-calculus [26]. While these bind term names, though, we needed to bind type names. Such an "uppercase ν" raises interesting issues about typing and reduction.

Developing proof techniques to show representation independence or extensionality [22, 24] of λ_{up}-existentials looks like an interesting challenge. There appears to be very little work on proof techniques for operational equivalence in non-parametric extensions of the λ-calculus. It is not obvious how techniques like logical relations [22, 27, 25] can be applied in such a setting.

Finally, it should be noted that, despite being insufficient for modelling encapsulation, standard existential types are still useful in a language with type analysis. In particular, they provide a straight-forward formulation of dynamics. Both kinds of existentials are thus complementary.

8. REFERENCES

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[2] E.S. Bainbridge, Peter Freyd, Andre Scedrov, and Philip Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1989. Corrigendum in 71(3):431, 1990.

[3] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[4] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. (Modula-3) language definition. In Greg Nelson, editor, *System Programming with Modula-3*, chapter 2, pages 11–66. Prentice Hall, 1991.

[5] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, March 1990.

[6] Karl Crary, Robert Harper, and Derek Dreyer. A type system for higher-order modules. In *29th Symposium on Principles of Programming Languages*, Portland, USA, January 2002.

[7] Karl Crary, Stephanie Weirich, and Greg Morisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, Baltimore, USA, September 1998.

[8] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages*, San Francisco, USA, January 1995.

[9] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, June 1972.

[10] Neal Glew. Type dispatch for named hierarchical types. In *International Conference on Functional Programming*, Paris, France, October 1999.

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

[12] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, USA, January 1995.

[13] Ada reference manual. Technical Report ISO/IEC 8652:1995(E), International Organization for Standardization, December 1994.

[14] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th Symposium on Principles of Programming Languages*, pages 177–188, New York, USA, January 1992. ACM Press.

[15] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium on Principles of Programming Languages*, pages 109–122, Portland, USA, January 1994. ACM.

[16] Xavier Leroy. *The Objective Caml System*. INRIA, `http://pauillac.inria.fr/ocaml/htmlman/`, July 2001.

[17] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[18] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, May 1997.

[19] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Robert Scheifler, and Alan Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, 1979.

[20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *Definition of Standard ML (Revised)*. The MIT Press, 1997.

[21] Yasuhiko Minamide and Jacque Garrigue. In the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming*, Baltimore, USA, September 1998.

[22] John Mitchell. On the equivalence of data representations. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[23] John Mitchell and Gordon Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *12th Symposium on Principles of Programming Languages*, 1985.

[24] Andrew Pitts. Existential types: Logical relations and operational equivalence. In *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.

[25] Andrew Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[26] Andrew Pitts and Ian Stark. On the observable properties of higher order functions that dynamically create local names. In Paul Hudak, editor, *Workshop on State in Programming Languages*, pages 31–45, Copenhagen, Denmark, 1993.

[27] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Beeze and Jan Friso Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, Berlin, 1993.

[28] John Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing*, pages 513–523, Amsterdam, 1983. North Holland.

[29] Claudio Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[30] Claudio Russo. Non-dependent types for Standard ML modules. In *International Conference on Principles and Practice of Declarative Programming*, Paris, France, September 1999.

[31] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, December 2001.

[32] Peter Sewell. Modules, abstract types, and distributed versioning. In *28th Symposium on Principles of Programming Languages*, London, UK, January 2001.

[33] Christopher Strachey. Fundamental concepts in programming languages. In *Lecture Notes, International Summer School in Computer Programming*. Copenhagen, August 1967. Reprinted in: *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

[34] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth International Conference on Functional Programming*, pages 82–93, Montreal, Canada, September 2000.

[35] Stephanie Weirich. Type-safe cast. In *International Conference on Functional Programming*, pages 58–67, Montreal, Canada, September 2000.

[36] Niklaus Wirth. *Programming in MODULA-2*. Springer-Verlag, 3rd edition, 1985.

APPENDIX

A. PROOFS

A.1 Typing

We omit well-formedness of types. Adding corresponding proofs is straight-forward. The following basic lemmata are easy to show along the lines of [3]:

LEMMA 1 (WEAKENING). *If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$ *is another basis, then* $\Gamma' \vdash e : \tau$.

PROOF. By induction on the derivation of $e : \tau$. □

LEMMA 2 (INVERSION).

1. *If* $\Gamma \vdash x : \tau$, *then* $x{:}\tau \in \Gamma(x)$.
2. *If* $\Gamma \vdash \mathrm{up}_\alpha\ e : \tau$, *then* $\tau \equiv \alpha$ *and there is a type* τ' *such that* $\Gamma \vdash e : \tau'$ *and* $\alpha \triangleright \tau' \in \Gamma$.
3. *If* $\Gamma \vdash \mathrm{dn}_\alpha : \tau$, *then* $\Gamma \vdash e : \alpha$ *and* $\alpha \triangleright \tau \in \Gamma$.
4. *If* $\Gamma \vdash (\lambda x{:}\tau'.e) : \tau$, *then there is a type* τ'' *such that* $\tau \equiv \tau' \to \tau''$ *and* $\Gamma, x{:}\tau' \vdash e : \tau''$.
5. *If* $\Gamma \vdash e_1\ e_2 : \tau$, *then there is a type* τ' *such that* $\Gamma \vdash e_1 : \tau' \to \tau$ *and* $\Gamma \vdash e_2 : \tau'$.
6. *If* $\Gamma \vdash \Lambda\alpha.e : \tau$, *then there is a type* τ' *such that* $\tau \equiv \forall\alpha.\tau'$ *and* $\Gamma, \alpha \triangleright \alpha \vdash e : \tau'$.
7. *If* $\Gamma \vdash e\ \tau' : \tau$, *then there is a type* τ'' *such that* $\Gamma \vdash e : \forall\alpha.\tau''$ *and* $\tau \equiv \tau''[\alpha := \tau']$.
8. *If* $\Gamma \vdash \langle \alpha = \tau', e\rangle : \tau$, *then there is a type* τ'' *such that* $\tau \equiv \exists\alpha.\tau''$ *and* $\Gamma, \alpha \triangleright \tau' \vdash e : \tau''$.
9. *If* $\Gamma \vdash (\text{open } \langle \alpha, x\rangle = e_1 \text{ in } e_2) : \tau$, *then there are types* τ', τ'' *such that* $\tau \equiv \exists\alpha.\tau''$ *and* $\Gamma \vdash e_1 : \exists\alpha.\tau'$ *and* $\Gamma, \alpha \triangleright \alpha, x{:}\tau' \vdash e_2 : \tau''$.

PROOF. By induction on the length of the corresponding derivation. □

LEMMA 3 (SUBSTITUTION). *If* $\Gamma, x{:}\tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$, *then* $\Gamma \vdash e[e'/x] : \tau$.

PROOF. By induction on the generation of $\Gamma, x{:}\tau' \vdash e : \tau$. □

LEMMA 4 (SPECIALISATION).

1. *If* $\Gamma, \alpha \triangleright \alpha \vdash \tau : \diamond$ *and* $\Gamma \vdash \tau' : \diamond$, *then* $\Gamma, \alpha \triangleright \tau' \vdash \tau : \diamond$.
2. *If* $\Gamma, \alpha \triangleright \alpha \vdash e : \tau$ *and* $\Gamma \vdash \tau' : \diamond$, *then* $\Gamma, \alpha \triangleright \tau' \vdash e : \tau$.

PROOF.

1. By induction on the generation of $\Gamma, \alpha \triangleright \alpha \vdash \tau : \diamond$.
2. By induction on the generation of $\Gamma, \alpha \triangleright \alpha \vdash e : \tau$, using 1. The side conditions of the typing rules (UP) and (DN) ensure that e does not contain any occurences of up_α or dn_α.

□

PROPOSITION 5 (PRESERVATION). *If* $\Gamma \vdash e : \tau$ *and* $e \to^* e'$, *then* $\Gamma \vdash e' : \tau$.

PROOF. By induction on the generation of $\to^*$ using lemmas 1–3. We treat the cases of basic one-step reduction not occuring in plain λ-calculus:

- Consider $e \equiv (\text{open } \langle \alpha, x\rangle = \langle \alpha = \tau', e_1\rangle \text{ in } e_2)$ and $e' \equiv \langle \alpha = \tau', e_2[e_1/x]\rangle$. From the inversion lemma it follows that $\tau \equiv \exists\alpha.\tau_2$ and $\Gamma \vdash \langle \alpha = \tau', e_1\rangle : \exists\alpha.\tau_1$ and $\Gamma, \alpha \triangleright \alpha, x{:}\tau_1 \vdash e_2 : \tau_2$ (*) for some τ_1, τ_2. From a second application of the inversion lemma it follows that $\Gamma, \alpha \triangleright \tau' \vdash e_1 : \tau_1$. By the specialisation lemma judgement (*) can be specialised to $\Gamma, \alpha \triangleright \tau', x{:}\tau_1 \vdash e_2 : \tau_2$. Therefore by the substitution lemma $\Gamma, \alpha \triangleright \tau' \vdash e_2[e_1/x] : \tau_2$. With rule (EX) we can derive $\Gamma \vdash \langle \alpha = \tau', e_2[e_1/x]\rangle : \tau$.
- Consider $e \equiv \mathrm{dn}_\alpha(\mathrm{up}_\alpha e_1)$ and $e' \equiv e_1$. From the inversion lemma it follows that $\alpha \triangleright \tau \in \Gamma$ and $\Gamma \vdash \mathrm{up}_\alpha\ e_1 : \alpha$. Repeated application of the same lemma then yields $\Gamma \vdash e_1 : \tau$.

□

PROPOSITION 6 (UNIQUE TYPES). *Whenever* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau \equiv \tau'$.

PROOF. By induction on the derivation. □

A.2 Progress

The following lemma describes the shape of λ_{up}-values at particular types:

LEMMA 7 (CANONICAL FORMS). *For any value* v:

1. *If* $\Gamma \vdash v : \alpha$ *then* $v \equiv \mathrm{up}_\alpha\ v'$ *(and* $\alpha \triangleright \tau \in \Gamma$ *with* $\tau \not\equiv \alpha$*).*
2. *If* $\Gamma \vdash v : \tau_1 \to \tau_2$ *then* $v \equiv \lambda x{:}\tau_1.e$.
3. *If* $\Gamma \vdash v : \forall\alpha.\tau$ *then* $v \equiv \Lambda\alpha.e$.
4. *If* $\Gamma \vdash v : \exists\alpha.\tau$ *then* $v \equiv \langle \alpha = \tau', v'\rangle$.

PROOF. By inspection of the cases for v. □

Package values are binders for type variables. Hence, in order to prove progress by induction, a slightly stronger induction hypothesis is necessary:

PROPOSITION 8 (PROGRESS). *Let* Γ *be an environment containing only type assertions (i.e. it is of the form* $\{\alpha_1 \triangleright \tau_1, \cdots, \alpha_n \triangleright \tau_n\}$*). Then for any expression* e *such that* $\Gamma \vdash e : \tau$, *either* $e \equiv v$ *for some value* v, *or* $e \to e'$ *for some expression* e'.

PROOF. By easy induction on the typing derivations. We treat the most interesting cases:

- Consider $e \equiv \langle \alpha = \tau, e' \rangle$: If e' is a value the statement follows immediately. Otherwise by induction using $\Gamma' = \Gamma, \alpha \triangleright \tau$ there is an applicable reduction rule $e' \to e''$ and thus $e \to \langle \alpha = \tau, e'' \rangle$.
- Consider $e \equiv (\mathrm{dn}_\alpha\ e')$: Inversion implies $\Gamma \vdash e' : \alpha$. If e' is a value then from the canonical forms lemma it follows that $e' \equiv (\mathrm{up}_\alpha\ e'')$ and the coercions can be cancelled, i.e. $e \to e''$. Otherwise there is a reduction $e' \to e''$ by induction and thus $e \to \mathrm{dn}_\alpha\ e''$.

□

COROLLARY 9 (PROGRESS FOR CLOSED EXPRESSIONS). *If $\emptyset \vdash e : \tau$, then either $e \equiv v$ for some value v, or there is an expression e' such that $e \to e'$.*

A.3 Termination

We prove termination (strong normalization) for λ_{up} by simulating reduction in the second-order lambda calculus $\lambda 2$, which is terminating [3]. We use the translation shown in figure 6, which is a variation of the standard encoding of existential types into polymorphic lambda calculus. Note that we need to insert additional type annotations, thus translation of expressions has to be defined on type derivations instead of plain terms. We assume there are distinct and unique identifiers up_α and dn_α for every type variable α. Occurences of β and f are meant to denote fresh variables. Moreover, we take the liberty to abbreviate some environments in the derivations.

If type translation is extended to environments as follows:

$$[\![\Gamma]\!] \quad = \quad \{x : [\![\tau]\!] \mid x : \tau \in \Gamma\} \cup \{\alpha \mid \alpha \triangleright \tau \in \Gamma\}$$

then the following lemma holds:

LEMMA 10 (WELL-DEFINEDNESS OF TRANSLATION). *If $\Gamma \vdash_{\lambda_{\mathrm{up}}} e : \tau$, then $[\![\Gamma]\!] \vdash_{\lambda 2} [\![\Gamma \vdash e : \tau]\!] : [\![\tau]\!]$.*

I.e. type and term translation are consistent and translation yields well-typed $\lambda 2$ terms.

PROOF. By induction on the derivation $\Gamma \vdash e : \tau$. □

The translation is consistent w.r.t. reduction and yields redexes as translation of redexes. It is thus suitable as a basis for simulation:

LEMMA 11 (SIMULATION). *Let e_1, e_2 be λ_{up} terms with $\Gamma \vdash_{\lambda_{\mathrm{up}}} e_1 : \tau$ and $e_1 \to_{\lambda_{\mathrm{up}}} e_2$ (by preservation this implies $\Gamma \vdash_{\lambda_{\mathrm{up}}} e_2 : \tau$).*

1. *$[\![\Gamma \vdash e_1 : \tau]\!] \not\equiv [\![\Gamma \vdash e_2 : \tau]\!]$.*
2. *There exists a reduction sequence $[\![\Gamma \vdash e_1 : \tau]\!] \to^*_{\lambda\exists} [\![\Gamma \vdash e_2 : \tau]\!]$.*

PROOF. Simultanously by inspection of the cases for $\to_{\lambda_{\mathrm{up}}}$. □

PROPOSITION 12 (TERMINATION). *There are no infinite reduction sequences for any given λ_{up} term.*

PROOF. By lemma 10 every λ_{up} term e can be translated into a well-formed $\lambda 2$ term e'. By lemma 11 any reduction of e can be simulated in e' and the simulation gives an upper bound to the number of reduction steps for e. There are no infinite reduction sequences in $\lambda 2$, hence all reductions in λ_{up} are finite as well. □

A.4 Confluence

The reduction rules for λ_{up} are mostly orthogonal to plain λ and do not perform substitution on type variables. Confluence can hence be proved by extending in a straight-forward way the proof by Barendregt [3] for the (untyped) λ-calculus (which also works for the typed λ-calculus).

PROPOSITION 13 (CONFLUENCE). *Whenever there are expressions e, e_1, e_2 such that $e \to^* e_1$ and $e \to^* e_2$, then their exists an expression e' with $e_1 \to^* e'$ and $e_2 \to^* e'$.*

PROOF. Extend the definition of underlined terms in [3] to open expressions and cancelling coercions. After adapting the definition of underlined reduction and underline contraction appropriately, the remainder of the proof of the Church-Rosser property (and thus confluence) goes through with only minor adjustments. We refer to [3] for details of the proof. □

A.5 A-posteriori abstraction

The abstraction transformation (figure 4) is well-formed:

LEMMA 14 (A-POSTERIORI ABSTRACTION INVARIANTS). *Let e be a λ_{up}-expression and $\Gamma \equiv \Gamma', \alpha \triangleright \tau'$ an evironment with $\tau' \not\equiv \alpha$.*

1. *If $\Gamma \vdash e : \tau[\alpha := \tau']$, then $\Gamma \vdash \lceil e : \tau \rceil_{\alpha \triangleright \tau'} : \tau$.*
2. *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \lfloor e : \tau \rfloor_{\alpha \triangleright \tau'} : \tau[\alpha := \tau']$.*

PROOF. By simultanous induction on the structure of τ. We only show two of the more interesting cases:

- Consider $\tau \equiv \alpha$:
  1. The premise simplifies to $\Gamma \vdash e : \tau'$. Using rule (UP) we can derive $\Gamma \vdash \mathrm{up}_\alpha\ e : \alpha$, which asserts the conjecture.
  2. Likewise.
- Consider $\tau \equiv \tau_1 \to \tau_2$:
  1. The premise becomes $\Gamma \vdash e : \tau_1[\alpha := \tau'] \to \tau_2[\alpha := \tau']$. Lemma 1 allows weakening to $\Gamma, x : \tau_1 \vdash e : \tau_1[\alpha := \tau'] \to \tau_2[\alpha := \tau']$ (*) for some $x \notin \mathrm{Dom}(\Gamma)$. For this x we trivially have $\Gamma, x : \tau_1 \vdash x : \tau_1$, which gives $\Gamma, x : \tau_1 \vdash \lfloor x : \tau_1 \rfloor_{\alpha \triangleright \tau'} : \tau_1[\alpha := \tau']$ by induction. Together with judgement (*) rule (APP) enables us to derive $\Gamma, x : \tau_1 \vdash e(\lfloor x : \tau_1 \rfloor_{\alpha \triangleright \tau'}) : \tau_2[\alpha := \tau']$. Again by induction it follows that $\Gamma, x : \tau_1 \vdash \lceil e(\lfloor x : \tau_1 \rfloor_{\alpha \triangleright \tau'}) : \tau_2 \rceil_{\alpha \triangleright \tau'} : \tau_2$. A final derivation using rule (ABS) yields $\Gamma \vdash (\lambda x :$

$$
\begin{array}{rcl}
[\![\alpha]\!] & \equiv & \alpha \\
[\![\tau_1 \to \tau_2]\!] & \equiv & [\![\tau_1]\!] \to [\![\tau_2]\!] \\
[\![\forall\alpha.\tau]\!] & \equiv & \forall\alpha.[\![\tau]\!] \\
[\![\exists\alpha.\tau]\!] & \equiv & \forall\beta.(\mathrm{E}^\alpha_\beta[\![\tau]\!] \to \beta) \\
\left[\!\!\left[\dfrac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\right]\!\!\right] & \equiv & x \\
\left[\!\!\left[\dfrac{\Gamma \vdash e : \tau \qquad \alpha \rhd \tau \in \Gamma}{\Gamma \vdash \mathrm{up}_\alpha e : \alpha}\right]\!\!\right] & \equiv & up_\alpha[\![\Gamma \vdash e : \tau]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash e : \alpha \qquad \alpha \rhd \tau \in \Gamma}{\Gamma \vdash \mathrm{dn}_\alpha e : \tau}\right]\!\!\right] & \equiv & dn_\alpha[\![\Gamma \vdash e : \alpha]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash \tau_1 : \diamond \qquad \Gamma' \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1.e) : \tau_1 \to \tau_2}\right]\!\!\right] & \equiv & \lambda x{:}[\![\tau_1]\!].[\![\Gamma' \vdash e : \tau_2]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\right]\!\!\right] & \equiv & [\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\ [\![\Gamma \vdash e_2 : \tau']\!] \\
\left[\!\!\left[\dfrac{\Gamma' \vdash \tau : \diamond}{\Gamma \vdash (\Lambda\alpha.e) : \forall\alpha.\tau}\right]\!\!\right] & \equiv & \Lambda\alpha.[\![\Gamma' \vdash e : \tau]\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash e : \forall\alpha.\tau \qquad \Gamma \vdash \tau' : \diamond}{\Gamma \vdash e\ \tau' : \tau[\alpha := \tau']}\right]\!\!\right] & \equiv & [\![\Gamma \vdash e : \forall\alpha.\tau]\!]\ [\![\tau']\!] \\
\left[\!\!\left[\dfrac{\Gamma \vdash \tau' : \diamond \qquad \Gamma' \vdash e : \tau}{\Gamma \vdash \langle\alpha{=}\tau', e\rangle : \exists\alpha.\tau}\right]\!\!\right] & \equiv & \Lambda\beta.\lambda f{:}\mathrm{E}^\alpha_\beta[\![\tau]\!]. \\
 & & f\ [\![\tau']\!]\ ((\Lambda\alpha.\lambda up_\alpha{:}[\![\tau']\!]{\to}\alpha.\lambda dn_\alpha{:}\alpha{\to}[\![\tau']\!].[\![\Gamma' \vdash e : \tau]\!])\ [\![\tau']\!]\ id_{[\![\tau']\!]}\ id_{[\![\tau']\!]}) \\
\left[\!\!\left[\dfrac{\Gamma \vdash e_1 : \exists\alpha.\tau' \qquad \Gamma' \vdash e_2 : \tau}{\Gamma \vdash (\mathrm{open}\ \langle\alpha, x\rangle = e_1\ \mathrm{in}\ e_2) : \exists\alpha.\tau}\right]\!\!\right] & \equiv & \Lambda\beta.[\![\Gamma \vdash e_1 : \exists\alpha.\tau]\!]\ (\mathrm{E}^\alpha_\beta[\![\tau]\!] \to \beta)\ (\Lambda\alpha.\lambda x{:}[\![\tau']\!].\lambda f{:}\mathrm{E}^\alpha_\beta[\![\tau]\!].f\ \alpha\ [\![\Gamma' \vdash e_2 : \tau]\!]) \\
\\
\text{where} & & \mathrm{E}^\alpha_\beta(\tau) \equiv \forall\alpha.(\tau \to \beta) \\
 & & id_\tau \equiv \lambda x{:}\tau.x
\end{array}
$$

Figure 6: Translation of λ_{up} into $\lambda 2$

$\tau_1.\lceil e(\lfloor x : \tau_1 \rfloor_{\alpha\rhd\tau'}) : \tau_2 \rceil_{\alpha\rhd\tau'}) : \tau_1 \to \tau_2$, the conjecture.

2. Similarly.

□

A.6 Translation

The *generative projection* of a $\lambda_\exists$-type τ is defined as follows:

$$
\begin{array}{rcl}
\tau^\exists & = & \begin{cases} \{\tau\} & \text{if } \tau \text{ small} \\ \{\hat{\exists}\vec{\alpha}.\tau' \mid \tau' \in \tau^{[\exists]}\} & \text{otherwise} \end{cases} \\
\\
\alpha^{[\exists]} & = & \{\alpha\} \\
(\tau_1 \to \tau_2)^{[\exists]} & = & \{\tau_1 \to \tau_2' \mid \tau_2' \in \tau_2^\exists\} \\
(\forall\alpha.\tau)^{[\exists]} & = & \{\forall\alpha.\tau' \mid \tau' \in \tau^\exists\} \\
(\exists\alpha.\tau)^{[\exists]} & = & \{\exists\alpha.\tau' \mid \tau' \in \tau^\exists\}
\end{array}
$$

The inverse *erasure* of generated λ_{up}-types is:

$$
\begin{array}{rcl}
\overline{\alpha} & \equiv & \alpha \\
\overline{\tau_1 \to \tau_2} & \equiv & \overline{\tau_1} \to \overline{\tau_2} \\
\overline{\forall\alpha.\tau} & \equiv & \forall\alpha.\overline{\tau} \\
\overline{\exists\alpha.\tau} & \equiv & \exists\alpha.\overline{\tau} \\
\overline{\hat{\exists}\alpha.\tau} & \equiv & \overline{\tau}
\end{array}
$$

Erasure can be generalised to environments, yielding $\lambda_\exists$-environments:

$$
\begin{array}{rcl}
\overline{\Gamma, \alpha \rhd \tau} & \equiv & \overline{\Gamma}, \alpha \\
\overline{\Gamma, x{:}\tau} & \equiv & \overline{\Gamma}, x{:}\overline{\tau}
\end{array}
$$

Correctness of the translation is not difficult to show:

PROPOSITION 15 (CORRECTNESS OF TRANSLATION). *Let e be a $\lambda_\exists$-term. If $\Gamma \vdash e \rightsquigarrow e' : \tau'$ for some well-formed λ_{up}-environment Γ and λ_{up}-type τ', then*

1. $\overline{\Gamma} \vdash_{\lambda_\exists} e : \overline{\tau'}$
2. $\Gamma \vdash_{\lambda_{\mathrm{up}}} e' : \tau'$

PROOF.

1. By easy induction on the derivation.
2. By induction on the derivation, using the a-posteriori abstraction lemma 14 in the case of packages.

□

Smallness of types is invariant under substitution:

LEMMA 16 (SMALL TYPE SUBSTITUTION). *If τ_1 is a small (large) type and τ_2 is a small type, then $\tau_1[\alpha := \tau_2]$ is a small (large) type.*

PROOF. By trivial induction on the structure of τ_1. □

Small terms are typed by small types in $\lambda_\exists$:

LEMMA 17 (SMALL AND LARGE TYPING). *If $\Gamma \vdash_{\lambda_\exists} e : \tau$, and e and $\Gamma|_{\mathrm{FV}(e)}$ are small (large), then τ is a small (large) type.*

PROOF. By induction on the derivation, using lemma 16 in cases (INST) and (SEAL). Note the use of a modified (OPEN) rule with an additional side condition on the size of the term variable (see section 5.3). □

Translation of small terms is the identity relation:

LEMMA 18 (SMALL TERM TRANSLATION). *If $\Gamma = \overline{\Gamma'}$ for some λ_{up}-environment, and $\Gamma \vdash_{\lambda_\exists} e : \tau$, and e, τ and $\Gamma|_{\mathrm{FV}(e)}$ are small, then $\Gamma' \vdash e \rightsquigarrow e : \tau$.*

PROOF. By easy induction on the derivation of $\Gamma \vdash_{\lambda_\exists} e : \tau$, using lemma 17. □

To prove completeness of the translation for large terms we require the following simple lemmas on generated types:

LEMMA 19 (TYPE GENERATION).

1. *For all τ, $\tau \in \tau^\exists$.*
2. *If $\tau' \in \tau^\exists$ then $\overline{\tau'} \equiv \tau$.*
3. *If τ_1 is a small type and $\tau_3 \in (\tau_2[\alpha := \tau_1])^\exists$ for some τ_2, then $\tau_3 \equiv \tau_2'[\alpha := \tau_1]$ such that $\tau_2' \in \tau_2^\exists$.*

PROOF. Each by easy induction on the definition of $\tau^\exists$. □

Translation is complete with respect to the $\lambda_\exists$-fragment of large terms:

PROPOSITION 20 (COMPLETENESS OF LARGE TRANSLATION). *If $\Gamma = \overline{\Gamma'}$ for some well-formed λ_{up}-environment, and $\Gamma \vdash_{\lambda_\exists} e : \tau$, and e and Γ are large, then $\Gamma' \vdash e \rightsquigarrow e' : \tau'$ such that $\tau' \in \tau^\exists$.*

PROOF. By induction on the derivation of $\Gamma \vdash_{\lambda_\exists} e : \tau$. Note that by lemma 17, τ is a large type. □

- case $\Gamma \vdash_{\lambda_\exists} x : \tau$
  1. by inversion, $\Gamma(x) = \tau$
  2. by lemma 19.1, $\tau \in \tau^\exists$
- case $\Gamma \vdash_{\lambda_\exists} \lambda x{:}\tau_1.e_2 : \tau_1 \to \tau_2$
  1. by inversion, $\Gamma, x{:}\tau_1 \vdash_{\lambda_\exists} e_2 : \tau_2$
  2. by induction, $\Gamma, x : \tau_1 \vdash e_2 \rightsquigarrow e_2 : \tau_2'$ such that $\tau_2' \in \tau_2^\exists$
  3. $(\tau_1 \to \tau_2') \in (\tau_1 \to \tau_2)^\exists$
- case $\Gamma \vdash_{\lambda_\exists} e_1\ e_2 : \tau_1$
  1. by inversion, $\Gamma \vdash_{\lambda_\exists} e_1 : \tau_2 \to \tau_1$ and $\Gamma \vdash_{\lambda_\exists} e_2 : \tau_2$
  2. by induction, $\Gamma \vdash e_1 \rightsquigarrow e_1 : \tau_3'$ such that $\tau_3' \in (\tau_2 \to \tau_1)^\exists$
  3. by definition of $\bullet^\exists$, $\tau_3' \equiv \hat{\exists}\vec{\alpha}.\tau_2 \to \tau_1'$ such that $\tau_1' \in \tau_1^\exists$
  4. by definition of large terms, e_2 is small
  5. by lemma 18, $\Gamma \vdash e_2 \rightsquigarrow e_2 : \tau_2$
  6. $\hat{\exists}\vec{\alpha}.\tau_1' \in \tau_1^\exists$
- case $\Gamma \vdash_{\lambda_\exists} \Lambda\alpha.e_1 : \forall\alpha.\tau_1$
  1. by inversion, $\Gamma, \alpha \vdash_{\lambda_\exists} e_1 : \tau_1$
  2. $\Gamma, \alpha = \overline{\Gamma, \alpha \triangleright \alpha}$
  3. by induction, $\Gamma, \alpha \triangleright \alpha \vdash e_1 \rightsquigarrow e_1 : \tau_1'$ such that $\tau_1' \in \tau_1^\exists$
  4. $\forall\alpha.\tau_1' \in (\forall\alpha.\tau_1)^\exists$
- case $\Gamma \vdash_{\lambda_\exists} e_1\ \tau_2 : \tau_1[\alpha := \tau_2]$
  1. by inversion, $\Gamma \vdash_{\lambda_\exists} e_1 : \forall\alpha.\tau_1$
  2. by induction, $\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_3'$ such that $\tau_3' \in (\forall\alpha.\tau_1)^\exists$
  3. by definition of $\bullet^\exists$, $\tau_3' \equiv \hat{\exists}\vec{\alpha}.\forall\alpha.\tau_1'$ such that $\tau_1' \in \tau_1^\exists$
  4. by definition of large terms, τ_2 is a small type
  5. by lemma 19.3, $\hat{\exists}\vec{\alpha}.\tau_1[\alpha := \tau_2] \in (\tau_1[\alpha := \tau_2])^\exists$
- case $\Gamma \vdash_{\lambda_\exists} \langle \alpha{=}\tau_1, e_2{:}\tau_2 \rangle : \exists\alpha.\tau_2$
  1. by inversion, $\Gamma \vdash_{\lambda_\exists} e_2 : \tau_2[\alpha := \tau_1]$
  2. by induction, $\Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_3'$ such that $\tau_3' \in (\tau_2[\alpha := \tau_1])^\exists$
  3. by definition of large terms, τ_1 is a small type
  4. by lemma 19.3, $\tau_3' \equiv \tau_2'[\alpha := \tau_1]$ such that $\tau_2' \in \tau_2^\exists$
  5. by lemma 19.2, $\overline{\tau_2'} \equiv \tau_2$
  6. $\exists\alpha.\tau_2' \in (\exists\alpha.\tau_2)^\exists$
- case $\Gamma \vdash_{\lambda_\exists} \mathrm{open}\ \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \tau_2$
  1. by inversion, $\Gamma \vdash_{\lambda_\exists} e_1 : \exists\alpha.\tau_1$ and $\Gamma, \alpha, x : \tau_1 \vdash_{\lambda_\exists} e_2 : \tau_2$
  2. by induction, $\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_3'$ such that $\tau_3' \in (\exists\alpha.\tau_1)^\exists$
  3. by definition of $\bullet^\exists$, $\tau_3' \equiv \hat{\exists}\vec{\alpha}.\exists\alpha.\tau_1'$ such that $\tau_1' \in \tau_1^\exists$
  4. by lemma 19.2, $\tau_1 \equiv \overline{\tau_1'}$
  5. $\Gamma, \alpha, x{:}\tau_1 = \overline{\Gamma, \alpha \triangleright \alpha, x{:}\tau_1'}$
  6. by induction, $\Gamma, \alpha \triangleright \alpha, x : \tau_1' \vdash e_2 \rightsquigarrow e_2' : \tau_2'$ such that $\tau_2' \in \tau_2^\exists$
  7. $\hat{\exists}\vec{\alpha}.\hat{\exists}\alpha.\tau_2' \in \tau_2^\exists$