Generativity and Dynamic Opacity for Abstract Types (Extended Version)

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

ABSTRACT

The standard formalism for explaining abstract types is existential quantification. While it provides a sufficient model for type abstraction in entirely statically typed languages, it proves to be too weak for languages enriched with forms of dynamic typing, where parametricity is violated. As an alternative approach to type abstraction that addresses this shortcoming we present a calculus for dynamic type generation. It features an explicit construct for generating new type names and relies on coercions for managing abstraction boundaries between generated types and their designated representation. Sealing is represented as a generalized form of these coercions. The calculus maintains abstractions dynamically without restricting type analysis.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Abstract data types; F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms

Languages, Theory

Keywords

abstract types, existential types, dynamic typing, generativity, opacity, encapsulation

1. INTRODUCTION

Type abstraction is an important tool for structuring programs and is a fundamental feature of most module systems. Languages like Modula [33, 4], CLU [16] and ML [17, 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 defines 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 [17]. 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 re : complex -> real
   val im : complex -> real
   val mul : complex * complex -> complex
end
```

An implementation 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 re(a,th) = a * cos th
   fun im(a,th) = a * sin th
   fun mul((a1,th1), (a2,th2)) =
       (a1*a2, rem(th1+th2, 2*pi))
end
```

An alternative implementation might use a cartesian representation for complex numbers. In any case, the abstraction operator `:>` hides the representation type `real * real` in the sense that, to the outside, the 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 compose and decompose complex numbers.

The advantage of the encapsulation idea implemented by type abstraction is twofold. First, the use of type abstraction enforces loose coupling: client code is compelled not to depend on internals of an abstract type's 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.

[*]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

Even more important is the second point: 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 $\theta \in [0; 2\pi[$. Type abstraction also prevents mixing values stemming from different (incompatible) implementations of the same siganture, e.g. a polar and a cartesian representation of complex numbers.

In their classic paper, Mitchell and Plotkin showed that abstract types can be formalised naturally as existential types [19], using the standard typing rules as found in constructive logic (e.g. System F [8]). We will review existential types and their relation 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, 14] and intensional type analysis [11] and extensional polymorphism [7]. 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:

$$\texttt{typecase}\ exp_1 : \tau_1\ \texttt{of}\ x : \tau_2\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3$$

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 2.3. An 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 polymorphic 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.

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 more critical:

```
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 apparent in section 2.3. 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 [30, 26] 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 [32]. While such a cast is still type-safe in the sense of not violating soundness, it clearly undermines any of the previously mentioned 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 invariant on the complex angle θ. Because type abstraction is no longer sufficient to ensure encapsulation, it is practically rendered useless.

1.2 Agenda

How can the conflict be solved? A simple possibility is to forbid analysis of abstract types altogether. For example, Harper and Morrisett curtly propose to distinguish between analyzable and non-analyzable types [11]. However, this clearly is overly 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(C.re x') ^
      (if im x' >= 0.0 then "+" else "-") ^
      Real.toString(abs(C.im x')) ^ "i"
    else ...
```

Similarly, in a language with type dynamic, it would become 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 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 analyzed.

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 means of dynamic type analysis. The second property effectively means that any type must be comparable (dynamically) to any other. We borrow the term *full reflexivity* from Trifonov, Saha and Shao [31], who 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 restriction is necessary for the weak typecase used here). 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* operational semantics: introduction of an abstract type dynamically generates a new type. Without generativity, type abstraction has no dynamic interpretation!

It should be noted that we solely discuss the requirements of 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 for 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 view external and internal use of dynamic typing as largely independent issues, and will not further consider the latter.

1.3 Plan

In section 2 we give a short overview over abstract types in the existential type encoding and investigate how it interferes with dynamic type analysis. In section 3 we introduce the basic features of the λ_{N}-calculus, which we propose as an alternative model. Section 4 discusses the calculus and its basic properties formally. In section 5 we look at its higher-order generalization and an extension incorporating an alternative, applicative notion of generativity. We review some related work in section 6 and conclude in section 7.

2. ABSTRACTION BY EXISTENTIAL QUANTIFICATION

In this section we will give a short recap of existential types and their correspondence to abstract types. We then discuss in more detail how this correspondence is destroyed by adding dynamic type analysis. We write $\equiv$ for syntactic equivalence (modulo α-conversion). $\text{FV}(e)$ denotes the set of free term variables of e defined in the usual way, and $\text{FTV}(t)$ the free type variables of type or term t. For clarity, we will sometimes use the notation (let $x = e_1$ in e_2) as an 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 Existential types

Figure 1 shows the syntax, evaluation and typing rules for existential types, as an extension to the plain polymorphic lambda calculus [3]. A value of existential type is usually called a *package*. It essentially is a pair $\langle\tau, e\rangle$, encapsulating a *representation type* τ and an *implementation* e. A package is assigned existential type $\exists\alpha.\tau'$ if its implementation matches the *signature type* τ', by replacing all occurrences of α with the actual representation type τ. Unfortunately, the signature type is not determined uniquely by the implementation and representation types alone, thus it has to be annotated explicitly, as apparent from the syntax.[1]

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 through variables α and x within e_2, respectively. Such an open expression evaluates to its body e_2, by substituting its bound variables with the actual type and value found in the package.[2]

$$\begin{array}{llll} \text{(types)} & \tau & ::= & \ldots \mid \exists\alpha.\tau \\ \text{(terms)} & e & ::= & \ldots \mid \langle\tau, e\rangle{:}\exists\alpha.\tau' \mid \text{open}\ \langle\alpha, x\rangle = e_1\ \text{in}\ e_2 \end{array}$$

$$\text{open}\ \langle\alpha, x\rangle = \langle\tau, e_1\rangle{:}\exists\alpha.\tau'\ \text{in}\ e_2 \quad\rightarrow\quad e_2[\alpha := \tau][x := e_1]$$

$$(\textsc{Pack})\ \frac{\Gamma \vdash e : \tau'[\alpha := \tau]}{\Gamma \vdash (\langle\tau, e\rangle{:}\exists\alpha.\tau') : \exists\alpha.\tau'}$$

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

Figure 1: Existential types

2.2 Encoding Abstract Types

An abstract type declaration introduces a new type bundled with a set of operations available on values of that type. An encoding via existential types is relatively straightforward. Let us assume that the polymorphic lambda calculus has been enriched further with product types and real numbers. Then the SML signature `COMPLEX` from the introduction can be represented by the type

$$\begin{aligned} \mathit{COMPLEX} \quad \equiv \quad & \exists\alpha.(\mathit{real} \times \mathit{real} \rightarrow \alpha) \times (\alpha \rightarrow \mathit{real}) \times \\ & (\alpha \rightarrow \mathit{real}) \times (\alpha \times \alpha \rightarrow \alpha) \end{aligned}$$

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):

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

ML style module access does not map as directly, because structure components are accessed using the dot notation, while a package has to be opened explicitly to make its content available. However, Cardelli and Leroy have shown that there exists a systematic translation from dot notation into plain existential types [5]. Essentially, the package encoding a structure is opened immediately. For our purpose,

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

might be encoded as

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

The typing rules for open plus the standard hygiene conventions for bound variables ensure that α is distinct from any other type variable in the same scope and thus behaves like

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

[2] Often the existential elimination form is written (unpack e_1 as α, x in e_2).

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

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

$$\begin{array}{rcl}\text{tcase } e_1 : \tau \text{ of } x : \tau \text{ then } e_2 \text{ else } e_3 & \to & e_2[x := e_1] \\ \text{tcase } e_1 : \tau_1 \text{ of } x : \tau_2 \text{ then } e_2 \text{ else } e_3 & \to & e_3 \quad (\tau_1 \neq \tau_2)\end{array}$$

Figure 2: A typecase extension

a "fresh" type. Moreover, it behaves fully abstract because in standard λ-calculi every expression is *parametric* [25] in all type variables, meaning that reduction can proceed uniformingly for all possible instantiations. In particular, the body e_2 of an open expression is parametric with respect to the bound variable α — 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 correspondence between existential types and abstract types.

2.3 Interaction with dynamic type analysis

Figure 2 specifies the semantics of our typecase, as an extension to the lambda calculus.[3] It provides only a simple form of type analysis but suffices to demonstrate the fundamental problem.

We have seen that the encoding of abstract types via existentials crucially relies on the parametricity property. That property breaks down in the face of operations for type analysis: if a polymorphic function is able to analyse its type argument using typecase, it obviously will not 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. Recall the typecase expression from section 1.1 that was incriminated to break the complex invariant. Expressed with existential types (and $\bot$) it may look like follows:

$$\begin{array}{l}\text{open } \langle \alpha, (\mathit{mk}, \mathit{re}, \mathit{im}, \mathit{mul})\rangle = \langle \mathit{real} \times \mathit{real}, \ldots\rangle \text{ in} \\ (\ldots \text{ tcase } (1, 1001\pi) : \mathit{real} \times \mathit{real} \text{ of } z : \alpha \\ \qquad \text{then } z \text{ else } \bot\ \ldots)\end{array}$$

But upon reduction of the open expression the type variable α naming the abstract type will be substituted and reveal

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

Both types in the typecase are now equal and the construct returns $z \equiv (1, 1001\pi)$ from its left branch having the same static type α as proper complex values.

3. TOWARDS A FORMAL SEMANTICS FOR DYNAMIC TYPE GENERATION

Although well known, 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, Pierce and Rémy [1] already suggested generativity as a solution, observing that dynamic opacity can be achieved by simply replacing the type variable bound by open with a "fresh" type constant during evaluation. Their idea amounts to changing the corresponding reduction rule to:

$$\text{open } \langle \alpha, x\rangle = \langle \tau, e_1\rangle{:}\exists\alpha.\tau' \text{ in } e_2 \quad \to \quad e_2[\alpha := t][x := e_1]$$

where t is a fresh type constant. Obviously, the representation type could no longer be analysed transparently. Unfortunately, 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 2.2:

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

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

$$(\lambda z{:}t.\ (\lambda(a, \theta) : \mathit{real} \times \mathit{real}\,.\ a \cdot \cos\theta)\ z)\ ((\lambda \ldots)\ (0, 1))$$

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 re.

Consequently, in order to make the idea of using generativity for abstraction work, we have to solve two problems: the concept of dynamic type "freshness" must be fleshed out formally, and transitions between abstract and concrete type must be managed in a sound way.

3.1 Generativity

Formalisms for describing dynamic generation of fresh *value* names are well developed. For example, the name restriction form $\nu n.P$ is a central feature of the π-calculus [28] and can be viewed as an expression that generates a new name n with local scope. Pitts' λ_ν-calculus [24] transfers that idea to the λ-calculus, although with a different implementation.

We introduce a similar construct, but for generating *type* names instead of value names. We will use the notation

$$\mathrm{N}\gamma{\approx}\tau.e$$

(with N read as upper-case nu) to introduce a fresh type name γ within expression e. N-bound names are subject to standard α-conversion rules. Because having a fresh type that is not inhabited by any values is not very interesting on its own, the N-form also declares a representation type τ. Within e the relation between the new type and its representation is known and can be used to construct and inspect values of type γ. Outside the scope of the corresponding N-expression that relation is not visible. We defer the discussion on how values are constructed to the next section.

How do we track generated type names? We chose to take the path of the π-calculus, where ν-expressions never get eliminated, but float outwards by special equivalence rules for *scope extrusion*.[4] In order to allow interaction between

[3] Adding typecase to the polymorphic lambda calculus without restricting the reduction relation breaks confluence. Consider $(\Lambda\alpha.\lambda x{:}\alpha.\text{tcase } x{:}\alpha \text{ of } y{:}\mathit{int} \text{ then } 1 \text{ else } 0)\ \mathit{int}\ 9$. Depending on which redex gets reduced first, this expression yields 1 or 0. For simplicity, we hence assume a call-by-value strategy.

[4] We also considered the alternative approach of introducing an explicit type store or heap as in the λ_ν-calculus, but that choice would produce a more complicated system.

a N-expression's body e and the expression's context, we incorporate reduction rules in a similar spirit, e.g.:

$$\begin{array}{lcll} (\mathrm{N}\gamma{\approx}\tau.e_1)\ e_2 & \rightarrow & \mathrm{N}\gamma{\approx}\tau.(e_1\ e_2) & (\gamma \notin \mathrm{FTN}(e_2)) \\ e_1\ (\mathrm{N}\gamma{\approx}\tau.e_2) & \rightarrow & \mathrm{N}\gamma{\approx}\tau.(e_1\ e_2) & (\gamma \notin \mathrm{FTN}(e_1)) \end{array}$$

The side conditions on the free type names of subterms (written $\mathrm{FTN}(e)$) ensure that the context cannot capture the type name γ.

The typing rule for N is straight-forward:

$$\frac{\Gamma, \gamma{\approx}\tau \vdash e : \tau'}{\Gamma \vdash \mathrm{N}\gamma{\approx}\tau.e : \tau'} (\gamma \notin \mathrm{FTN}(\tau'))$$

We need to be able to record the *type assertion* $\gamma{\approx}\tau$ in the environment, so that γ is properly related to its representation. The side condition keeps γ from escaping its scope, and is similar to the side condition of the (Open) typing rule for existential types.

3.2 Coercions

What does it mean for a new type γ to be 'represented' by τ? It certainly cannot mean that both types are simply equivalent (i.e. $\gamma = \tau$), since such an interpretation would not make γ particularly 'new' and inevitably bring us back to a semantics that violates dynamic opacity. We always have to be able to distinguish both types. Consequently, in order to avoid running into type preservation problems, we also need to be able to distinguish values of both types. Hence we require appropriate coercions to go from the abstract type to its representation and vice versa. We will use the notation

$$\{e\}^+_\gamma$$

for coercing a value e of representation type τ to the abstract type γ. Dually, we have

$$\{e'\}^-_\gamma$$

for the inverse coercion. Coercions allow an implementation to perform appropriate type conversions for any value of abstract type γ that crosses the abstraction boundaries in either direction. Positive coercions can be seen as constructors for values of the new type. They are eliminated only by negative coercions, the corresponding destructors. Consequently, the only evaluation possible with coercions is *cancellation*, implemented by a single reduction rule:

$$\{\{e\}^+_\gamma\}^-_\gamma \quad \rightarrow \quad e$$

The standard scoping rules guarantee that only coercions belonging to the same abstract type can cancel out each other.

The typing rules for coercions are obvious:

$$\frac{\Gamma \vdash e : \tau \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e\}^+_\gamma : \gamma} \qquad\qquad \frac{\Gamma \vdash e : \gamma \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e\}^-_\gamma : \tau}$$

Due to the side conditions in the rules, coercions are only available within the lexical scope of the corresponding type generator, thus the transition across abstraction boundaries can only be triggered from within the abstraction.

3.3 Example

Recall the *complex* example from the introduction and its encoding with existential types. Rewritten using type generation it looks as follows:

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

We still use an existential type. However, it is no longer utilized for providing abstraction, but merely for closing the signature of the abstraction (recall that γ itself may not appear in the signature). By putting the abstract type into a package it becomes accessible from the outside. The signature of the abstraction is uniquely determined by the uses of γ and corresponding coercions in its implementation. Hence *COMPLEX* is the only possible type for the package.

The rewritten abstract type C' can be used as before, via simply opening the package:

$$\mathrm{open}\ \langle\alpha, (\mathit{mk}, \mathit{re}, \_, \_)\rangle = C'\ \mathrm{in}\ \mathit{re}\ (\mathit{mk}\ (0,1))$$

The contained N-binder will be shifted outwards automatically by the corresponding scope extrusion rules.

3.4 A-posteriori abstraction

So far, to build an abstraction its implementation has to use coercions internally, in order to meet the intended signature type. We speak of *a priori* abstraction: an implementation must be tailored to a particular signature. On the other hand, abstraction based on existential types happens *a posteriori*: arbitrary parts of a given implementation's type are just hidden away without affecting the implementation itself, a construction sometimes called *sealing* in the context of modules [6]. Can we recover that flexibility?

The answer is yes: given an abstract signature type and a suitable implementation, we can systematically construct an expression that coerces the whole implementation into the desired signature type. Let e be an expression that shall be sealed with signature τ', abstracting away some type τ as γ. Assuming $e : \tau'[\gamma := \tau]$, the term produced by applying the transformation $\{e : \tau'\}^+_{\gamma\approx\tau}$ defined in figure 3 will have the desired signature type. The transformation is defined inductively over the signature type τ'. It constructs an η-expansion of the original term e, wrapping all parts e' that are supposed to get type γ into a suitable coercion $\{e'\}^+_\gamma$. Function types require an inverse treatment of their argument, where $e' : \gamma$ is replaced by $\{e'\}^-_\gamma$ instead. Since the inverse transformation is completely dual, we use the notation ± to capture both directions in a single set of rules. Coercion polarity is inverted for function arguments. The following lemma captures the central invariants:

Lemma 1 (A-posteriori Abstraction Invariants). *Let e be an expression and Γ an evironment with $\gamma{\approx}\tau \in \Gamma$.*

1. *If $\Gamma \vdash e : \tau'[\gamma := \tau]$, then $\Gamma \vdash \{e : \tau'\}^+_{\gamma\approx\tau} : \tau'$.*

2. *If $\Gamma \vdash e : \tau'$, then $\Gamma \vdash \{e : \tau'\}^-_{\gamma\approx\tau} : \tau'[\gamma := \tau]$.*

$$\begin{array}{lcll}
\{e:\gamma\}^{\pm}_{\gamma\approx\tau} & = & \{e\}^{\pm}_{\gamma} & \\
\{e:\gamma'\}^{\pm}_{\gamma\approx\tau} & = & e & (\gamma' \not\equiv \gamma) \\
\{e:\alpha\}^{\pm}_{\gamma\approx\tau} & = & e & \\
\{e:\tau_1 \to \tau_2\}^{\pm}_{\gamma\approx\tau} & = & \lambda x{:}\{\tau_1\}^{\pm}_{\gamma\approx\tau}.\{e\ \{x:\tau_1\}^{\mp}_{\gamma\approx\tau} : \tau_2\}^{\pm}_{\gamma\approx\tau} & \\
\{e:\forall\alpha.\tau_1\}^{\pm}_{\gamma\approx\tau} & = & \Lambda\alpha.\{e\ \alpha:\tau_1\}^{\pm}_{\gamma\approx\tau} & \\
 & & & \\
\{\tau'\}^{+}_{\gamma\approx\tau} & = & \tau' & \\
\{\tau'\}^{-}_{\gamma\approx\tau} & = & \tau'[\gamma := \tau] &
\end{array}$$

Figure 3: A-posteriori abstraction

With this in mind we introduce another building block of our calculus: in order to allow it to express sealing directly, we generalize coercion expressions to arbitrary types and make the transformation rules from figure 3 built-in by turning them into reduction rules. That is, coercions will have the actual form

$$\{e:\tau'\}^{\pm}_{\gamma\approx\tau}$$

In this generalized form, coercions are reminiscent of the abstraction brackets by Grossman, Morrisett and Zdancewic [10]. We will discuss that connection in section 6. Using generalized coercions, first-class abstract types can be represented by expressions of the form

$$\mathrm{N}\gamma{\approx}\tau.\langle\gamma, \{e:\tau'\}^{+}_{\gamma\approx\tau}\rangle$$

where τ is the representation type, τ' the signature, and e the implementation of the abstract type.

3.5 Polymorphic coercions

We have not yet given the typing rules for generalized coercions. The lemma from the previous section suggests

$$\frac{\Gamma \vdash e : \tau'[\gamma := \tau] \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e:\tau'\}^{+}_{\gamma\approx\tau} : \tau'} \qquad \frac{\Gamma \vdash e : \tau' \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e:\tau'\}^{-}_{\gamma\approx\tau} : \tau'[\gamma := \tau]}$$

Obviously, these rules generalize the typing rules for simple coercions, for $\{e\}^{\pm}_{\gamma} = \{e:\gamma\}^{\pm}_{\gamma\approx\tau}$. Unfortunately, they are not correct. First note that it is not possible to seal a value twice with respect to the same type γ: in the sealing rule, the type of e must be free of γ. Dually, unsealing always delivers a γ-free type. Consequently, a complication arises with *polymorphic coercions*. Consider the following term, for example:

$$P \equiv (\Lambda\alpha.\lambda x{:}\alpha.\{x:\alpha\}^{-}_{\gamma\approx\tau})\ \gamma$$

Under the above rules, P would be assigned type $\gamma \to \gamma$. However, standard β-reduction yields

$$\lambda x{:}\gamma.\{x:\gamma\}^{-}_{\gamma\approx\tau}$$

which has type $\gamma \to \tau$. Type preservation is violated. The problem is, that turned into reduction steps, the abstraction transformation is not static but interleaved with other reductions. Hence, the typing rules must account for potential substitutions. Intuitively, the γ-substitutions in the coercion typing rules must be delayed until all free type variables in τ' have been substituted. We deal with this by introducing *unsealed types* of the form

$$\{\tau'\}^{-}_{\gamma\approx\tau}$$

that essentially perform a substitution on τ', as the following set of special equivalence rules reveals:

$$\begin{array}{rcll}
\{\gamma\}^{-}_{\gamma\approx\tau} & = & \tau & \\
\{\gamma'\}^{-}_{\gamma\approx\tau} & = & \gamma' & (\gamma' \not\equiv \gamma) \\
\{\tau_1 \to \tau_2\}^{-}_{\gamma\approx\tau} & = & \{\tau_1\}^{-}_{\gamma\approx\tau} \to \{\tau_2\}^{-}_{\gamma\approx\tau} & \\
\{\forall\alpha.\tau_1\}^{-}_{\gamma\approx\tau} & = & \forall\alpha.\{\tau_1\}^{-}_{\gamma\approx\tau} &
\end{array}$$

There is no equivalence rule for type variables, so that a type of the form $\{\alpha\}^{-}_{\gamma\approx\tau}$ maintains the pending substitution for γ until α is substituted. Using this setup, sound typing rules can be given for coercions:

$$\frac{\Gamma \vdash e : \{\tau'\}^{-}_{\gamma\approx\tau} \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e:\tau'\}^{+}_{\gamma\approx\tau} : \tau'} \qquad \frac{\Gamma \vdash e : \tau' \qquad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \{e:\tau'\}^{-}_{\gamma\approx\tau} : \{\tau'\}^{-}_{\gamma\approx\tau}}$$

As long as no type variables occur in τ', they are equivalent to the rules above under the stated type equivalence. However, for the term P they correctly allow derivation of the type $\gamma \to \{\gamma\}^{-}_{\gamma\approx\tau} = \gamma \to \tau$. Likewise, they prohibit double sealing even in polymorphic cases. That implies that a polymorphic sealing function like

$$\Lambda\alpha.\lambda x{:}\alpha.\{x:\alpha\}^{+}_{\gamma\approx\tau}$$

is not well-typed. It must be formulated as either

$$\Lambda\alpha.\lambda x{:}\{\alpha\}^{-}_{\gamma\approx\tau}.\{x:\alpha\}^{+}_{\gamma\approx\tau}\ :\ \forall\alpha.\{\alpha\}^{-}_{\gamma\approx\tau} \to \alpha$$

or

$$\Lambda\alpha.\lambda x{:}\alpha.\{\{x:\alpha\}^{-}_{\gamma\approx\tau}:\alpha\}^{+}_{\gamma\approx\tau}\ :\ \forall\alpha.\alpha \to \alpha$$

In the former version, the function is not applicable to argumens that already contain occurrences of abstract type γ. The latter version lifts this restriction by simply unsealing any such potential values first.

4. THE λ_{N}-CALCULUS

We are now prepared to look at the calculus that we will refer to as λ_{N} as a whole.

4.1 Syntax

The syntax of the λ_{N}-calculus is shown in figure 4. Essentially, it is a polymorphic lambda calculus with recursive functions, extended with the constructs introduced in the previous section. It also contains a typecase expression, so that it allows discussion of the issues raised by dynamic typing. Values are defined in the usual way, but include abstract values of the form $\{\hat{e}:\tau\}^{+}_{\gamma\approx\tau'}$ with the side condition $\tau = \gamma$. Further, we distinguish a second subclass of terms called *results*, which is necessary to formulate deterministic evaluation rules for scope extrusion.

We will write $\lambda x{:}\tau.e$ for (fix $x'(x{:}\tau){:}\tau'.e$) if $x' \notin \mathrm{FV}(e)$ and τ' is the (unique) type of e. We also abbreviate $\{e:\gamma\}^{\pm}_{\gamma\approx\tau}$ to $\{e\}^{\pm}_{\gamma}$ if τ is clear from context. We will use notation for existential types in some examples, which can be encoded in λ_{N} using universal types in the usual way [20]. Also, we sometimes silently assume additional types like *int*, *real*, *unit* and respective constants, or cartesian products.

Environments are extended to include type assertions $\gamma{\approx}\tau$ for type names. They track the validity of coercions. We take the liberty to treat environments as sets of the contained assignments and assertions, or as finite functions mapping variables to types. We write $\mathrm{Dom}(\Gamma)$ to denote the set of all names and variables bound by Γ.

$$
\begin{array}{llll}
\text{(types)} & \tau & ::= & \alpha \mid \gamma \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \{\tau\}^{-}_{\gamma\approx\tau'} \\
\text{(terms)} & e & ::= & x \mid \mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau \mid \\
 & & & \mathrm{N}\gamma{\approx}\tau.e \mid \{e : \tau\}^{\pm}_{\gamma\approx\tau'} \mid \\
 & & & \mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 \\
\text{(results)} & \hat{e} & ::= & \hat{\hat{e}} \mid \mathrm{N}\gamma{\approx}\tau.\hat{e} \\
\text{(values)} & \hat{\hat{e}} & ::= & \mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e \mid \Lambda\alpha.e \mid \{\hat{\hat{e}} : \tau\}^{+}_{\gamma\approx\tau'}\ (\tau = \gamma) \\
\text{(env's)} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha \mid \Gamma, \gamma{\approx}\tau
\end{array}
$$

Figure 4: λ_{N} Syntax

4.2 Reduction

Figure 5 collects the one-step evaluation rules for λ_{N}. They can be categorized into five groups: standard β-reduction rules (1–2), coercion rules (3–6), type analysis rules (7–8), scope extrusion (9–13), and structural rules (14–19). Together, they specify a deterministic call-by-value evaluation strategy. We will write $=$ for convertibility with respect to the corresponding equivalence relation generated from $\to$.

Note that the coercion rules 4–6 are overloaded for both polarities. At function type we use the following definition for substituting type annotations depending on polarity:

$$\{\tau\}^{+}_{\gamma\approx\tau'} \quad = \quad \tau$$

Reduction of coercions and typecase is type-directed. The equivalence relation on types will be given in the next section. In rules 5 and 6 we implicitly require the equivalent types to be well-formed, i.e. $\vdash \tau_2{\to}\tau_1 : \Omega$ and $\vdash \forall\alpha.\tau_1 : \Omega$, respectively. For deterministic scope extrusion we have introduced the syntactic class of *results*. A result is a value prefixed by a sequence of N-binders. Scope extrusion only applies to the outermost binder of a result; evaluation has to proceed under a N-binder until its body has become a result. Intuitively, a result may be thought of as an expression's "return value" paired with the *heap* of type names its evaluation allocated. The type names generated in different subexpressions will all accumulate in the heap of the complete expressions's result. For example, consider the following reduction sequence:

$$
\begin{array}{ll}
 & (\lambda f{:}\mathit{int}{\to}\mathit{int}.f\,(\mathrm{N}\gamma_1{\approx}\tau_1.f\,4))\ (\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2{\approx}\tau_2.x) \\
\to & (\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2{\approx}\tau_2.x)\,(\mathrm{N}\gamma_1{\approx}\tau_1.(\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2'{\approx}\tau_2.x)\,4) \\
\to & (\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2{\approx}\tau_2.x)\,(\mathrm{N}\gamma_1{\approx}\tau_1.\mathrm{N}\gamma_2'{\approx}\tau_2.4) \\
\to & \mathrm{N}\gamma_1{\approx}\tau_1.(\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2{\approx}\tau_2.x)\,(\mathrm{N}\gamma_2'{\approx}\tau_2.4) \\
\to & \mathrm{N}\gamma_1{\approx}\tau_1.\mathrm{N}\gamma_2'{\approx}\tau_2.(\lambda x{:}\mathit{int}.\mathrm{N}\gamma_2{\approx}\tau_2.x)\,4 \\
\to & \mathrm{N}\gamma_1{\approx}\tau_1.\mathrm{N}\gamma_2'{\approx}\tau_2.\mathrm{N}\gamma_2{\approx}\tau_2.4
\end{array}
$$

Generation is fully dynamic, i.e. the number of type names generated is not determined statically. The following non-terminating expression will actually generate an infinite number of types:

$$(\mathrm{fix}\, f(x{:}\mathit{unit}){:}\mathit{unit}.\mathrm{N}\gamma{\approx}\tau.f\ x)\ ()$$

4.3 Typing

The typing and well-formedness rules of the calculus are a simple extension of the rules for the polymorphic λ-calculus. We need the usual three judgment forms:

$\vdash \Gamma : \diamond$	well-formedness of environments
$\Gamma \vdash \tau : \Omega$	well-formedness of types
$\Gamma \vdash e : \tau$	well-typedness of terms

In comparison to the plain λ-calculus, λ_{N} adds the previously discussed typing rules for N, coercions, and typecase, as well as extended well-formedness rules for dealing with type names and type assertions. The presence of a non-trivial type equivalence relation requires an additional structural typing rule (Equiv) for assigning of equivalent types to a term. The type equivalence relation is defined in figure 7.

It is not difficult to show that type soundness properties hold for λ_{N}:

Theorem 1 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$.*

Theorem 2 (Progress). *If $\cdot \vdash e : \tau$ (i.e. e is closed), then either $e \equiv \hat{e}$ for some result $\hat{e}$, or $e \to e'$ for some e'.*

Note that if $\hat{e}$ is closed, then $\hat{e} \not\equiv \{\hat{e}'\}^{+}_{\gamma}$. We also have

Theorem 3 (Unique Types). *Whenever $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$ then $\tau = \tau'$.*

4.4 Opacity

To see how opacity is still preserved in the non-parametric setting of λ_{N} let us go back to the λ_{N}-encoding of complex numbers, as shown in section 3.3. It is safe with respect to dynamic typing, as the reduction of the expression representing the second offending example from section 1.1 shows:

$$
\begin{array}{l}
\quad \mathrm{open}\ \langle\alpha, \_\rangle = \mathrm{N}\gamma{\approx}\mathit{real} \times \mathit{real}\,.\,\langle\gamma, \dots\rangle\ \mathrm{in} \\
\quad (\dots\ \mathrm{tcase}\ (1, 1001\pi) : \mathit{real} \times \mathit{real}\ \mathrm{of}\ z : \alpha\ \mathrm{then}\ z\ \mathrm{else} \perp\ \dots) \\
\to \mathrm{N}\gamma{\approx}\mathit{real} \times \mathit{real}\,.\,\mathrm{open}\ \langle\alpha, \_\rangle = \langle\gamma, \dots\rangle\ \mathrm{in} \\
\quad (\dots\ \mathrm{tcase}\ (1, 1001\pi) : \mathit{real} \times \mathit{real}\ \mathrm{of}\ z : \alpha\ \mathrm{then}\ z\ \mathrm{else} \perp\ \dots) \\
\to \mathrm{N}\gamma{\approx}\mathit{real} \times \mathit{real}\,. \\
\quad (\dots\ \mathrm{tcase}\ (1, 1001\pi) : \mathit{real} \times \mathit{real}\ \mathrm{of}\ z : \gamma\ \mathrm{then}\ z\ \mathrm{else} \perp\ \dots) \\
\to \mathrm{N}\gamma{\approx}\mathit{real} \times \mathit{real}\,.\,(\dots \perp \dots)
\end{array}
$$

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

More generally, consider a closed, well-typed function of the form $\Lambda\alpha.\lambda x{:}\alpha.e$ (which may contain random uses of typecase). Applied to an abstract type γ and value $\hat{\hat{e}} : \gamma$, its result will be independent of γ's respective representation type, as well as of the concrete value $\hat{\hat{e}}$. Formally, we can phrase the following property:

Theorem 4 (Opacity). *Let e be an expression with $\alpha, x{:}\alpha \vdash e : \tau$. Assume a set of values $\hat{\hat{e}}_i$ ($i = 1, \dots, n$) such that $\gamma_i{\approx}\tau_i \vdash \hat{\hat{e}}_i : \gamma_i$. Let $\sigma_i = [\alpha := \gamma_i, x := \hat{\hat{e}}_i]$ for all i. If $e\sigma_1$ is not a value then there is an e' with $\alpha, x{:}\alpha \vdash e' : \tau$ such that $e\sigma_i \to e'\sigma_i$ for all σ_i.*

Opacity subsumes *value abstraction* [10], but is slightly stronger because it also implies *type abstraction*, i.e. independence from abstract type representations.

4.5 Sharing

Despite opacity, λ_{N} still allows expressing (dynamic) *sharing* between abstract types. For example, the following function checks if two given complex types are compatible and mixes operations from both of them if that is the case:

$$
\begin{array}{l}
\lambda C_1 : \mathit{COMPLEX}.\ \lambda C_2 : \mathit{COMPLEX}. \\
\quad \mathrm{open}\ \langle\alpha_1, (\mathit{mk}_1, \_, \_, \_)\rangle = C_1\ \mathrm{in} \\
\quad \mathrm{open}\ \langle\alpha_2, (\_, \_, \mathit{im}_2, \_)\rangle = C_2\ \mathrm{in} \\
\quad \mathrm{tcase}\ \mathit{im}_2 : \alpha_2 \to \mathit{real}\ \mathrm{of}\ \mathit{im}_2' : \alpha_1 \to \mathit{real} \\
\qquad \mathrm{then}\ \mathit{im}_2'(\mathit{mk}_1(0, 2))\ \mathrm{else} \perp
\end{array}
$$

$$
\begin{array}{lrcll}
(1) & (\mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.e)\ \hat{\hat{e}} & \to & e[x_1 := \mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.e, x_2 := \hat{\hat{e}}] & \\
(2) & (\Lambda\alpha.e)\ \tau & \to & e[\alpha := \tau] & \\
 & & & & \\
(3) & \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'} & \to & \hat{\hat{e}} & (\text{if } \tau_1 = \tau_2 = \gamma) \\
(4) & \{\hat{\hat{e}} : \tau\}^{\pm}_{\gamma\approx\tau'} & \to & \hat{\hat{e}} & (\text{if } \tau = \gamma' \not\equiv \gamma) \\
(5) & \{\hat{\hat{e}} : \tau\}^{\pm}_{\gamma\approx\tau'} & \to & \mathrm{fix}\,x_1(x_2 : \{\tau_2\}^{\pm}_{\gamma\approx\tau'}) : \{\tau_1\}^{\pm}_{\gamma\approx\tau'}. & (\text{if } \tau = \tau_2 {\to} \tau_1;\ x_1, x_2 \notin \mathrm{FV}(\hat{\hat{e}})) \\
 & & & \quad \{\hat{\hat{e}}\ \{x_2 : \tau_2\}^{\mp}_{\gamma\approx\tau'} : \tau_1\}^{\pm}_{\gamma\approx\tau'} & \\
(6) & \{\hat{\hat{e}} : \tau\}^{\pm}_{\gamma\approx\tau'} & \to & \Lambda\alpha.\{\hat{\hat{e}}\ \alpha : \tau_1\}^{\pm}_{\gamma\approx\tau'} & (\text{if } \tau = \forall\alpha.\tau_1) \\
 & & & & \\
(7) & \mathrm{tcase}\ \hat{\hat{e}} : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & \to & e_2[x := \hat{\hat{e}}] & (\text{if } \tau_1 = \tau_2) \\
(8) & \mathrm{tcase}\ \hat{\hat{e}} : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & \to & e_3 & (\text{if } \tau_1 \neq \tau_2) \\
 & & & & \\
(9) & (\mathrm{N}\gamma{\approx}\tau.\hat{e})\ e & \to & \mathrm{N}\gamma{\approx}\tau.\hat{e}\ e & (\gamma \notin \mathrm{FTN}(e)) \\
(10) & \hat{\hat{e}}\ (\mathrm{N}\gamma{\approx}\tau.\hat{e}) & \to & \mathrm{N}\gamma{\approx}\tau.\hat{\hat{e}}\ \hat{e} & (\gamma \notin \mathrm{FTN}(\hat{\hat{e}})) \\
(11) & (\mathrm{N}\gamma{\approx}\tau.\hat{e})\ \tau' & \to & \mathrm{N}\gamma{\approx}\tau.\hat{e}\ \tau' & (\gamma \notin \mathrm{FTN}(\tau')) \\
(12) & \{\mathrm{N}\gamma{\approx}\tau.\hat{e} : \tau''\}^{\pm}_{\gamma'\approx\tau'} & \to & \mathrm{N}\gamma{\approx}\tau.\{\hat{e} : \tau''\}^{\pm}_{\gamma'\approx\tau'} & (\gamma \not\equiv \gamma'; \gamma \notin \mathrm{FTN}(\tau', \tau'')) \\
(13) & \mathrm{tcase}\ \mathrm{N}\gamma{\approx}\tau.\hat{e}_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & \to & \mathrm{N}\gamma{\approx}\tau.\mathrm{tcase}\ \hat{e}_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & (\gamma \notin \mathrm{FTN}(\tau_1, \tau_2, e_2, e_3)) \\
 & & & & \\
(14) & e\ e_2 & \to & e'\ e_2 & (\text{if } e \to e') \\
(15) & \hat{\hat{e}}\ e & \to & \hat{\hat{e}}\ e' & (\text{if } e \to e') \\
(16) & e\ \tau & \to & e'\ \tau & (\text{if } e \to e') \\
(17) & \mathrm{N}\gamma{\approx}\tau.e & \to & \mathrm{N}\gamma{\approx}\tau.e' & (\text{if } e \to e') \\
(18) & \{e : \tau'\}^{\pm}_{\gamma\approx\tau} & \to & \{e' : \tau'\}^{\pm}_{\gamma\approx\tau} & (\text{if } e \to e') \\
(19) & \mathrm{tcase}\ e : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & \to & \mathrm{tcase}\ e' : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 & (\text{if } e \to e')
\end{array}
$$

Figure 5: λ_{N} Reduction

That is, we are able to find out dynamically whether abstract types are compatible, although we cannot look at their representation. This ability is important for dynamic programming with abstract types. For example, consider a scenario where a process retrieves values of abstract types from different, statically undetermined locations. In order to combine those values, the program must be able to dynamically verify their compatibility.

5. HIGHER-ORDER TYPES

The λ_{N}-calculus is equipped with a second-order type system. That enables it to model generativity of proper types. However, many programming languages allow the definition of "polymorphic" abstract types. E.g. in SML, we can define a polymorphic abstract stack:

```
signature STACK =
sig
   type 'a stack
   val empty : 'a stack
   val push : 'a * 'a stack -> 'a stack
   val pop : 'a stack -> 'a * 'a stack
end
structure Stack :> STACK =
struct
   type 'a stack = 'a list
   ...
end
```

Such higher-order type abstraction can be captured by extending the calculus with higher-order types, allowing to define

$$\mathrm{N}\gamma{\approx}(\lambda\alpha{:}\Omega.\mathit{list}\ \alpha).e$$

(assuming existence of a type constructor $\mathit{list} : \Omega \to \Omega$). The essentials of the higher-order calculus are shown in figure 8. Besides the standard modifications to typing rules that are necessary when moving from System F to F^{ω} (which have been omitted for space reasons) [8, 20], typing has to deal with higher-kinded type names. As a minor technicality, we also add a kind annotation to N-binders that is not strictly necessary in $\lambda^{\omega}_{\mathrm{N}}$ per se, but needed for the extension presented in the next section. We will omit this annotation where obvious. The type equivalence relation (omitted) has to be extended with β and η-rules as well as obvious rules for pushing unsealed types through type abstraction and application. Values of abstract type no longer need to have plain type γ, but generally have a type of the form $\gamma\ \tau_1 \ldots \tau_n$ (with $n \geq 0$), which we abbreviate as $\gamma\ \vec{\tau}$. For example, in an encoding of the stack abstraction, integer stack values have shape $\{\hat{\hat{e}} : \gamma\ \mathit{int}\}^{+}_{\gamma\approx\lambda\alpha:\Omega.\mathit{list}\ \alpha} : \gamma\ \mathit{int}$.

The primary complication in the higher-order extension appears with the reduction rules for coercions: an abstract type may have the form $\gamma_1\ \vec{\tau}$, and some γ_2 may appear in $\vec{\tau}$. For example, consider an expression

$$\{\hat{\hat{e}} : \gamma_1\ \gamma_2\}^{-}_{\gamma_2\approx\tau_2}$$

How can the coercion be pushed inward, across the unrelated abstract type γ_1? Fortunately, the canonical forms lemma for the calculus implies $\hat{\hat{e}} \equiv \{\hat{\hat{e}}' : \gamma_1\ \gamma_2\}^{+}_{\gamma_1\approx\tau_1}$. We can hence exchange both coercions, yielding simpler ones that can be

$$\frac{}{\vdash \cdot : \diamond} \qquad \frac{\vdash \Gamma : \diamond \qquad \Gamma \vdash \tau : \Omega}{\vdash \Gamma, x{:}\tau : \diamond}(x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\vdash \Gamma : \diamond}{\vdash \Gamma, \alpha : \diamond}(\alpha \notin \mathrm{Dom}(\Gamma)) \qquad \frac{\vdash \Gamma : \diamond \qquad \Gamma \vdash \tau : \Omega}{\vdash \Gamma, \gamma{\approx}\tau : \diamond}(\gamma \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\vdash \Gamma : \diamond \quad \alpha \in \Gamma}{\Gamma \vdash \alpha : \Omega} \quad \frac{\vdash \Gamma : \diamond \quad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \gamma : \Omega} \quad \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega}$$

$$\frac{\Gamma, \alpha \vdash \tau : \Omega}{\Gamma \vdash \forall\alpha.\tau : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \Omega \quad \gamma{\approx}\tau_2 \in \Gamma}{\Gamma \vdash \{\tau_1\}^-_{\gamma\approx\tau_2} : \Omega}$$

$$(\textsc{Id})\ \frac{\vdash \Gamma : \diamond \quad x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\textsc{App})\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$(\textsc{Fix})\ \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega \qquad \Gamma, x_1{:}\tau_2{\to}\tau_1, x_2{:}\tau_2 \vdash e : \tau_1}{\Gamma \vdash (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e) : \tau_2 \to \tau_1}$$

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

$$(\textsc{New})\ \frac{\Gamma \vdash \tau' : \Omega \qquad \Gamma, \gamma{\approx}\tau' \vdash e : \tau}{\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'.e : \tau}(\gamma \notin \mathrm{FTN}(\tau))$$

$$(\textsc{Seal})\ \frac{\Gamma \vdash e : \{\tau\}^-_{\gamma\approx\tau'} \quad \gamma{\approx}\tau' \in \Gamma}{\Gamma \vdash \{e : \tau\}^+_{\gamma\approx\tau'} : \tau}$$

$$(\textsc{Unseal})\ \frac{\Gamma \vdash e : \tau \quad \gamma{\approx}\tau' \in \Gamma}{\Gamma \vdash \{e : \tau\}^-_{\gamma\approx\tau'} : \{\tau\}^-_{\gamma\approx\tau'}}$$

$$(\textsc{Tcase})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \tau_2 : \Omega \quad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3) : \tau}$$

$$(\textsc{Equiv})\ \frac{\Gamma \vdash e : \tau' \qquad \tau' = \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash e : \tau}$$

Figure 6: λ_{N} Typing

$$\frac{}{\tau = \tau} \qquad \frac{\tau' = \tau}{\tau = \tau'} \qquad \frac{\tau = \tau' \quad \tau' = \tau''}{\tau = \tau''}$$

$$\frac{\tau_1 = \tau_1' \quad \tau_2 = \tau_2'}{\tau_1{\to}\tau_2 = \tau_1'{\to}\tau_2'} \qquad \frac{\tau = \tau'}{\forall\alpha.\tau = \forall\alpha.\tau'} \qquad \frac{\tau_1 = \tau_1'}{\{\tau_1\}^-_{\gamma\approx\tau_2} = \{\tau_1'\}^-_{\gamma\approx\tau_2}}$$

$$\frac{}{\{\gamma\}^-_{\gamma\approx\tau} = \tau} \qquad \frac{}{\{\gamma'\}^-_{\gamma\approx\tau} = \gamma'}(\gamma \not\equiv \gamma')$$

$$\frac{}{\{\tau_1{\to}\tau_2\}^-_{\gamma\approx\tau_3} = \{\tau_1\}^-_{\gamma\approx\tau_3}{\to}\{\tau_2\}^-_{\gamma\approx\tau_3}}$$

$$\frac{}{\{\forall\alpha.\tau_1\}^-_{\gamma\approx\tau_2} = \forall\alpha.\{\tau_1\}^-_{\gamma\approx\tau_2}}(\alpha \notin \mathrm{FTV}(\tau_2))$$

Figure 7: λ_{N} Type equivalence

coped with as usual (or by another step of the same sort):

$$\{\{\hat{\hat{e}}' : \gamma_1\,\gamma_2\}^+_{\gamma_1} : \gamma_1\,\gamma_2\}^-_{\gamma_2} \quad \to \quad \{\{\hat{\hat{e}}' : \tau_1\,\gamma_2\}^-_{\gamma_2} : \gamma_1\,\tau_2\}^+_{\gamma_1}$$

In general however, γ_1 can occur in τ_2, or γ_2 may occur in τ_1 (since type assertions cannot be circular, at most one of these cases can actually arise at a time). Either way, the reduct would not be well-typed. We hence have to insert an auxiliary coercion, reducing to either

$$\{\{\{\hat{\hat{e}}' : \tau_1\,\gamma_2\}^-_{\gamma_2} : \tau_1\,\tau_2\}^-_{\gamma_1} : \gamma_1\,\tau_2\}^+_{\gamma_1} \qquad (\gamma_2 \notin \mathrm{FTN}(\tau_1))$$

or

$$\{\{\{\hat{\hat{e}}' : \tau_1\,\gamma_2\}^-_{\gamma_2} : \tau_1\,\{\tau_2\}^-_{\gamma_1}\}^+_{\gamma_2} : \gamma_1\,\tau_2\}^+_{\gamma_1} \quad (\gamma_1 \notin \mathrm{FTN}(\tau_2))$$

depending on which case actually applies. In a similar vein, positive coercions have to be handled. The modified reduction rules for higher-order coercions are shown in figure 8. To keep side conditions readable we use the following conventions: (1) $\gamma \not\equiv \gamma'$; (2) $\{\hat{\hat{e}}{:}\tau'\}^{\pm}_{\gamma\approx\tau}$ matches any term $\{\hat{\hat{e}}{:}\tau''\}^{\pm}_{\gamma\approx\tau}$ with $\tau'' = \tau'$ and $\cdot \vdash \tau'' : \kappa$; (3) $\gamma \in \mathrm{FTN}(\tau)$ means $\forall\tau'.\tau' = \tau \Rightarrow \gamma \in \mathrm{FTN}(\tau')$. Reduction rule (4a) generalizes the previous rule (4), while rules (4b)–(4e) treat the cases discussed above. They also handle γ' occuring in its own argument vector $\vec{\tau}$.

Soundness results extend to $\lambda_{\mathrm{N}}^{\omega}$ in a straight-forward manner. Opacity has to be reformulated as follows:

THEOREM 5 ($\lambda_{\mathrm{N}}^{\omega}$ OPACITY). *Let e be an expression with $\alpha, x{:}\alpha\,\vec{\tau} \vdash e : \tau$ for some $\vec{\tau}$. Assume a set of values $\hat{\hat{e}}_i$ ($i = 1, \ldots, n$) such that $\gamma_i{\approx}\tau_i \vdash \hat{\hat{e}}_i : \gamma_i\,\vec{\tau}$. Let $\sigma_i = [\alpha := \gamma_i, x := \hat{\hat{e}}_i]$ for all i. If $e\sigma_1$ is not a value then there is an e' with $\alpha, x{:}\alpha\,\vec{\tau} \vdash e' : \tau$ such that $e\sigma_i \to e'\sigma_i$ for all σ_i.*

5.1 Applicative generativity

A function containing a N-binder will produce a new copy of the binder on every application. For example,

$$\begin{aligned} & \mathrm{let}\ f = \lambda x{:}\mathit{unit}.\mathrm{N}\gamma{\approx}\mathit{int}.\langle\gamma, \{13\}^+_{\gamma}\rangle\ \mathrm{in}\ (f\ (), f\ ()) \\ = \quad & \mathrm{N}\gamma_1{\approx}\mathit{int}.\mathrm{N}\gamma_2{\approx}\mathit{int}.(\langle\gamma_1, \{13\}^+_{\gamma_1}\rangle, \langle\gamma_2, \{13\}^+_{\gamma_2}\rangle) \end{aligned}$$

The standard α-renaming rules make γ_1 and γ_2 two incompatible types. That behaviour is analoguous to functors in SML, where the following snippet declares two incompatible types `X1.t` and `X2.t`:

```
functor F () :> sig type t; val x : t end
    = struct type t = int; val x = 13 end
structure X1 = F ()
structure X2 = F ()
```

In other words, λ_{N} implements a *fully generative* type abstraction discipline. There are alternative approaches to functors that make `X1.t` and `X2.t` equivalent types. After Leroy, such functors are called *applicative* [12]. Dreyer, Crary and Harper propose two alternative sealing operators `:>` and `::` to allow generative and applicative functors to coexist [6]. In their approach, replacing `:>` by `::` in the above example would yield compatible types `X1.t` and `X2.t`.

We can incorporate applicative generativity into $\lambda_{\mathrm{N}}^{\omega}$ by extending it with a second form of N-binder, which we distinguish by marking it as follows:

$$\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.e$$

Typing for this binder is the same as for plain N, but it comes with different reduction rules — the fundamental idea being that it is lifted out of lambdas prior to β-reduction, avoiding

$$
\begin{array}{llll}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \\
\text{(types)} & \tau & ::= & \ldots \mid \forall\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
\text{(terms)} & e & ::= & \ldots \mid \Lambda\alpha{:}\kappa.e \mid \mathrm{N}\gamma{:}\kappa{\approx}\tau.e \\
\text{(values)} & \hat{\hat{e}} & ::= & \ldots \mid \Lambda\alpha{:}\kappa.e \mid \{\hat{\hat{e}} : \tau\}^{+}_{\gamma\approx\tau'}\ (\tau = \gamma\,\vec{\tau}) \\
\text{(env's)} & \Gamma & ::= & \ldots \mid \Gamma, \alpha{:}\kappa
\end{array}
$$

$$
\frac{\gamma{\approx}\tau \in \Gamma \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \gamma : \kappa}
$$

$$
(\textsc{New})\ \frac{\Gamma \vdash \tau' : \kappa \qquad \Gamma, \gamma{\approx}\tau' \vdash e : \tau}{\Gamma \vdash \mathrm{N}\gamma{:}\kappa{\approx}\tau'.e : \tau}(\gamma \notin \mathrm{FTN}(\tau))
$$

$$
\begin{array}{llcll}
(3) & \{\{\hat{\hat{e}} : \gamma\,\vec{\tau}\}^{+}_{\gamma\approx\tau} : \gamma\,\vec{\tau}\}^{-}_{\gamma\approx\tau} & \to & \hat{\hat{e}} & \\
(4a) & \{\{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'\approx\tau'} : \gamma'\,\vec{\tau}\}^{\pm}_{\gamma\approx\tau} & \to & \{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'} & (\text{if } \gamma \notin \mathrm{FTN}(\vec{\tau})) \\
(4b) & \{\{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'\approx\tau'} : \gamma'\,\vec{\tau}\}^{+}_{\gamma\approx\tau} & \to & \{\{\{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \vec{\tau}\}^{+}_{\gamma'} & (\text{if } \gamma' \notin \mathrm{FTN}(\tau); \gamma \in \mathrm{FTN}(\vec{\tau})) \\
(4c) & \{\{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'\approx\tau'} : \gamma'\,\vec{\tau}\}^{-}_{\gamma\approx\tau} & \to & \{\{\{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \{\vec{\tau}\}^{-}_{\gamma}\}^{+}_{\gamma'} & (\text{if } \gamma' \notin \mathrm{FTN}(\tau); \gamma \in \mathrm{FTN}(\vec{\tau})) \\
(4d) & \{\{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'\approx\tau'} : \gamma'\,\vec{\tau}\}^{+}_{\gamma\approx\tau} & \to & \{\{\{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{+}_{\gamma'} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \vec{\tau}\}^{+}_{\gamma'} & (\text{if } \gamma' \in \mathrm{FTN}(\tau); \gamma \in \mathrm{FTN}(\vec{\tau})) \\
(4e) & \{\{\hat{\hat{e}} : \gamma'\,\vec{\tau}'\}^{+}_{\gamma'\approx\tau'} : \gamma'\,\vec{\tau}\}^{-}_{\gamma\approx\tau} & \to & \{\{\{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} : \gamma'\ \{\vec{\tau}\}^{-}_{\gamma}\}^{+}_{\gamma'} & (\text{if } \gamma' \in \mathrm{FTN}(\tau); \gamma \in \mathrm{FTN}(\vec{\tau}))
\end{array}
$$

Figure 8: The $\lambda^{\omega}_{\mathrm{N}}$-calculus (excerpt)

duplication. Figure 9 shows the extended syntax. For reasons that will become appearent shortly, we call the class of terms extended with $\check{\mathrm{N}}$ *pre-terms*. Again, we need an additional class of results, called *pre-results*, to get a deterministic evaluation relation. Pre-results are ordinary $\lambda^{\omega}_{\mathrm{N}}$-terms prefixed by a sequence of $\check{\mathrm{N}}$-binders.

$$
\begin{array}{llll}
\text{(pre-terms)} & \check{\check{e}} & ::= & x \mid \mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{\check{e}} \mid \check{\check{e}}_1\ \check{\check{e}}_2 \mid \Lambda\alpha{:}\kappa.\check{\check{e}} \mid \\
 & & & \check{\check{e}}\ \tau \mid \mathrm{N}\gamma{:}\kappa{\approx}\tau.\check{\check{e}} \mid \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{\check{e}} \mid \{\check{\check{e}} : \tau\}^{\pm}_{\gamma\approx\tau'} \mid \\
 & & & \text{tcase } \check{\check{e}}_1 : \tau_1 \text{ of } x : \tau_2 \text{ then } \check{\check{e}}_2 \text{ else } \check{\check{e}}_3 \\
\text{(pre-results)} & \check{e} & ::= & e \mid \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}
\end{array}
$$

Figure 9: Syntax of pre-terms

Figure 10 reveals the extended reduction relation. It contains scope extrusion rules for $\check{\mathrm{N}}$ (20–29), structural rules on pre-terms (30–40), and a single *fixation* rule (41). Unlike plain N, the scope of $\check{\mathrm{N}}$ can be extruded across binders like fix, Λ and plain N, as well as from the branches of a typecase. Moreover, pre-term reduction may proceed under all these constructs. The only interesting rules are (23) and (25), where a $\check{\mathrm{N}}$-binder has to be lifted out of a binder for a type variable. Since that variable may occur free in the respective representation type, special care has to be taken. We borrow an idea from Russo [27], who models applicative functors by representing abstract types in their result signature as higher-order abstractions over all types of the functor's argument. In a similar vein, the aforementioned reduction rules raise the type name γ to higher order by abstracting over the respective type variable whose binding the scope will be extruded from. As a simple example, consider the following reduction:

$$
\begin{array}{rl}
 & \text{let } f = \Lambda\alpha{:}\Omega.\lambda x{:}\alpha.\check{\mathrm{N}}\gamma{\approx}\alpha.\langle\gamma, \{x : \gamma\}^{+}_{\gamma\approx\alpha}\rangle \\
 & \ \text{in } (f\ \mathit{int}\ 3, f\ \mathit{int}\ 4, f\ \mathit{real}\ 5.0) \\
= & \text{let } f = \check{\mathrm{N}}\gamma{\approx}(\lambda\alpha{:}\Omega.\alpha).\Lambda\alpha{:}\Omega.\lambda x{:}\alpha.\langle\gamma\ \alpha, \{x : \gamma\ \alpha\}^{+}_{\gamma\approx\lambda\alpha:\Omega.\alpha}\rangle \\
 & \ \text{in } (f\ \mathit{int}\ 3, f\ \mathit{int}\ 4, f\ \mathit{real}\ 5.0) \\
= & \check{\mathrm{N}}\gamma{\approx}(\lambda\alpha{:}\Omega.\alpha).\text{let } f = \Lambda\alpha{:}\Omega.\lambda x{:}\alpha.\langle\gamma\ \alpha, \{x : \gamma\ \alpha\}^{+}_{\gamma\approx\lambda\alpha:\Omega.\alpha}\rangle \\
 & \qquad\qquad\qquad \text{in } (f\ \mathit{int}\ 3, f\ \mathit{int}\ 4, f\ \mathit{real}\ 5.0) \\
= & \mathrm{N}\gamma{\approx}(\lambda\alpha{:}\Omega.\alpha).(\langle\gamma\ \mathit{int}, \{3\}^{+}_{\gamma}\rangle, \langle\gamma\ \mathit{int}, \{4\}^{+}_{\gamma}\rangle, \langle\gamma\ \mathit{real}, \{5.0\}^{+}_{\gamma}\rangle)
\end{array}
$$

The first two packages carry the same type $\gamma\ \mathit{int}$, while the last one contains the different type $\gamma\ \mathit{real}$.

Since scope extrusion is the only actual evaluation taking place on non-plain pre-terms, the effect of reducing pre-terms is lifting out all $\check{\mathrm{N}}$-binders until the pre-term has become a pre-result, i.e. its body is a plain term. Once that form has been reached, the fixation rule (41) turns all its now outermost $\check{\mathrm{N}}$-binders into plain N-binders, leaving an ordinary $\lambda^{\omega}_{\mathrm{N}}$-term, for which evaluation proceeds as before. In other words, evaluation happens in two phases: first, rules (20–41) transform the pre-term into a plain term, then the rules (1–19) perform proper evaluation. The first phase will always terminate:

Theorem 6 (Finite Pre-term Reduction). *Every well-typed pre-term $\check{\check{e}}$ reduces to a plain term e by a finite reduction sequence (involving only rules (20)–(41)).*

In this light, it is valid to view pre-terms as an external language, which is transformed into the internal core language of plain terms prior to evaluation via a static elaboration process.

6. RELATED WORK

Although being relatively simple in spirit, to our knowledge there is no previous work that isolates the dynamic 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 influential theories for ML modules [12, 15, 27, 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 reduction semantics.

One notable exception is Sewell, who uses generativity for modelling certain aspects of type abstraction [29]. However, in his system generated abstract types are recorded as manifestly equal to their representation in a global environment, so that opacity is not properly maintained dynamically.

Glew presented a calculus for generating new tagged types at runtime and dispatching on them [9]. His system is more complex than ours in order to allow for hierarchical types, but it is not fully reflexive since untagged types cannot be analysed.

The work most relevant to ours is by Grossman, Morrisett and Zdancewic on proof techniques for abstraction

$$\mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{e} \qquad (\gamma \notin \mathrm{FTN}(\tau_1, \tau_2)) \tag{20}$$

$$(\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e})\ \check{\check{e}} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}\ \check{\check{e}} \qquad (\gamma \notin \mathrm{FTN}(\check{\check{e}})) \tag{21}$$

$$e\ (\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}) \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.e\ \check{e} \qquad (\gamma \notin \mathrm{FTN}(e)) \tag{22}$$

$$\Lambda\alpha{:}\kappa'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa'{\to}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau).\Lambda\alpha{:}\kappa'.\check{e}[\gamma := \gamma\ \alpha] \tag{23}$$

$$(\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e})\ \tau' \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}\ \tau' \qquad (\gamma \notin \mathrm{FTN}(\tau')) \tag{24}$$

$$\mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa'{\to}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau[\gamma' := \alpha]).\mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{e}[\gamma := \gamma\ \gamma'] \qquad (\gamma \not\equiv \gamma'; \alpha \notin \mathrm{FTV}(\tau); \gamma \notin \mathrm{FTN}(\tau')) \tag{25}$$

$$\{\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} : \tau''\}^{\pm}_{\gamma'\approx\tau'} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\{\check{e} : \tau''\}^{\pm}_{\gamma'\approx\tau'} \qquad (\gamma \not\equiv \gamma'; \gamma \notin \mathrm{FTN}(\tau', \tau'')) \tag{26}$$

$$\mathrm{tcase}\ \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}_2\ \mathrm{else}\ \check{\check{e}}_3 \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\mathrm{tcase}\ \check{e}_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}_2\ \mathrm{else}\ \check{\check{e}}_3 \qquad (\gamma \notin \mathrm{FTN}(\tau_1, \tau_2, \check{\check{e}}_2, \check{\check{e}}_3)) \tag{27}$$

$$\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}_2\ \mathrm{else}\ \check{\check{e}}_3 \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{e}_2\ \mathrm{else}\ \check{\check{e}}_3 \qquad (\gamma \notin \mathrm{FTN}(\tau_1, \tau_2, e_1, \check{\check{e}}_3)) \tag{28}$$

$$\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}_3 \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ \check{e}_3 \qquad (\gamma \notin \mathrm{FTN}(\tau_1, \tau_2, e_1, e_2)) \tag{29}$$

$$\mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{\check{e}} \;\longrightarrow\; \mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{30}$$

$$\check{\check{e}}\ \check{\check{e}}_2 \;\longrightarrow\; \check{\check{e}}'\ \check{\check{e}}_2 \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{31}$$

$$e\ \check{\check{e}} \;\longrightarrow\; e\ \check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{32}$$

$$\Lambda\alpha{:}\kappa.\check{\check{e}} \;\longrightarrow\; \Lambda\alpha{:}\kappa.\check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{33}$$

$$\check{\check{e}}\ \tau \;\longrightarrow\; \check{\check{e}}'\ \tau \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{34}$$

$$\mathrm{N}\gamma{:}\kappa{\approx}\tau.\check{\check{e}} \;\longrightarrow\; \mathrm{N}\gamma{:}\kappa{\approx}\tau.\check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{35}$$

$$\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{\check{e}} \;\longrightarrow\; \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{36}$$

$$\{\check{\check{e}} : \tau'\}^{\pm}_{\gamma\approx\tau} \;\longrightarrow\; \{\check{\check{e}}' : \tau'\}^{\pm}_{\gamma\approx\tau} \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{37}$$

$$\mathrm{tcase}\ \check{\check{e}} : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}_2\ \mathrm{else}\ \check{\check{e}}_3 \;\longrightarrow\; \mathrm{tcase}\ \check{\check{e}}' : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}_2\ \mathrm{else}\ \check{\check{e}}_3 \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{38}$$

$$\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}\ \mathrm{else}\ \check{\check{e}}_3 \;\longrightarrow\; \mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ \check{\check{e}}'\ \mathrm{else}\ \check{\check{e}}_3 \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{39}$$

$$\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ \check{\check{e}} \;\longrightarrow\; \mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ \check{\check{e}}' \qquad (\text{if } \check{\check{e}} \to \check{\check{e}}';\ \check{\check{e}} \not\equiv \check{e}) \tag{40}$$

$$\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.e \;\longrightarrow\; \mathrm{N}\gamma{:}\kappa{\approx}\tau.e \tag{41}$$

Figure 10: Reduction for applicative generation

[10]. They present a calculus that uses annotated brackets for marking abstraction boundaries during reduction. These are similar to the generalized coercions in λ_{N}. However, in their system abstraction brackets are not directed, i.e. it does not distinguish between sealing and unsealing. Instead, all directly nested brackets are collapsed on reduction and annotated with the sequence of 'principals' that own the corresponding abstractions. That appears to be slightly more complex, but avoids the need for the artefact of unsealed types, as well as η-expansion during reduction. The latter is advantegeous for proving a type erasure theorem. On the other hand, in their system the definition of type equivalence depends on an additional type assertion environment. This complicates the operational semantics, because the environment has to be maintained dynamically to cope with abstraction scoping. Furthermore, their calculus cannot express dynamic abstraction, but requires identifying a fixed set of abstractions statically, since technically, the reduction relation has to be extended for each occuring abstraction. Both these aspects make it less suited as a simple operational model for type abstraction.

The λ_{N}-calculus also reveals close similarities to Pierce and Sumii's cryptographic lambda calculus [21]: N-binders correspond to key generation and sealing/unsealing to encryption/decryption operations in that calculus. However, their type system is weaker in the sense that decryption may fail dynamically. They present an encoding of type abstracting polymorphism using ciphertext, but do not prove anything about it.

None of the mentioned work considers higher-order types and higher-order sealing, or applicative generativity.

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. In non-parametric settings it is necessary to capture generativity to achieve dynamic opacity and thereby encapsulation.

As a solution we proposed a calculus whose core feature is a syntactic treatment of dynamic generativity, using a variation of name generation as known from π-calculus and other systems. It relies on coercions as explicit transition markers for abstraction boundaries. By generalizing these coercions inductively over all types they can be used to express sealing. The calculus can be extended to higher-order abstract types and augmented with support for applicative generativity.

As future work, we would like to integrate aspects of $\lambda_{\mathrm{N}}^{\omega}$ with recent module theories [6], in order to get a full theory of modules with dynamic typing. For example, the language Alice ML that is currently being developed [2] provides so-called *packages* as a form of dynamics generalized to modules. A combined theory is needed to give a formal semantics for that feature.

Full representation independence or extensionality [18, 22] of abstract types in λ_{N} is a challenge to prove. There appears to be very little work on proof techniques for operational equivalence in non-parametric extensions of the λ-calculus. It is not clear how techniques like logical relations [18, 25, 23] can be applied in such a setting.

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, Jan. 1995.

[2] Alice Team. *The Alice System*. Programming System Lab, Universität des Saarlandes, `http://www.ps.un-sb.de/alice/`, 2003.

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

[4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. In G. Nelson, editor, *System Programming with Modula-3*, chapter 2, pages 11–66. Prentice Hall, 1991.

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

[6] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *30th Symposium on Principles of Programming Languages*, New Orleans, USA, Jan. 2003.

[7] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages*, San Francisco, USA, Jan. 1995.

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

[9] N. Glew. Type dispatch for named hierarchical types. In *International Conference on Functional Programming*, Paris, France, Oct. 1999.

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

[11] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, USA, Jan. 1995.

[12] X. Leroy. Applicative functors and fully transparent higher-order modules. In *22nd Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, USA, Jan. 1995. ACM.

[13] X. Leroy. *The Objective Caml System*. INRIA, 2003. `http://pauillac.inria.fr/ocaml/htmlman/`.

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

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

[16] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, 1979.

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *Definition of Standard ML (Revised)*. The MIT Press, 1997.

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

[19] J. Mitchell and G. 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.

[20] B. Pierce. *Types and Programming Languages*. The MIT Press, Feb. 2002.

[21] B. Pierce and E. Sumii. Relating cryptography and polymorphism. Technical report, July 2000. `http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/`.

[22] A. 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.

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

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

[25] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Beeze and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, Berlin, 1993.

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

[27] C. Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[28] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, Dec. 2001.

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

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

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

[32] S. Weirich. Type-safe cast. In *International Conference on Functional Programming*, pages 58–67, Montreal, Canada, Sept. 2000.

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

APPENDIX

A. PROOFS

A.1 Typing λ_N

The following basic lemmata are easy to show:

Lemma 1 (Type Equivalence Inversion).

1. *If $\gamma = \gamma'$, then $\gamma \equiv \gamma'$.*
2. *If $\tau_1 \to \tau_2 = \tau_1' \to \tau_2'$, then $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.*
3. *If $\forall\alpha.\tau = \forall\alpha.\tau'$, then $\tau = \tau'$.*
4. *If $\{\tau_1\}^-_{\gamma\approx\tau_2} = \{\tau_1'\}^-_{\gamma\approx\tau_2'}$, then $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.*

Proof. By relating type equivalence to a confluent parallel reduction, see e.g. [20]. □

Lemma 2 (Canonical Types). *For every type τ, exactly one of the following equivalences holds:*

- $\tau = \gamma$
- $\tau = \tau_1 \to \tau_2$
- $\tau = \forall\alpha.\tau_1$
- $\tau = \alpha^-$

where α^- is defined by the following grammar:

$$\alpha^- \ ::= \ \alpha \mid \{\alpha^-\}^-_{\gamma\approx\tau}$$

Proof. By relating type equivalence to a confluent parallel reduction and proving strong normalisation. □

Lemma 3 (Weakening). *If $\Gamma \vdash e : \tau$ and $\vdash \Gamma' : \diamond$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash e : \tau$.*

Proof. By induction on the derivation. □

Lemma 4 (Strengthening). *If $\Gamma \vdash e : \tau$ and $\Gamma' \subseteq \Gamma$ with $\vdash \Gamma' : \diamond$, and $\mathrm{FV}(e, \Gamma') \cup \mathrm{FTV}(e, \Gamma') \cup \mathrm{FTN}(e, \Gamma') \subseteq \mathrm{Dom}(\Gamma')$, then $\Gamma' \vdash e : \tau$.*

Proof. By induction on the derivation. □

Lemma 5 (Substitution).

1. *If $\tau_1 = \tau_2$ then $\tau_1[\alpha := \tau'] = \tau_2[\alpha := \tau']$.*
2. *If $\Gamma, \alpha \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$, then $\Gamma \vdash \tau[\alpha := \tau'] : \Omega$.*
3. *If $\Gamma, \alpha \vdash e : \tau$ and $\Gamma \vdash \tau' : \Omega$, then $\Gamma \vdash e[\alpha := \tau'] : \tau[\alpha := \tau']$.*
4. *If $\Gamma, x{:}\tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[x := e'] : \tau$.*

Proof. Each by induction on the original derivation. For the case of unsealed types in (2) and the (Seal) and (Unseal) cases in (3) observe that if $\vdash \Gamma, \alpha : \diamond$ and $\gamma\approx\tau \in \Gamma$ then $\alpha \notin \mathrm{FTV}(\tau)$, hence $\gamma\approx\tau[\alpha := \tau'] \in \Gamma$ is equivalent to $\gamma\approx\tau \in \Gamma$. □

Lemma 6 (Type inversion).

1. *If $\Gamma \vdash \alpha : \Omega$, then $\vdash \Gamma : \diamond$ and $\alpha \in \Gamma$.*
2. *If $\Gamma \vdash \gamma : \Omega$, then $\vdash \Gamma : \diamond$ and $\gamma\approx\tau \in \Gamma$.*
3. *If $\Gamma \vdash \tau_1 \to \tau_2 : \Omega$, then $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$.*
4. *If $\Gamma \vdash \forall\alpha.\tau : \Omega$, then $\Gamma, \alpha \vdash \tau : \Omega$.*
5. *If $\Gamma \vdash \{\tau_1\}^-_{\gamma\approx\tau_2} : \Omega$, then $\Gamma \vdash \tau_1 : \Omega$ and $\gamma\approx\tau_2 \in \Gamma$.*

Proof. By induction on the corresponding derivation. □

We formulate term inversion modulo type equivalence:

Lemma 7 (Inversion).

1. *If $\Gamma \vdash x : \tau$, then $\vdash \Gamma : \diamond$ and $x{:}\tau' \in \Gamma$ with $\tau' = \tau$.*
2. *If $\Gamma \vdash (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e) : \tau$, then $\tau = \tau_2 \to \tau_1$ and $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma, x_1{:}\tau_2 \to \tau_1, x_2{:}\tau_2 \vdash e : \tau_1$.*
3. *If $\Gamma \vdash e_1\ e_2 : \tau$, then there is a type τ' such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$.*
4. *If $\Gamma \vdash \Lambda\alpha.e : \tau$, then there is a type τ' such that $\tau = \forall\alpha.\tau'$ and $\Gamma, \alpha \vdash e : \tau'$.*
5. *If $\Gamma \vdash e\ \tau' : \tau$, then there is a type τ'' such that $\Gamma \vdash e : \forall\alpha.\tau''$ and $\Gamma \vdash \tau' : \Omega$ and $\tau = \tau''[\alpha := \tau']$.*
6. *If $\Gamma \vdash \mathrm{N}\gamma\approx\tau'.e : \tau$, then $\Gamma \vdash \tau' : \Omega$ and $\Gamma, \gamma\approx\tau' \vdash e : \tau$ and $\gamma \notin \mathrm{FTN}(\tau)$.*
7. *If $\Gamma \vdash \{e{:}\tau''\}^+_{\gamma\approx\tau'} : \tau$, then $\tau = \tau''$ and $\Gamma \vdash e : \{\tau''\}^-_{\gamma\approx\tau'}$ and $\gamma\approx\tau' \in \Gamma$.*
8. *If $\Gamma \vdash \{e{:}\tau''\}^-_{\gamma\approx\tau'} : \tau$, then $\tau = \{\tau''\}^-_{\gamma\approx\tau'}$ and $\Gamma \vdash e : \tau''$ and $\gamma\approx\tau' \in \Gamma$.*
9. *If $\Gamma \vdash (\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3) : \tau$, then $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma, x{:}\tau_2 \vdash e_2 : \tau$ and $\Gamma \vdash e_3 : \tau$.*

Proof. By induction on the corresponding derivation. □

Theorem 1 (Unique Types). *Whenever $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$ then $\tau = \tau'$.*

Proof. By induction on the derivation. □

Lemma 8 (Validity).

1. *If $\Gamma \vdash \tau : \Omega$, then $\vdash \Gamma : \diamond$.*
2. *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \Omega$.*

Proof. Each by induction on the original derivation using the previous lemmas. We treat the most interesting cases of (2):

- case $\mathrm{N}\gamma\approx\tau'.e' : \tau$
  1. by inverting (New), $\Gamma, \gamma\approx\tau' \vdash e' : \tau$ and $\Gamma \vdash \tau' : \Omega$ and $\gamma \notin \mathrm{FTN}(\tau)$
  2. by induction, $\Gamma, \gamma\approx\tau' \vdash \tau : \Omega$
  3. by strengthening, $\Gamma \vdash \tau : \Omega$
- case $\{e' : \tau\}^+_{\gamma\approx\tau'} : \tau$
  1. by inverting (Seal), $\Gamma \vdash e' : \{\tau\}^-_{\gamma\approx\tau'}$ and $\gamma\approx\tau' \in \Gamma$
  2. by induction, $\Gamma \vdash \{\tau\}^-_{\gamma\approx\tau'} : \Omega$
  3. by type inversion, $\Gamma \vdash \tau : \Omega$
- case $\{e' : \tau\}^-_{\gamma\approx\tau'} : \{\tau\}^-_{\gamma\approx\tau'}$
  1. by inverting (Unseal), $\Gamma \vdash e' : \tau$
  2. by induction, $\Gamma \vdash \tau : \Omega$
  3. by type formation, $\Gamma \vdash \{\tau\}^-_{\gamma\approx\tau'} : \Omega$

□

Theorem 2 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$.*

Proof. By induction on the generation of → using previous lemmas. Note that by validity, $\Gamma \vdash \tau : \Omega$. We treat the basic cases:

- case $e \equiv (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1)\ \hat{e}$ and $e' \equiv e_1[x_1 := (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1), x_2 := \hat{e}]$

1. by inverting (App), $\Gamma \vdash (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1) : \tau' \to \tau$ and $\Gamma \vdash \hat{\hat{e}} : \tau'$ for some τ'
2. by inverting (Fix), $\tau'{\to}\tau = \tau_2{\to}\tau_1$ and $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma, x_1{:}\tau_2{\to}\tau_1, x_2{:}\tau_2 \vdash e_1 : \tau$
3. by type equivalence inversion, $\tau = \tau_1$ and $\tau' = \tau_2$
4. by rule (Equiv), $\Gamma \vdash (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1) : \tau_2 \to \tau_1$ and $\Gamma \vdash \hat{\hat{e}} : \tau_2$
5. by substitution (4),
   $\Gamma \vdash e_1[x_1 := (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1), x_2 := \hat{\hat{e}}] : \tau_1$
6. by rule (Equiv),
   $\Gamma \vdash e_1[x_1 := (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1), x_2 := \hat{\hat{e}}] : \tau$

- case $e \equiv (\Lambda\alpha.e_1)\ \tau'$ and $e' \equiv e_1[\alpha := \tau']$
  1. by inverting (Inst), $\Gamma \vdash (\Lambda\alpha.e_1) : \forall\alpha.\tau_1$ and $\Gamma \vdash \tau' : \Omega$ and $\tau = \tau_1[\alpha := \tau']$ for some τ_1
  2. by inverting (Gen), $\Gamma, \alpha \vdash e_1 : \tau_1$
  3. by substitution (3), $\Gamma \vdash e_1[\alpha := \tau'] : \tau_1[\alpha := \tau']$
  4. by rule (Equiv), $\Gamma \vdash e_1[\alpha := \tau'] : \tau$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'}$ and $e' \equiv \hat{\hat{e}}$ (with $\tau_1 = \tau_2 = \gamma$)
  1. by inverting (Unseal), $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2$ and $\tau = \{\tau_2\}^{-}_{\gamma\approx\tau'}$
  2. by inverting (Seal), $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma\approx\tau'}$
  3. by type equivalence, $\{\tau_1\}^{-}_{\gamma\approx\tau'} = \tau' = \tau$
  4. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}} : \tau$
- case $e \equiv \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'}$ and $e' \equiv \hat{\hat{e}}$ (with $\tau_1 = \gamma \not\equiv \gamma'$)
  1. by inverting (Seal), $\tau = \tau_1$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma\approx\tau'}$
  2. by type equivalence, $\{\tau_1\}^{-}_{\gamma\approx\tau'} = \tau$
  3. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}} : \tau$
- case $e \equiv \{\hat{\hat{e}} : \tau_1\}^{-}_{\gamma\approx\tau'}$ and $e' \equiv \hat{\hat{e}}$ (with $\tau_1 = \gamma \not\equiv \gamma'$)
  Similarly.
- case $e \equiv \{\hat{\hat{e}} : \tau''\}^{+}_{\gamma\approx\tau'}$ (with $\tau'' = \tau_2{\to}\tau_1$)
  and $e' \equiv \mathrm{fix}\, x_1(x_2 : \tau_2) : \tau_1.\{\hat{\hat{e}}\ \{x : \tau_1\}^{-}_{\gamma\approx\tau'} : \tau_2\}^{+}_{\gamma\approx\tau'}$
  1. by inverting (Seal), $\tau = \tau''$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau''\}^{-}_{\gamma\approx\tau'}$ and $\gamma{\approx}\tau' \in \Gamma$
  2. by type inversion, $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$.
  3. by type formation, $\Gamma \vdash \{\tau_2\}^{-}_{\gamma\approx\tau'} \to \{\tau_1\}^{-}_{\gamma\approx\tau'} : \Omega$.
  4. by type equivalence, $\{\tau''\}^{-}_{\gamma\approx\tau'} = \{\tau_2\}^{-}_{\gamma\approx\tau'} \to \{\tau_1\}^{-}_{\gamma\approx\tau'}$
  5. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}} : \{\tau_2\}^{-}_{\gamma\approx\tau'} \to \{\tau_1\}^{-}_{\gamma\approx\tau'}$
  6. by weakening, $\Gamma' \vdash \hat{\hat{e}} : \{\tau_2\}^{-}_{\gamma\approx\tau'} \to \{\tau_1\}^{-}_{\gamma\approx\tau'}$ with $\Gamma' \equiv \Gamma, x_1 : \tau_2{\to}\tau_1, x_2 : \tau_2$ (well-formed by validity)
  7. by rule (Id), $\Gamma' \vdash x_2 : \tau_2$
  8. by rule (Unseal), $\Gamma' \vdash \{x_2 : \tau_2\}^{-}_{\gamma\approx\tau'} : \{\tau_2\}^{-}_{\gamma\approx\tau'}$
  9. by rule (App), $\Gamma' \vdash \hat{\hat{e}}\ \{x_2 : \tau_2\}^{-}_{\gamma\approx\tau'} : \{\tau_1\}^{-}_{\gamma\approx\tau'}$
  10. by rule (Seal), $\Gamma' \vdash \{\hat{\hat{e}}\ \{x_2 : \tau_2\}^{-}_{\gamma\approx\tau'} : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_1$
  11. by rule (Fix), $\Gamma \vdash e' : \tau_2 \to \tau_1$
  12. by rule (Equiv), $\Gamma \vdash e' : \tau$
- case $e \equiv \{\hat{\hat{e}} : \tau''\}^{-}_{\gamma\approx\tau'}$ (with $\tau'' = \tau_2{\to}\tau_1$)
  and $e' \equiv \mathrm{fix}\, x_1(x_2 : \{\tau_2\}^{-}_{\gamma\approx\tau'}) : \{\tau_1\}^{-}_{\gamma\approx\tau'}.\{\hat{\hat{e}}\ \{x : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'}$
  Similarly.
- case $e \equiv \{\hat{\hat{e}} : \tau''\}^{+}_{\gamma\approx\tau'}$ and $e' \equiv \Lambda\alpha.\{\hat{\hat{e}}\ \alpha : \tau'''\}^{+}_{\gamma\approx\tau'}$ (with $\tau'' = \forall\alpha.\tau'''$)
  1. by inverting (Seal), $\tau = \tau''$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau''\}^{-}_{\gamma\approx\tau'}$ and $\gamma{\approx}\tau' \in \Gamma$
  2. by type inversion, $\Gamma \vdash \tau''' : \Omega$.
  3. by type formation, $\Gamma \vdash \forall\alpha.\{\tau'''\}^{-}_{\gamma\approx\tau'} : \Omega$.
  4. by type equivalence, $\{\tau''\}^{-}_{\gamma\approx\tau'} = \forall\alpha.\{\tau'''\}^{-}_{\gamma\approx\tau'}$
  5. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}} : \forall\alpha.\{\tau'''\}^{-}_{\gamma\approx\tau'}$
  6. by weakening, $\Gamma' \vdash \hat{\hat{e}} : \forall\alpha.\{\tau'''\}^{-}_{\gamma\approx\tau'}$ with $\Gamma' \equiv \Gamma, \alpha$
  7. by rule (Inst), $\Gamma' \vdash \hat{\hat{e}}\ \alpha : \{\tau'''\}^{-}_{\gamma\approx\tau'}$
  8. by rule (Seal), $\Gamma' \vdash \{\hat{\hat{e}}\ \alpha : \tau'''\}^{+}_{\gamma\approx\tau'} : \tau'''$
  9. by rule (Gen), $\Gamma' \vdash e' : \forall\alpha.\tau'''$
  10. by rule (Equiv), $\Gamma \vdash e' : \tau$
- case $e \equiv \{\hat{\hat{e}} : \tau''\}^{-}_{\gamma\approx\tau'}$ and $e' \equiv \Lambda\alpha.\{\hat{\hat{e}}\ \alpha : \tau'''\}^{-}_{\gamma\approx\tau'}$ (with $\tau'' = \forall\alpha.\tau'''$)
  Similarly.
- case $e \equiv (\mathrm{N}\gamma{\approx}\tau'.\hat{e})\ e_2$ and $e' \equiv \mathrm{N}\gamma{\approx}\tau'.\hat{e}\ e_2$
  1. by inverting (App), $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'.\hat{e} : \tau_2 \to \tau$ and $\Gamma \vdash e_2 : \tau_2$ (*) for some τ_2
  2. by inverting (New), $\Gamma, \gamma{\approx}\tau' \vdash \hat{e} : \tau_2 \to \tau$ and $\gamma \notin \mathrm{FTN}(\tau_2 \to \tau)$
  3. by weakening (*), $\Gamma, \gamma{\approx}\tau' \vdash e_2 : \tau_2$
  4. by rule (App), $\Gamma, \gamma{\approx}\tau' \vdash \hat{e}\ e_2 : \tau$
  5. by rule (New), $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'.\hat{e}\ e_2 : \tau$
- case $e \equiv \hat{\hat{e}}\ (\mathrm{N}\gamma{\approx}\tau'.\hat{e})$ and $e' \equiv \mathrm{N}\gamma{\approx}\tau'.\hat{\hat{e}}\ \hat{e}$
  Similarly.
- case $e \equiv (\mathrm{N}\gamma{\approx}\tau'.\hat{e})\ \tau$ and $e' \equiv \mathrm{N}\gamma{\approx}\tau'.\hat{e}\ \tau$
  Similarly.
- case $e \equiv \{\mathrm{N}\gamma{\approx}\tau'''.\hat{e} : \tau''\}^{+}_{\gamma'\approx\tau'}$
  and $e' \equiv \mathrm{N}\gamma{\approx}\tau'''.\{\hat{e} : \tau''\}^{+}_{\gamma'\approx\tau'}$
  1. by inverting (Seal), $\tau'' = \tau$ and $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'''.\hat{e} : \{\tau''\}^{-}_{\gamma'\approx\tau'}$ and $\gamma'{\approx}\tau' \in \Gamma$
  2. by inverting (New), $\Gamma, \gamma{\approx}\tau''' \vdash \hat{e} : \{\tau''\}^{-}_{\gamma'\approx\tau'}$ and $\Gamma \vdash \tau''' : \Omega$ and $\gamma \notin \mathrm{FTN}(\{\tau''\}^{-}_{\gamma'\approx\tau'})$
  3. by rule (Seal), $\Gamma, \gamma{\approx}\tau''' \vdash \{\hat{e} : \tau''\}^{+}_{\gamma'\approx\tau'} : \tau''$
  4. by rule (New), $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'''.\{\hat{e} : \tau''\}^{+}_{\gamma'\approx\tau'} : \tau''$
  5. by rule (Equiv), $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'''.\{\hat{e} : \tau''\}^{+}_{\gamma'\approx\tau'} : \tau$
- case $e \equiv \{\mathrm{N}\gamma{\approx}\tau'''.\hat{e} : \tau''\}^{-}_{\gamma'\approx\tau'}$
  and $e' \equiv \mathrm{N}\gamma{\approx}\tau'''.\{\hat{e} : \tau''\}^{-}_{\gamma'\approx\tau'}$
  Similarly.
- case $e \equiv \mathrm{tcase}\ \mathrm{N}\gamma{\approx}\tau'.\hat{e}_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3
  and $e' \equiv \mathrm{N}\gamma{\approx}\tau'.\mathrm{tcase}\ \hat{e}_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3
  1. by inverting (Tcase), $\Gamma \vdash \mathrm{N}\gamma{\approx}\tau'.\hat{e}_1 : \tau_1$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma, x{:}\tau_1 \vdash e_2 : \tau$ and $\Gamma \vdash e_3 : \tau$
  2. by inverting (New), $\Gamma, \gamma{\approx}\tau' \vdash \hat{e}_1 : \tau_1$ and $\Gamma \vdash \tau' : \Omega$ and $\gamma \notin \mathrm{FTN}(\tau_1)$
  3. by weakening, $\Gamma, \gamma{\approx}\tau' \vdash \tau_2 : \Omega$ and $\Gamma, \gamma{\approx}\tau', x{:}\tau_1 \vdash e_2 : \tau$ and $\Gamma, \gamma{\approx}\tau' \vdash e_3 : \tau$
  4. by rule (Tcase), $\Gamma, \gamma{\approx}\tau' \vdash (\mathrm{tcase}\ \hat{e}_1 : \tau_1$ of $x : \tau_2$ then e_2 else $e_3) : \tau$
  5. by rule (New), $\Gamma \vdash e' : \tau$

- case $e \equiv \text{tcase}\ \hat{\hat{e}}_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3 and $e' \equiv e_2[x := \hat{\hat{e}}_1]$ (with $\tau_1 = \tau_2$)
  1. by inverting (TCASE), $\Gamma \vdash \hat{\hat{e}}_1 : \tau_1$ and $\Gamma, x{:}\tau_2 \vdash e_2 : \tau$ and $\Gamma \vdash \tau_2 : \Omega$
  2. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}}_1 : \tau_2$
  3. by substitution (4), $\Gamma \vdash e_2[x := \hat{\hat{e}}_1] : \tau$
- case $e \equiv \text{tcase}\ \hat{\hat{e}}_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3 and $e' \equiv e_3$ (with $\tau_1 \neq \tau_2$)
  1. by inverting (TCASE), $\Gamma \vdash e_3 : \tau$

□

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

LEMMA 9 (CANONICAL VALUES).
1. If $\Gamma \vdash \hat{\hat{e}} : \gamma$, then $\hat{\hat{e}} \equiv \{\hat{\hat{e}}' : \tau'\}^{+}_{\gamma\approx\tau}$ (with $\tau' = \gamma$).

2. If $\Gamma \vdash \hat{\hat{e}} : \tau_2 \rightarrow \tau_1$ then $\hat{\hat{e}} \equiv \text{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e$.

3. If $\Gamma \vdash \hat{\hat{e}} : \forall\alpha.\tau$ then $\hat{\hat{e}} \equiv \Lambda\alpha.e$.

PROOF. By inspection of the cases for $\hat{\hat{e}}$. □

N is a binder for type variables. Hence, in order to prove progress by induction, a slightly stronger induction hypothesis is necessary:

THEOREM 3 (PROGRESS). *Let Γ be an environment containing only type assertions (i.e. $\Gamma \equiv \gamma_1{\approx}\tau_1, \cdots, \gamma_n{\approx}\tau_n$). If $\Gamma \vdash e : \tau$, then either $e \equiv \hat{e}$ for some result $\hat{e}$, or $e \rightarrow e'$ for some expression e'. Moreover, in the latter case, there is exactly one applicable reduction rule, i.e. reduction is deterministic.*

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

- case $e \equiv \text{N}\gamma{\approx}\tau_1.e_1$ with $e_1 \not\equiv \hat{e}_1$
  1. by inverting (NEW), $\Gamma, \gamma{\approx}\tau_1 \vdash e_1 : \tau$
  2. by induction, $e_1 \rightarrow e_1'$ (using $\Gamma' \equiv \Gamma, \gamma{\approx}\tau_1$)
  3. by reduction rule (17), $\text{N}\gamma{\approx}\tau_1.e_1 \rightarrow \text{N}\gamma{\approx}\tau_1.e_1'$
- case $e \equiv \{e_1 : \tau_1\}^{+}_{\gamma\approx\tau'}$ by canonical types, we have the following subcases:
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma$
    1. by definition, e is a result
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma'$ (with $\gamma \not\equiv \gamma'$)
    1. by reduction rule (4), $e \rightarrow \hat{\hat{e}}_1$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \tau_2{\rightarrow}\tau_3$
    1. by reduction rule (5), $e \rightarrow \text{fix}\, x_1(x_2 : \tau_2) : \tau_3.\{\hat{\hat{e}}_1\ \{x : \tau_3\}^{-}_{\gamma\approx\tau'} : \tau_2\}^{+}_{\gamma\approx\tau'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \forall\alpha.\tau_2$
    1. by reduction rule (6), $e \rightarrow \Lambda\alpha.\{\hat{\hat{e}}_1\ \alpha : \tau_2\}^{+}_{\gamma\approx\tau'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \alpha^{-}$ cannot occur since e_1 is closed wrt. type variables.
  - subcase $e_1 \equiv \text{N}\gamma'{\approx}\tau_2.\hat{e}_1$
    1. by reduction rule (12), $e \rightarrow \text{N}\gamma'{\approx}\tau_2.\{\hat{e}_1 : \tau_1\}^{+}_{\gamma\approx\tau'}$
  - subcase $e_1 \not\equiv \hat{e}_1$
    1. by inverting (SEAL), $\Gamma \vdash e_1 : \{\tau_1\}^{-}_{\gamma}$
    2. by induction, $e_1 \rightarrow e_1'$
    3. by reduction rule (18), $e \rightarrow \{e_1' : \tau_1\}^{+}_{\gamma\approx\tau'}$
- case $e \equiv \{e_1 : \tau_1\}^{-}_{\gamma\approx\tau'}$ by canonical types, we have the following subcases:
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma$
    1. by inverting (UNSEAL), $\Gamma \vdash \hat{\hat{e}}_1 : \tau_1$
    2. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}}_1 : \gamma$
    3. as canonical value, $\hat{\hat{e}}_1 \equiv \{\hat{\hat{e}}_1' : \tau_2\}^{+}_{\gamma\approx\tau'}$ with $\tau_2 = \gamma$
    4. by reduction rule (3), $e \rightarrow \hat{\hat{e}}_1'$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma'$ (with $\gamma \not\equiv \gamma'$)
    1. by reduction rule (4), $e \rightarrow \hat{\hat{e}}_1$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \tau_2{\rightarrow}\tau_3$
    1. by reduction rule (5), $e \rightarrow \text{fix}\, x_1(x_2 : \{\tau_2\}^{-}_{\gamma\approx\tau'}) : \{\tau_3\}^{-}_{\gamma\approx\tau'}.\{\hat{\hat{e}}_1\ \{x : \tau_3\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \forall\alpha.\tau_2$
    1. by reduction rule (6), $e \rightarrow \Lambda\alpha.\{\hat{\hat{e}}_1\ \alpha : \tau_2\}^{-}_{\gamma\approx\tau'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \alpha^{-}$ cannot occur since e_1 is closed wrt. type variables.
  - subcase $e_1 \equiv \text{N}\gamma'{\approx}\tau_2.\hat{e}_1$
    1. by reduction rule (12), $e \rightarrow \text{N}\gamma'{\approx}\tau_2.\{\hat{e}_1 : \tau_1\}^{-}_{\gamma\approx\tau'}$
  - subcase $e_1 \not\equiv \hat{e}_1$
    1. by inverting (UNSEAL), $\Gamma \vdash e_1 : \tau_1$
    2. by induction, $e_1 \rightarrow e_1'$
    3. by reduction rule (18), $e \rightarrow \{e_1' : \tau_1\}^{-}_{\gamma\approx\tau'}$
- case $e \equiv \text{tcase}\ e_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \tau_2$
    1. by reduction rule (7), $e \rightarrow e_2[x := \hat{\hat{e}}_1]$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 \neq \tau_2$
    1. by reduction rule (8), $e \rightarrow e_3$
  - subcase $e_1 \equiv \text{N}\gamma{\approx}\tau'.\hat{e}_1$
    1. by reduction rule (13), $e \rightarrow \text{N}\gamma{\approx}\tau'.\text{tcase}\ \hat{e}_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3
  - subcase $e_1 \not\equiv \hat{e}_1$
    1. by inverting (TCASE), $\Gamma \vdash e_1 : \tau_1$
    2. by induction, $e_1 \rightarrow e_1'$
    3. by reduction rule (19), $e \rightarrow \text{tcase N}\gamma{\approx}\tau'.e_1' : \tau_1$ of $x : \tau_2$ then e_2 else e_3

Deterministic reduction follows from the fact that all cases are disjoint, and in each case no other reduction rule is applicable. □

COROLLARY 4 (PROGRESS FOR CLOSED EXPRESSIONS). *If $\cdot \vdash e : \tau$, then either $e \equiv \hat{e}$ for some result $\hat{e}$, or there is an expression e' such that $e \rightarrow e'$.*

A.2 Opacity

LEMMA 10 (RESULT AND VALUE SUBSTITUTION). *Let σ be an arbitrary substitution.*

1. $\hat{e}\sigma \equiv \hat{e}'$

2. $\hat{\hat{e}}\sigma \equiv \hat{\hat{e}}'$

PROOF. By trivial induction on the structure of the original expression. □

LEMMA 11 (NAME SUBSTITUTION). *If $\sigma = [\alpha := \gamma]$ and $\gamma \notin \text{FTN}(\tau_1, \tau_2)$ then*

1. $\tau_1\sigma = \tau_2 \quad \Rightarrow \quad \tau_2 \equiv \tau_3\sigma$
(for some τ_3 with $\gamma \notin \mathrm{FTN}(\tau_3)$)

2. $\tau_1\sigma \equiv \tau_2\sigma \quad \Leftrightarrow \quad \tau_1 \equiv \tau_2$

3. $\tau_1\sigma = \tau_2\sigma \quad \Leftrightarrow \quad \tau_1 = \tau_2$

Proof.

1. By induction on the derivation.
2. By induction on the structure of τ_1.
3. By induction on the derivation, using (2) in the reflexive case and (1) in the associative case of the "⇒" direction.

□

Theorem 5 (Opacity). *Let Γ be an environment containing only type assertions (i.e. $\Gamma \equiv \gamma'_1{\approx}\tau_1, \cdots, \gamma'_n{\approx}\tau_n$) and e an expression with $\Gamma, \alpha, x{:}\alpha \vdash e : \tau$. Assume a set of values $\hat{\hat{e}}_i$ $(i = 1, \ldots, n)$ such that $\gamma_i{\approx}\tau_i \vdash \hat{\hat{e}}_i : \gamma_i$ with $\gamma_i \notin \mathrm{Dom}(\Gamma)$. Let $\sigma_i = [\alpha := \gamma_i, x := \hat{\hat{e}}_i]$. If $e\sigma_1 \not\equiv \hat{\hat{e}}$ then there is an e' with $\Gamma, \alpha, x{:}\alpha \vdash e' : \tau$ such that*

$$e\sigma_i \rightarrow e'\sigma_i$$

for all σ_i.

Proof. By canonical values, $\hat{\hat{e}}_i \equiv \{\hat{\hat{e}}'_i : \tau'_i\}^+_{\gamma_i}$ (with $\tau'_i = \gamma_i$). By progress, $e\sigma_1 \rightarrow e''$ for some e''. We can hence prove the conjecture by induction on the derivation of $\rightarrow$, using lemmata 10–11.

- case (1): There are 2 possibilities:
  - subcase $e \equiv (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1)\ \hat{\hat{e}}$
    For all i we have
    $$\begin{array}{rcl} e\sigma_i & \equiv & (\mathrm{fix}\, x_1(x_2{:}\tau_2\sigma_i){:}\tau_1\sigma_i.e_1\sigma_i)\ (\hat{\hat{e}}\sigma_i) \\ & \rightarrow & (e_1\sigma_i)[x_1 := \mathrm{fix}\, x_1(x_2{:}\tau_2\sigma_i){:}\tau_1\sigma_i.e_1\sigma_i, x_2 := \hat{\hat{e}}\sigma_i] \\ & \equiv & (e_1\sigma_i)[x_1 := (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1)\sigma_i, x_2 := \hat{\hat{e}}\sigma_i] \\ & \equiv & (e_1[x_1 := \mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1, x_2 := \hat{\hat{e}}])\sigma_i \end{array}$$
    Hence $e' \equiv e_1[x_1 := \mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1, x_2 := \hat{\hat{e}}]$, which is well-typed.
  - subcase $e \equiv (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e_1)\ x$
    The proof proceeds likewise, with $\hat{\hat{e}}\sigma_i$ replaced by $x\sigma_i$.
- case (2): Similarly.
- case (3): Due to the assumptions, $\gamma \not\equiv \gamma_i$ for any i. Hence, there are 2 possibilities:
  - subcase $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^+_{\gamma\approx\tau'} : \tau_2\}^-_{\gamma\approx\tau'}$
    By substitution (1), $\tau_1\sigma_i = \tau_2\sigma_i$. Hence for all i we have
    $$e\sigma_i \ \equiv\ \{\{\hat{\hat{e}}\sigma_i : \tau_1\sigma_i\}^+_{\gamma\approx\tau'\sigma_i} : \tau_2\sigma_i\}^-_{\gamma\approx\tau'\sigma_i} \ \rightarrow\ \hat{\hat{e}}\sigma_i$$
    Hence $e' \equiv \hat{\hat{e}}$, which is well-typed by inversion.
  - subcase $e \equiv \{\{x : \tau_1\}^+_{\gamma\approx\tau'} : \tau_2\}^-_{\gamma\approx\tau'}$
    Likewise.
- case (4): Due to the assumptions, $\gamma' \not\equiv \gamma_i$ for any i. Hence, by inversion there is only one possibility:
  - $e \equiv \{\hat{\hat{e}} : \tau''\}^\pm_{\gamma\approx\tau'}$ (with $\tau'' = \gamma'$)
    By substitution (1), $\tau''\sigma_i = \gamma'\sigma_i = \gamma' = \tau''$, hence for all i we have
    $$e\sigma_i \ \equiv\ \{\hat{\hat{e}}\sigma_i : \tau''\}^\pm_{\gamma\approx\tau'\sigma_i} \ \rightarrow\ \hat{\hat{e}}\sigma_i$$
    Hence $e' \equiv \hat{\hat{e}}$, which is well-typed by inversion.
- cases (5)–(6): Similarly.
- case (7): By lemma 11, $\tau_1\sigma_i = \tau_2\sigma_i$, hence there are only 2 possibilities:
  - subcase $e \equiv \mathrm{tcase}\ \hat{\hat{e}} : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3$
    $$\begin{array}{rcl} e\sigma_i & \equiv & \mathrm{tcase}\ \hat{\hat{e}}\sigma_i : \tau_1\sigma_i\ \mathrm{of}\ x : \tau_2\sigma_i\ \mathrm{then}\ e_2\sigma_i\ \mathrm{else}\ e_3\sigma_i \\ & \rightarrow & (e_2\sigma_i)[x := \hat{\hat{e}}\sigma_i] \\ & \equiv & (e_2[x := \hat{\hat{e}}])\sigma_i \end{array}$$
  - subcase $e \equiv \mathrm{tcase}\ x : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3$
    Likewise.
- case (8): Similarly.
- case (9): There are only 2 possibilities:
  - subcase $e \equiv (\mathrm{N}\gamma{\approx}\tau'.\hat{\hat{e}})\ e'$
    $$\begin{array}{rcl} e\sigma_i & \equiv & (\mathrm{N}\gamma{\approx}\tau'\sigma_i.\hat{\hat{e}}\sigma_i)\ (e'\sigma_i) \\ & \rightarrow & \mathrm{N}\gamma{\approx}\tau'\sigma_i.(\hat{\hat{e}}\sigma_i)\ (e'\sigma_i) \\ & \equiv & (\mathrm{N}\gamma{\approx}\tau'.\hat{\hat{e}}\ e')\sigma_i \end{array}$$
  - subcase $e \equiv (\mathrm{N}\gamma{\approx}\tau'.x)\ e'$
    Likewise.
- cases (10)–(13): Similarly.
- case (14): We have $e\sigma_1 \equiv (e_1\sigma_1)\ (e_2\sigma_1)$ and $e_1\sigma_1 \rightarrow e''_1$ for some e''_1. By induction, $e_1\sigma_i \rightarrow e'_1\sigma_i$ for all σ_i. So
  $$e\sigma_i \ \equiv\ (e_1\sigma_i)\ (e_2\sigma_i) \ \rightarrow\ (e'_1\sigma_i)\ (e_2\sigma_i) \ \equiv\ (e'_1\ e_2)\sigma_i$$
- cases (15)–(19): Similarly, using an extended environment $\Gamma' \equiv \Gamma, \gamma{\approx}\tau'$ for induction in case (17).

□

A.3 Typing $\lambda^\omega_\mathrm{N}$

Figure 11 shows the complete syntax of $\lambda^\omega_\mathrm{N}$, figure 12 gives its typing rules and figure 13 the respective type equivalence relation. Most lemmas from section A.1 still hold for $\lambda^\omega_\mathrm{N}$, if Ω is generalized to arbitrary κ at the right places. We restate only the interesting ones:

Lemma 12 (Canonical Types). *For every type $\tau : \Omega$, exactly one of the following equivalences holds:*

- $\tau = \gamma\ \vec{\tau}$
- $\tau = \tau_1{\rightarrow}\tau_2$
- $\tau = \forall\alpha.\tau_1$
- $\tau = \alpha^-$

where α^- is defined by the following grammar:

$$\alpha^- \ ::=\ \alpha \mid \{\alpha^-\}^-_{\gamma\approx\tau}$$

Proof. By relating type equivalence to a confluent parallel reduction and proving strong normalisation. □

Lemma 13 (Type inversion).

1. *If $\Gamma \vdash \alpha : \kappa$, then $\vdash \Gamma : \diamond$ and $\alpha{:}\kappa \in \Gamma$.*

2. *If $\Gamma \vdash \gamma : \kappa$, then $\vdash \Gamma : \diamond$ and $\gamma{\approx}\tau \in \Gamma$ and $\Gamma \vdash \tau : \kappa$.*

3. *If $\Gamma \vdash \tau_1 \rightarrow \tau_2 : \Omega$, then $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$.*

4. *If $\Gamma \vdash \forall\alpha{:}\kappa.\tau : \kappa$, then $\kappa \equiv \Omega$ and $\Gamma, \alpha{:}\kappa \vdash \tau : \Omega$.*

5. *If $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa$, then $\kappa \equiv \kappa_1{\rightarrow}\kappa_2$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau : \kappa_2$.*

6. *If $\Gamma \vdash \tau_1\ \tau_2 : \kappa_1$, then $\Gamma \vdash \tau_1 : \kappa_2{\rightarrow}\kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2$.*

7. If $\Gamma \vdash \{\tau_1\}^{-}_{\gamma\approx\tau_2} : \kappa$*, then* $\Gamma \vdash \tau_1 : \kappa$ *and* $\gamma{\approx}\tau_2 \in \Gamma$*.*

PROOF. By induction on the corresponding derivation. □

We need the following lemma to prove preservation for higher-order coercions:

LEMMA 14 (UNSEALING).
1. If $\gamma \notin \mathrm{FTN}(\tau')$*, then* $\{\tau'\}^{-}_{\gamma\approx\tau} = \tau'$*.*
2. If $\gamma \notin \mathrm{FTN}(\tau')$*, then* $\{\{\{\tau''\}^{-}_{\gamma'\approx\tau'}\}^{-}_{\gamma\approx\tau}\}^{-}_{\gamma'\approx\tau'} = \{\{\tau''\}^{-}_{\gamma\approx\tau}\}^{-}_{\gamma'\approx\tau'}$*.*

PROOF. Each by easy induction on the structure of τ. □

LEMMA 15 (NON-CIRCULARITY). *Let* $\vdash \Gamma : \diamond$ *with* $\gamma{\approx}\tau \in \Gamma$ *and* $\gamma'{\approx}\tau' \in \Gamma$*. If* $\gamma' \in \mathrm{FTN}(\tau)$*, then* $\gamma \notin \mathrm{FTN}(\tau')$*.*

PROOF. By induction on the derivation of $\vdash \Gamma : \diamond$. □

THEOREM 6 ($\lambda^{\omega}_{\mathrm{N}}$ PRESERVATION). *If* $\Gamma \vdash e : \tau_0$ *and* $e \rightarrow e'$*, then* $\Gamma \vdash e' : \tau_0$*.*

PROOF. By induction on the generation of $\rightarrow$. We treat the new cases (4a–4e):

- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{+}_{\gamma\approx\tau}$ and $e' \equiv \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'}$ (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma \notin \mathrm{FTN}(\vec{\tau})$)
  1. by inverting (SEAL), $\tau_0 = \tau_2$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \{\tau_2\}^{-}_{\gamma}$
  2. unsealing (1), $\{\tau_2\}^{-}_{\gamma} = \tau_2$
  3. by rule (EQUIV), $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \tau_0$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau}$ and $e' \equiv \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'}$ (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma \notin \mathrm{FTN}(\vec{\tau})$)
  1. by inverting (UNSEAL), $\tau_0 = \{\tau_2\}^{-}_{\gamma}$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \tau_2$
  2. unsealing (1), $\{\tau_2\}^{-}_{\gamma} = \tau_2$
  3. by rule (EQUIV), $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \tau_0$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{+}_{\gamma\approx\tau}$
  and $e' \equiv \{\{\{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \vec{\tau}\}^{+}_{\gamma'}$
  (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma' \notin \mathrm{FTN}(\tau)$)
  1. by inverting (SEAL), $\tau_0 = \tau_2$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \{\tau_2\}^{-}_{\gamma}$
  2. by inverting (SEAL), $\tau_1 = \{\tau_2\}^{-}_{\gamma}$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma'}$
  3. by type equivalence, $\{\tau_1\}^{-}_{\gamma'} = \{\{\tau_2\}^{-}_{\gamma}\}^{-}_{\gamma'} = \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  4. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  5. by rule (UNSEAL), $\Gamma \vdash e''' : \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma}$ with $e''' \equiv \{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  6. by unsealing (2), $\{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\}^{-}_{\gamma}\ \{\{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\}^{-}_{\gamma}\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  7. by rule (EQUIV), $\Gamma \vdash e''' : \{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  8. by rule (SEAL), $\Gamma \vdash e'' : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}$ with $e'' \equiv \{e''' : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma}$
  9. by rule (EQUIV), $\Gamma \vdash e'' : \{\gamma'\ \vec{\tau}\}^{-}_{\gamma'}$
  10. by rule (SEAL), $\Gamma \vdash e' : \gamma'\ \vec{\tau}$
  11. by rule (EQUIV), $\Gamma \vdash e' : \tau_0$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau}$
  and $e' \equiv \{\{\{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \{\vec{\tau}\}^{-}_{\gamma}\}^{+}_{\gamma'}$
  (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma' \notin \mathrm{FTN}(\tau)$)
  1. by inverting (UNSEAL), $\tau_0 = \{\tau_2\}^{-}_{\gamma}$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \tau_2$
  2. by inverting (SEAL), $\tau_1 = \tau_2$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma'}$
  3. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}$
  4. by rule (UNSEAL), $\Gamma \vdash e''' : \{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$ with $e''' \equiv \{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  5. by unsealing (2), $\{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\}^{-}_{\gamma}\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\}^{-}_{\gamma}\ \{\{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma} = \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  6. by rule (EQUIV), $\Gamma \vdash e''' : \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  7. by rule (SEAL), $\Gamma \vdash e'' : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}$ with $e'' \equiv \{e''' : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}\}^{+}_{\gamma}$
  8. by rule (EQUIV), $\Gamma \vdash e'' : \{\gamma'\ \{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  9. by rule (SEAL), $\Gamma \vdash e' : \gamma'\ \{\vec{\tau}\}^{-}_{\gamma}$
  10. by rule (EQUIV), $\Gamma \vdash e' : \tau_0$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{+}_{\gamma\approx\tau}$
  and $e' \equiv \{\{\{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{+}_{\gamma'} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma} : \gamma'\ \vec{\tau}\}^{+}_{\gamma'}$
  (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma' \in \mathrm{FTN}(\tau)$)
  1. by validity, $\vdash \Gamma : \diamond$
  2. by non-circularity, $\gamma \notin \mathrm{FTN}(\tau')$
  3. by inverting (SEAL), $\tau_0 = \tau_2$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \{\tau_2\}^{-}_{\gamma}$
  4. by inverting (SEAL), $\tau_1 = \{\tau_2\}^{-}_{\gamma}$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma'}$
  5. by unsealing (2+1), $\{\tau_1\}^{-}_{\gamma'} = \{\{\tau_2\}^{-}_{\gamma}\}^{-}_{\gamma'} = \{\{\{\tau_2\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} = \{\{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} = \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  6. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}} : \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  7. by rule (SEAL), $\Gamma \vdash e''' : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma'}$ with $e''' \equiv \{\hat{\hat{e}} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{+}_{\gamma'}$
  8. by rule (EQUIV), $\Gamma \vdash e''' : \{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  9. by rule (SEAL), $\Gamma \vdash e'' : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}$ with $e'' \equiv \{e''' : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{+}_{\gamma}$
  10. by rule (EQUIV), $\Gamma \vdash e'' : \{\gamma'\ \vec{\tau}\}^{-}_{\gamma'}$
  11. by rule (SEAL), $\Gamma \vdash e' : \gamma'\ \vec{\tau}$
  12. by rule (EQUIV), $\Gamma \vdash e' : \tau_0$
- case $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau}$
  and $e' \equiv \{\{\{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma} : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} : \gamma'\ \{\vec{\tau}\}^{-}_{\gamma}\}^{+}_{\gamma'}$
  (with $\tau_1 = \gamma'\ \vec{\tau}'$, $\tau_2 = \gamma\ \vec{\tau}$ and $\gamma' \in \mathrm{FTN}(\tau)$)
  1. by validity, $\vdash \Gamma : \diamond$
  2. by non-circularity, $\gamma \notin \mathrm{FTN}(\tau')$
  3. by inverting (UNSEAL), $\tau_0 = \{\tau_2\}^{-}_{\gamma}$ and $\Gamma \vdash \{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'} : \tau_2$
  4. by inverting (SEAL), $\tau_1 = \tau_2$ and $\Gamma \vdash \hat{\hat{e}} : \{\tau_1\}^{-}_{\gamma'}$
  5. by rule (EQUIV), $\Gamma \vdash \hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}$
  6. by rule (UNSEAL), $\Gamma \vdash e''' : \{\tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$ with $e''' \equiv \{\hat{\hat{e}} : \tau'\ \{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  7. by rule (EQUIV), $\Gamma \vdash e''' : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}$
  8. by rule (UNSEAL), $\Gamma \vdash e'' : \{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'}$ with $e'' \equiv \{e''' : \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'}$
  9. by unsealing (2+1), $\{\tau'\ \{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} = \{\tau'\}^{-}_{\gamma'}\ \{\{\{\vec{\tau}\}^{-}_{\gamma'}\}^{-}_{\gamma}\}^{-}_{\gamma'} = \{\tau'\}^{-}_{\gamma'}\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'} = \tau'\ \{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'}$

10. by rule (Equiv), $\Gamma \vdash e'' : \tau' \; \{\{\vec{\tau}\}^-_\gamma\}^-_{\gamma'}$
11. by rule (Seal), $\Gamma \vdash e' : \gamma' \; \{\vec{\tau}\}^-_\gamma$
12. by rule (Equiv), $\Gamma \vdash e' : \tau_0$

□

Lemma 16 (Canonical values). *For any value $\hat{\hat{e}}$ and environment Γ :*

1. *If $\Gamma \vdash \hat{\hat{e}} : \gamma \; \vec{\tau}$, then $\hat{\hat{e}} \equiv \{\hat{\hat{e}}' : \tau'\}^+_{\gamma\approx\tau}$ (with $\tau' = \gamma \; \vec{\tau}$).*
2. *If $\Gamma \vdash \hat{\hat{e}} : \tau_2 \rightarrow \tau_1$ then $\hat{\hat{e}} \equiv \mathrm{fix}\, x_1(x_2{:}\tau_2'){:}\tau_1'.e$.*
3. *If $\Gamma \vdash \hat{\hat{e}} : \forall\alpha.\tau$ then $\hat{\hat{e}} \equiv \Lambda\alpha.e$.*

Proof. By induction of the cases for $\hat{\hat{e}}$ and induction on the derivations. □

Theorem 7 ($\lambda^\omega_{\mathrm{N}}$ Progress). *Let Γ be an environment containing only type assertions (i.e. $\Gamma \equiv \gamma_1{\approx}\tau_1, \cdots, \gamma_n{\approx}\tau_n$). If $\Gamma \vdash e : \tau_0$, then either $e \equiv \hat{e}$ for some result $\hat{e}$, or $e \rightarrow e'$ for some expression e'. Moreover, in the latter case, there is exactly one applicable reduction rule.*

Proof. By easy induction on the typing derivations. We treat just the cases that are different from λ_{N}:

- case $e \equiv \{e_1 : \tau_1\}^+_{\gamma\approx\tau}$
  by canonical types, we have the following subcases:
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma \; \vec{\tau}$
    1. by definition, e is a result
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma' \; \vec{\tau}$ (with $\gamma \not\equiv \gamma'$)
    1. by inverting (Seal), $\Gamma \vdash \hat{\hat{e}}_1 : \{\tau_1\}^-_{\gamma\approx\tau}$
    2. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}}_1 : \gamma' \; \{\vec{\tau}\}^-_{\gamma\approx\tau}$
    3. as canonical value, $\hat{\hat{e}}_1 \equiv \{\hat{\hat{e}}_1' : \tau_1'\}^+_{\gamma'\approx\tau'}$ with $\tau_1' = \gamma' \; \vec{\tau}'$
    4. if $\exists\vec{\tau}' = \vec{\tau}.\gamma \notin \mathrm{FTN}(\vec{\tau}')$, then by reduction rule (4a), $e \rightarrow \{\hat{\hat{e}}_1' : \tau_1'\}^+_{\gamma'}$
    5. else if $\exists\tau'' = \tau.\gamma' \notin \mathrm{FTN}(\tau'')$, then by reduction rule (4b),
       $e \rightarrow \{\{\{\hat{\hat{e}} : \tau'\{\{\vec{\tau}\}^-_\gamma\}^-_{\gamma'}\}^-_\gamma : \tau'\{\vec{\tau}\}^-_{\gamma'}\}^+_\gamma : \gamma'\vec{\tau}\}^+_{\gamma'}$
    6. else by reduction rule (4d),
       $e \rightarrow \{\{\{\hat{\hat{e}} : \tau'\{\{\vec{\tau}\}^-_{\gamma'}\}^-_\gamma\}^+_{\gamma'} : \tau'\{\vec{\tau}\}^-_{\gamma'}\}^+_\gamma : \gamma'\vec{\tau}\}^+_{\gamma'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \tau_2{\rightarrow}\tau_3$
    1. by reduction rule (5), $e \rightarrow \mathrm{fix}\, x_1(x_2 : \tau_2) : \tau_3.\{\hat{\hat{e}}_1 \; \{x : \tau_3\}^-_\gamma : \tau_2\}^+_\gamma$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \forall\alpha.\tau_2$
    1. by reduction rule (6), $e \rightarrow \Lambda\alpha.\{\hat{\hat{e}}_1 \; \alpha : \tau_2\}^+_\gamma$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \alpha^-$
    cannot appear since e_1 is closed wrt. type variables.
  - subcase $e_1 \equiv \mathrm{N}\gamma'{\approx}\tau_2.\hat{e}_1$
    1. by reduction rule (12), $e \rightarrow \mathrm{N}\gamma'{\approx}\tau_2.\{\hat{e}_1 : \tau_1\}^+_\gamma$
  - subcase $e_1 \not\equiv \hat{e}_1$
    1. by inverting (Seal), $\Gamma \vdash e_1 : \{\tau_1\}^-_\gamma$
    2. by induction, $e_1 \rightarrow e_1'$
    3. by reduction rule (18), $e \rightarrow \{e_1' : \tau_1\}^+_\gamma$
- case $e \equiv \{e_1 : \tau_1\}^-_{\gamma\approx\tau}$
  by canonical types, we have the following subcases:
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma \; \vec{\tau}$
    1. by inverting (Unseal), $\Gamma \vdash \hat{\hat{e}}_1 : \tau_1$
    2. as canonical value, $\hat{\hat{e}}_1 \equiv \{\hat{\hat{e}}_1' : \tau_2\}^+_{\gamma\approx\tau}$
    3. by inverting (Seal), $\tau_2 = \gamma$
    4. by reduction rule (3), $e \rightarrow \hat{\hat{e}}_1'$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \gamma' \; \vec{\tau}$ (with $\gamma \not\equiv \gamma'$)
    1. by inverting (Unseal), $\Gamma \vdash \hat{\hat{e}}_1 : \tau_1$
    2. by rule (Equiv), $\Gamma \vdash \hat{\hat{e}}_1 : \gamma' \; \vec{\tau}$
    3. as canonical value, $\hat{\hat{e}}_1 \equiv \{\hat{\hat{e}}_1' : \tau_1'\}^+_{\gamma'\approx\tau'}$ with $\tau_1' = \gamma' \; \vec{\tau}'$
    4. if $\exists\vec{\tau}' = \vec{\tau}.\gamma \notin \mathrm{FTN}(\vec{\tau}')$, then by reduction rule (4a), $e \rightarrow \{\hat{\hat{e}}_1' : \tau_1'\}^-_{\gamma'}$
    5. else if $\exists\tau'' = \tau.\gamma' \notin \mathrm{FTN}(\tau'')$, then by reduction rule (4c),
       $e \rightarrow \{\{\{\hat{\hat{e}} : \tau'\{\vec{\tau}\}^-_{\gamma'}\}^-_\gamma : \tau'\{\{\vec{\tau}\}^-_\gamma\}^-_{\gamma'}\}^+_\gamma : \gamma'\{\vec{\tau}\}^-_\gamma\}^+_{\gamma'}$
    6. else by reduction rule (4e),
       $e \rightarrow \{\{\{\hat{\hat{e}} : \tau'\{\vec{\tau}\}^-_{\gamma'}\}^-_\gamma : \tau'\{\{\vec{\tau}\}^-_{\gamma'}\}^-_\gamma\}^-_{\gamma'} : \gamma'\{\vec{\tau}\}^-_\gamma\}^+_{\gamma'}$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \tau_2{\rightarrow}\tau_3$
    1. by reduction rule (5), $e \rightarrow \mathrm{fix}\, x_1(x_2 : \{\tau_2\}^-_\gamma) : \{\tau_3\}^-_\gamma.\{\hat{\hat{e}}_1 \; \{x : \tau_3\}^+_\gamma : \tau_2\}^-_\gamma$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \forall\alpha.\tau_2$
    1. by reduction rule (6), $e \rightarrow \Lambda\alpha.\{\hat{\hat{e}}_1 \; \alpha : \tau_2\}^-_\gamma$
  - subcase $e_1 \equiv \hat{\hat{e}}_1$ and $\tau_1 = \alpha^-$
    cannot appear since e_1 is closed wrt. type variables.
  - subcase $e_1 \equiv \mathrm{N}\gamma'{\approx}\tau_2.\hat{e}_1$
    1. by reduction rule (12), $e \rightarrow \mathrm{N}\gamma'{\approx}\tau_2.\{\hat{e}_1 : \tau_1\}^-_\gamma$
  - subcase $e_1 \not\equiv \hat{e}_1$
    1. by inverting (Unseal), $\Gamma \vdash e_1 : \tau_1$
    2. by induction, $e_1 \rightarrow e_1'$
    3. by reduction rule (18), $e \rightarrow \{e_1' : \tau_1\}^-_\gamma$

Deterministic reduction follows from the fact that all cases are disjoint, and in each case no other reduction rule is applicable. In particular, at most one of the rules (4a)-(4e) can ever apply. □

(kinds)	κ	$::=$	$\Omega \mid \kappa_1 \rightarrow \kappa_2$
(types)	τ	$::=$	$\alpha \mid \gamma \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \{\tau\}^-_{\gamma\approx\tau'} \mid$
			$\lambda{:}\kappa.\tau \mid \tau_1 \; \tau_2$
(terms)	e	$::=$	$x \mid \mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e \mid e_1 \; e_2 \mid \Lambda\alpha{:}\kappa.e \mid e \; \tau \mid$
			$\mathrm{N}\gamma{:}\kappa{\approx}\tau.e \mid \{e : \tau\}^\pm_{\gamma\approx\tau'} \mid$
			tcase $e_1 : \tau_1$ of $x : \tau_2$ then e_2 else e_3
(results)	$\hat{e}$	$::=$	$\hat{\hat{e}} \mid \mathrm{N}\gamma{:}\kappa{\approx}\tau.\hat{e}$
(values)	$\hat{\hat{e}}$	$::=$	$\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e \mid \Lambda\alpha{:}\kappa.e \mid$
			$\{\hat{\hat{e}} : \tau\}^+_{\gamma\approx\tau'}$ $(\tau = \gamma \; \vec{\tau})$
(env's)	Γ	$::=$	$\cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha \mid \Gamma, \gamma{\approx}\tau$

Figure 11: $\lambda^\omega_{\mathrm{N}}$ Syntax

A.4 Reduction of $\lambda^\omega_{\mathrm{N}}$

The reduction rules (4b)-(4e) for higher-order coercions replace two nested coercions by three. We have to prove that this cannot lead to diverging sequences of coercion reductions. We assign suitable weights $w \in \mathbb{N}$ to types of kind Ω and show that the total weight of non-value coercions in an expression decreases with each reduction step. Higher-order types are mapped to higher-order functions over $\mathbb{N}$,

$$\frac{}{\vdash \cdot : \diamond} \qquad \frac{\vdash \Gamma : \diamond \qquad \Gamma \vdash \tau : \Omega}{\vdash \Gamma, x{:}\tau : \diamond}(x \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\vdash \Gamma : \diamond}{\vdash \Gamma, \alpha{:}\kappa : \diamond}(\alpha \notin \mathrm{Dom}(\Gamma)) \qquad \frac{\vdash \Gamma : \diamond \qquad \Gamma \vdash \tau : \kappa}{\vdash \Gamma, \gamma{\approx}\tau : \diamond}(\gamma \notin \mathrm{Dom}(\Gamma))$$

$$\frac{\vdash \Gamma : \diamond \quad \alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \kappa \quad \gamma{\approx}\tau \in \Gamma}{\Gamma \vdash \gamma : \kappa}$$

$$\frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \Omega}{\Gamma \vdash \forall\alpha{:}\kappa.\tau : \Omega}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda\alpha{:}\kappa.\tau : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa \quad \gamma{\approx}\tau_2 \in \Gamma}{\Gamma \vdash \{\tau_1\}^{-}_{\gamma\approx\tau_2} : \kappa}$$

$$(\textsc{Id})\ \frac{\vdash \Gamma : \diamond \quad x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\textsc{App})\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$(\textsc{Fix})\ \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega \qquad \Gamma, x_1{:}\tau_2{\to}\tau_1, x_2{:}\tau_2 \vdash e : \tau_1}{\Gamma \vdash (\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e) : \tau_2 \to \tau_1}$$

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

$$(\textsc{New})\ \frac{\Gamma \vdash \tau' : \kappa \qquad \Gamma, \gamma{\approx}\tau' \vdash e : \tau}{\Gamma \vdash \mathsf{N}\gamma{:}\kappa{\approx}\tau'.e : \tau}(\gamma \notin \mathrm{FTN}(\tau))$$

$$(\textsc{Seal})\ \frac{\Gamma \vdash e : \{\tau\}^{-}_{\gamma\approx\tau'} \qquad \gamma{\approx}\tau' \in \Gamma}{\Gamma \vdash \{e : \tau\}^{+}_{\gamma\approx\tau'} : \tau}$$

$$(\textsc{Unseal})\ \frac{\Gamma \vdash e : \tau \qquad \gamma{\approx}\tau' \in \Gamma}{\Gamma \vdash \{e : \tau\}^{-}_{\gamma\approx\tau'} : \{\tau\}^{-}_{\gamma\approx\tau'}}$$

$$(\textsc{Tcase})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \tau_2 : \Omega \quad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\mathrm{tcase}\ e_1 : \tau_1\ \mathrm{of}\ x : \tau_2\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3) : \tau}$$

$$(\textsc{Equiv})\ \frac{\Gamma \vdash e : \tau' \quad \tau' = \tau \quad \Gamma \vdash \tau : \Omega}{\Gamma \vdash e : \tau}$$

Figure 12: $\lambda^{\omega}_{\mathrm{N}}$ Typing

$$\frac{}{\tau = \tau} \qquad \frac{\tau' = \tau}{\tau = \tau'} \qquad \frac{\tau = \tau' \quad \tau' = \tau''}{\tau = \tau''}$$

$$\frac{\tau_1 = \tau_1' \quad \tau_2 = \tau_2'}{\tau_1{\to}\tau_2 = \tau_1'{\to}\tau_2'} \qquad \frac{\tau = \tau'}{\forall\alpha{:}\kappa.\tau = \forall\alpha{:}\kappa.\tau'}$$

$$\frac{\tau = \tau'}{\lambda\alpha{:}\kappa.\tau = \lambda\alpha{:}\kappa.\tau'} \qquad \frac{\tau_1 = \tau_1' \quad \tau_2 = \tau_2'}{\tau_1\ \tau_2 = \tau_1'\ \tau_2'}$$

$$\frac{}{(\lambda\alpha{:}\kappa.\tau_1)\ \tau_2 = \tau_1[\alpha := \tau_2]} \qquad \frac{}{\lambda\alpha{:}\kappa.\tau\ \alpha = \tau}(\alpha \notin \mathrm{FTV}(\tau))$$

$$\frac{\tau_1 = \tau_1'}{\{\tau_1\}^{-}_{\gamma\approx\tau_2} = \{\tau_1'\}^{-}_{\gamma\approx\tau_2}}$$

$$\frac{}{\{\gamma\}^{-}_{\gamma\approx\tau} = \tau} \qquad \frac{}{\{\gamma'\}^{-}_{\gamma\approx\tau_2} = \gamma'}(\gamma \not\equiv \gamma')$$

$$\frac{}{\{\tau_1{\to}\tau_2\}^{-}_{\gamma\approx\tau_3} = \{\tau_1\}^{-}_{\gamma\approx\tau_3}{\to}\{\tau_2\}^{-}_{\gamma\approx\tau_3}}$$

$$\frac{}{\{\forall\alpha{:}\kappa.\tau_1\}^{-}_{\gamma\approx\tau_2} = \forall\alpha{:}\kappa.\{\tau_1\}^{-}_{\gamma\approx\tau_2}}(\alpha \notin \mathrm{FTV}(\tau_2))$$

$$\frac{}{\{\lambda\alpha{:}\kappa.\tau_1\}^{-}_{\gamma\approx\tau_2} = \lambda\alpha{:}\kappa.\{\tau_1\}^{-}_{\gamma\approx\tau_2}}(\alpha \notin \mathrm{FTV}(\tau_2))$$

$$\frac{}{\{\tau_1\ \tau_2\}^{-}_{\gamma\approx\tau_3} = \{\tau_1\}^{-}_{\gamma\approx\tau_3}\ \{\tau_2\}^{-}_{\gamma\approx\tau_3}}$$

Figure 13: $\lambda^{\omega}_{\mathrm{N}}$ Type equivalence

such that in general types are mapped to terms in a simply typed lambda calculus with natural numbers given by the following grammar:

$$\begin{array}{lcl} T & ::= & \mathbb{N} \mid T_1 \to T_2 \\ W & ::= & n \mid W_1 + W_2 \mid x \mid \lambda x{:}T.W \mid W_1\ W_2 \end{array}$$

Equality on these terms is defined by $\beta\eta$-equivalence plus the obvious rule for addition.

The following function defines the mapping from $\lambda^{\omega}_{\mathrm{N}}$-types and kinds to W-terms and types. We assume there is an injective mapping from type variables α to W-variables, which we write as x_α. The mapping is defined relative to an $\lambda^{\omega}_{\mathrm{N}}$-environment Γ that records the necessary type assertions for abstract types.

$$\begin{array}{rcl} W(\Omega) & = & \mathbb{N} \\ W(\kappa_1 \to \kappa_2) & = & W(\kappa_1) \to W(\kappa_2) \\ \\ W_\Gamma(\alpha) & = & x_\alpha \\ W_\Gamma(\gamma) & = & 1_{W(\kappa)} +_{W(\kappa)} 2 \cdot_{W(\kappa)} W_\Gamma(\tau) \\ & & (\text{with } \gamma{\approx}\tau \in \Gamma \text{ and } \Gamma \vdash \tau : \kappa) \\ W_\Gamma(\tau_1 \to \tau_2) & = & 1 + W_\Gamma(\tau_1) + W_\Gamma(\tau_2) \\ W_\Gamma(\forall\alpha{:}\kappa.\tau) & = & 1 + (\lambda x_\alpha{:}W(\kappa).W_{\Gamma,\alpha:\kappa}(\tau))\ 1_{W(\kappa)} \\ W_\Gamma(\{\tau'\}^{-}_{\gamma\approx\tau}) & = & (\lambda x_\alpha{:}W(\kappa).W_{\Gamma,\alpha:\kappa}(\tau'[\gamma := \alpha]))\ W_\Gamma(\tau) \\ & & (\text{with } \alpha \text{ fresh and } \Gamma \vdash \tau : \kappa) \\ W_\Gamma(\lambda\alpha{:}\kappa.\tau) & = & \lambda x_\alpha{:}W(\kappa).W_\Gamma(\tau) \\ W_\Gamma(\tau_1\ \tau_2) & = & W_\Gamma(\tau_1)\ W_\Gamma(\tau_2) \end{array}$$

By lemma 17 the definition for unsealed types is β-equivalent to

$$W_\Gamma(\{\tau'\}^{-}_{\gamma\approx\tau}) = W_\Gamma(\tau'[\gamma := \tau])$$

but is more suitable to inductive proofs. The definitions also use constants and addition at higher types, which are defined by lifting as follows:

$$\begin{array}{rcl} n_{\mathbb{N}} & = & n \\ n_{(T_1 \to T_2)} & = & \lambda x{:}T_1.n_{T_2} \\ \\ W_1 +_{\Omega} W_2 & = & W_1 + W_2 \\ W_1 +_{(T_1 \to T_2)} W_2 & = & \lambda x{:}T_1.W_1\, x +_{T_2} W_2\, x \\ \\ 2 \cdot_T W & = & W +_T W \end{array}$$

The weighting is defined such that the following equivalences hold. We hence do not need to distinguish between equivalent types:

Lemma 17 (Weight equivalence).

1. $W_\Gamma(\tau[\alpha := \tau']) = W_\Gamma(\tau)[x_\alpha := W_\Gamma(\tau')]$

2. $\tau_1 = \tau_2 \Rightarrow W_\Gamma(\tau_1) = W_\Gamma(\tau_2)$

3.

Proof.

1. By induction on the structure of τ.
2. By induction on the derivation of $\tau_1 = \tau_2$, using (1).

□

We define a a family of orderings on weights as follows:

$$\begin{array}{rcl} W_1 \leq_{\Omega} W_2 & \Leftrightarrow & W_1 \leq W_2 \\ W_1 \leq_{(T_1 \to T_2)} W_2 & \Leftrightarrow & \forall W : T_1,\ W_1\, W \leq_{T_2} W_2\, W \end{array}$$

Each ordering in the family is a partial order with a smallest element:

Lemma 18 (Partial order on weights). *Let* $W, W_1, W_2, W_3 : T$.

1. $W \leq_T W$

2. $W_1 \leq_T W_2 \wedge W_2 \leq_T W_3 \Rightarrow W_1 \leq_T W_2$

3. $W_1 \leq_T W_2 \wedge W_2 \leq_T W_1 \Rightarrow W_1 = W_2$

4. $0_T \leq_T W$

Proof. Each by induction on the structure of T. □

All W-functions are monotonic with respect to the ordering:

Lemma 19 (Weight monotonicity). *Let* $W, W_1, W_2 : T$ *and* $W' : T \to T'$.

1. $W_1 \leq_T W_1 +_T W_2$ *and* $W_2 \leq_T W_1 +_T W_2$

2. $W_1 \leq_T W_2 \Rightarrow W[x := W_1] \leq_T W[x := W_2]$ *(assuming* $W[x := W_i]$ *well-typed)*

3. $W_1 \leq_T W_2 \Rightarrow W'\, W_1 \leq_{T'} W'\, W_2$

4. $W_1 \leq_T W_2 \Rightarrow W_1 +_T W \leq_T W_2 +_T W$

Proof.

1. By induction on the structure of T.
2. By induction on the structure of W.
3. Follows immediatetly from (2).
4. By induction on the structure of T, using (3).

□

Let $<_T$ be the quasi order corresponding to $\leq_T$, i.e.:

$$W_1 <_T W_2 \Leftrightarrow W_1 \leq_T W_2 \wedge W_1 \neq W_2$$

Obviously, it is well-founded with 0_T being the smallest element. Moreover, we have:

Lemma 20 (Weight increase). *Let* $W : T$.

1. $W <_T 1_T +_T W$

2. $0_T <_T 1_T +_T W$

Proof.

1. From monotonicity it follows that $W \leq_T 1_T +_T W$. It hence suffices to show that $W \neq 1_T +_T W$, by induction on the structure of T.
2. Follows directly from (1) and transitivity.

□

A direct consequence is that the weighting function only assigns non-zero weights:

Lemma 21 (Non-zero weights). *If* $\Gamma \vdash \tau : \kappa$ *and* $\forall \alpha{:}\kappa' \in \Gamma, 0_{W(\kappa')} <_{W(\kappa')} x_\alpha$, *then* $0_{W(\kappa)} <_{W(\kappa)} W_\Gamma(\tau)$.

Proof. By induction on the structure of τ, using a slightly stronger induction hypothesis with the additional assumption $\forall \gamma{:}\kappa' {\approx} \tau' \in \Gamma, 0_{W(\kappa')} <_{W(\kappa')} W_\Gamma(\tau')$. The conjecture then follows by straight-forward induction on the structure of Γ. □

The weight of an abstract type is always larger than that of its representation:

Lemma 22 (Weight of abstract types). *Let* $\gamma{\approx}\tau \in \Gamma$ *and* $\Gamma \vdash \tau : \kappa$, *and* $\Gamma \vdash \tau' : \kappa'$. *Assume* $\forall \alpha{:}\kappa' \in \Gamma, 0_{W(\kappa')} <_{W(\kappa')} x_\alpha$.

1. $W_\Gamma(\tau) <_{W(\kappa)} W_\Gamma(\gamma)$

2. $W_\Gamma(\tau'[\gamma := \tau]) \leq_{W(\kappa')} W_\Gamma(\tau')$

3. $W_\Gamma(\{\tau'\}^-_{\gamma\approx\tau}) \leq_{W(\kappa')} W_\Gamma(\tau')$

Proof.

1. By monotonicity and weight increase:

$$\begin{array}{rl} W_\Gamma(\tau) \leq_{W(\kappa)} & 2 \cdot_{W(\kappa)} W_\Gamma(\tau) \\ <_{W(\kappa)} & 1_{W(\kappa)} +_{W(\kappa)} 2 \cdot_{W(\kappa)} W_\Gamma(\tau) \\ = & W_\Gamma(\gamma) \end{array}$$

2. By induction on the structure of τ'. We show the case for unsealing:

   - case $\tau' = \{\tau_1\}^-_{\gamma'\approx\tau_2}$:

$$\begin{array}{rl} & W_\Gamma(\{\tau_1\}^-_{\gamma'\approx\tau_2}[\gamma := \tau]) \\ = & W_\Gamma(\{\tau_1[\gamma := \tau]\}^-_{\gamma'\approx\tau_2[\gamma:=\tau]}) \\ = & (\lambda x_\alpha{:}\kappa_2.W_\Gamma(\tau_1[\gamma := \tau][\gamma' := \alpha]))\, W_\Gamma(\tau_2[\gamma := \tau]) \\ = & (\lambda x_\alpha{:}\kappa_2.W_\Gamma(\tau_1[\gamma' := \alpha][\gamma := \tau]))\, W_\Gamma(\tau_2[\gamma := \tau]) \\ \leq_{W(\kappa')} & (\lambda x_\alpha{:}\kappa_2.W_\Gamma(\tau_1[\gamma' := \alpha]))\, W_\Gamma(\tau_2) \\ = & W_\Gamma(\tau') \end{array}$$

3. Using (2) and lemma 17 (1):

$$\begin{array}{rcl} W_\Gamma(\{\tau'\}^-_{\gamma\approx\tau}) & = & (\lambda x_\alpha{:}\kappa.W_\Gamma(\tau'[\gamma := \alpha]))\, W_\Gamma(\tau) \\ & = & W_\Gamma(\tau[\gamma := \alpha])[x_\alpha := W_\Gamma(\tau)] \\ & = & W_\Gamma(\tau[\gamma := \tau]) \\ & \leq_{W(\kappa')} & W_\Gamma(\tau') \end{array}$$

□

The coercion weight of $\lambda_{\mathrm{N}}^{\omega}$-expressions can now be defined as follows:

$$\begin{array}{rcl}
W_\Gamma(x) & = & 0 \\
W_\Gamma(\mathrm{fix}\, x_1(x_2{:}\tau_2){:}\tau_1.e) & = & W_\Gamma(e) \\
W_\Gamma(e_1\ e_2) & = & W_\Gamma(e_1) + W_\Gamma(e_2) \\
W_\Gamma(\Lambda\alpha{:}\kappa.e) & = & W_{\Gamma,\alpha:\kappa}(e) \\
W_\Gamma(e\ \tau) & = & W_\Gamma(e) \\
W_\Gamma(\mathrm{N}\gamma{:}\kappa{\approx}\tau.e) & = & W_{\Gamma,\gamma\approx\tau}(e) \\
W_\Gamma(\{e:\tau'\}^{+}_{\gamma\approx\tau}) & = & \left\{\begin{array}{ll} W_\Gamma(e) & \text{if } \tau' = \gamma\,\vec{\tau} \\ W_\Gamma(e) + \underline{W}_\Gamma(\tau') & \text{otherwise} \end{array}\right. \\
W_\Gamma(\{e:\tau'\}^{-}_{\gamma\approx\tau}) & = & W_\Gamma(e) + \underline{W}_\Gamma(\tau') \\
W_\Gamma(\text{tcase } e_1{:}\tau_1 \text{ of } x{:}\tau_2 & & \\
\text{then } e_2 \text{ else } e_3) & = & W_\Gamma(e_1) + W_\Gamma(e_2) + W_\Gamma(e_3)
\end{array}$$

where

$$\underline{W}_\Gamma(\tau) := W_\Gamma(\tau)\sigma \qquad (\text{with } \sigma = [x_\alpha := 1_{W(\kappa)} \mid \alpha{:}\kappa \in \Gamma])$$

Note that coercions of the form $\{e : \gamma\ \vec{\tau}\}^{+}_{\gamma}$ do not add any weight, we only weigh non-value coercions.

Theorem 8 (Coercion convergence). *There are no infinite sequences of reductions using only the coercion rules (3), (4a)–(4e), (5) and (6).*

Proof. In each of the rules the redex (left-hand side) has non-zero weight. Simultanously, each reduct (right-hand side) has less weight than the corresponding redex, which can be shown with the previous lemmata (we write $\vec{W}(\vec{\tau})$ for the vector of weights of types $\vec{\tau}$):

- case (3):

$$\begin{array}{rcl}
W_\Gamma(\mathrm{LHS}) & = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\gamma\ \vec{\tau}) \\
& > & W_\Gamma(\hat{\hat{e}}) \\
& = & W_\Gamma(\mathrm{RHS})
\end{array}$$

- case (4a):

$$\begin{array}{rcl}
W_\Gamma(\mathrm{LHS}) & = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\gamma\ \vec{\tau}) \\
& > & W_\Gamma(\hat{\hat{e}}) \\
& = & W_\Gamma(\mathrm{RHS})
\end{array}$$

- case (4b):

$$\begin{array}{rcl}
W_\Gamma(\mathrm{LHS}) & = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\gamma'\ \vec{\tau}) \\
& = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\gamma')(\underline{\vec{W}}_\Gamma(\vec{\tau})) \\
& = & W_\Gamma(\hat{\hat{e}}) + \\
& & (1_{W(\kappa)} +_{W(\kappa)} 2 \cdot_{W(\kappa)} \underline{W}_\Gamma(\tau'))(\underline{\vec{W}}_\Gamma(\vec{\tau})) \\
& > & W_\Gamma(\hat{\hat{e}}) + (2 \cdot_{W(\kappa)} \underline{W}_\Gamma(\tau'))(\underline{\vec{W}}_\Gamma(\vec{\tau})) \\
& = & W_\Gamma(\hat{\hat{e}}) + 2 \cdot (\underline{W}_\Gamma(\tau')(\underline{\vec{W}}_\Gamma(\vec{\tau}))) \\
& = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau')(\underline{\vec{W}}_\Gamma(\vec{\tau})) \\
& & \qquad + \underline{W}_\Gamma(\tau')(\underline{\vec{W}}_\Gamma(\vec{\tau})) \\
& \geq & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau')(\underline{\vec{W}}_\Gamma(\{\{\vec{\tau}\}^{-}_{\gamma}\}^{-}_{\gamma'})) \\
& & \qquad + \underline{W}_\Gamma(\tau')(\underline{\vec{W}}_\Gamma(\{\vec{\tau}\}^{-}_{\gamma'})) \\
& \geq & W_\Gamma(\mathrm{RHS})
\end{array}$$

  The last line is an inequation because $\tau'\,\vec{\tau}'$ may be equivalent to $\gamma\,\vec{\tau}''$, in which case it is not weighed for the right-hand side.

- cases (4c)–(4e): Likewise

- case (5):

$$\begin{array}{rcl}
W_\Gamma(\mathrm{LHS}) & = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau_1 \to \tau_2) \\
& = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau_1) + \underline{W}_\Gamma(\tau_2) + 1 \\
& > & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau_1) + \underline{W}_\Gamma(\tau_2) \\
& \geq & W_\Gamma(\mathrm{RHS})
\end{array}$$

  The last line is an inequation because τ_1 and τ_2 may each be equivalent to $\gamma\,\vec{\tau}$.

- case (6):

$$\begin{array}{rcl}
W_\Gamma(\mathrm{LHS}) & = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\forall\alpha{:}\kappa.\tau_1) \\
& = & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau_1)[x_\alpha := 1_{W(\kappa)}] + 1 \\
& > & W_\Gamma(\hat{\hat{e}}) + \underline{W}_\Gamma(\tau_1)[x_\alpha := 1_{W(\kappa)}] \\
& \geq & W_\Gamma(\mathrm{RHS})
\end{array}$$

  The last line is an inequation because τ_1 may be equivalent to $\gamma\,\vec{\tau}$.

□

A.5 Opacity for $\lambda_{\mathrm{N}}^{\omega}$

The lemmata on substitutions from section A.2 easily extend to $\lambda_{\mathrm{N}}^{\omega}$. Opacity itself needs to be restated as follows:

Theorem 9 ($\lambda_{\mathrm{N}}^{\omega}$ Opacity). *Let Γ be an environment containing only type assertions (i.e. $\Gamma \equiv \gamma_1'{\approx}\tau_1, \cdots, \gamma_n'{\approx}\tau_n$) and e an expression with $\Gamma, \alpha{:}\kappa, x{:}\alpha\,\vec{\tau} \vdash e : \tau$ for some $\vec{\tau}$. Assume a set of values $\hat{\hat{e}}_i$ ($i = 1, \ldots, n$) such that $\gamma_i{\approx}\tau_i \vdash \hat{\hat{e}}_i : \gamma_i\,\vec{\tau}$ with $\gamma_i \notin \mathrm{Dom}(\Gamma)$. Let $\sigma_i = [\alpha := \gamma_i, x := \hat{\hat{e}}_i]$. If $e\sigma_1 \not\equiv \hat{\hat{e}}$ then there is an e' with $\Gamma, \alpha{:}\kappa, x{:}\alpha\,\vec{\tau} \vdash e' : \tau$ such that*

$$e\sigma_i \to e'\sigma_i$$

for all σ_i.

Proof. Note first that by validity, $\Gamma, \alpha{:}\kappa \vdash \alpha\,\vec{\tau} : \Omega$. By canonical values, $\hat{\hat{e}}_i \equiv \{\hat{\hat{e}}_i' : \tau_i'\}^{+}_{\gamma_i}$ (with $\tau_i' = \gamma_i\,\vec{\tau}$). By progress, $e\sigma_1 \to e''$ for some e''. We can hence prove the conjecture by induction on the derivation of $\to$. Only cases (3)–(4e) are new:

- case (3): Due to the assumptions, $\gamma \not\equiv \gamma_i$ for any i. Hence, there are 2 possibilities:
  - subcase $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'}$ with $\tau_1 = \tau_2 = \gamma\,\vec{\tau}'$
    By substitution (1), $\tau_1\sigma_i = \tau_2\sigma_i = \gamma\ \vec{\tau}'\sigma$. Hence for all i we have
    $$e\sigma_i \ \equiv\ \{\{\hat{\hat{e}}\sigma_i : \tau_1\sigma_i\}^{+}_{\gamma\approx\tau'\sigma_i} : \tau_2\sigma_i\}^{-}_{\gamma\approx\tau'\sigma_i} \ \to\ \hat{\hat{e}}\sigma_i$$
    Hence $e' \equiv \hat{\hat{e}}$, which is well-typed by inversion.
  - subcase $e \equiv \{\{x : \tau_1\}^{+}_{\gamma\approx\tau'} : \tau_2\}^{-}_{\gamma\approx\tau'}$
    Likewise.
- case (4a): Due to the assumptions, $\gamma \not\equiv \gamma_i$ and $\gamma' \not\equiv \gamma_i$ for any i. Hence 2 subcases:
  - subcase $e \equiv \{\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma'\approx\tau''} : \tau_2\}^{\pm}_{\gamma\approx\tau'}$
    (with $\tau_1 = \gamma'\,\vec{\tau}_1'$ and $\tau_2 = \gamma'\,\vec{\tau}_2'$ and $\gamma \notin \mathrm{FTN}(\vec{\tau}_2'))$
    By substitution, $\tau_1\sigma_i = \gamma'\,\vec{\tau}_1'\sigma_i$ and $\tau_2\sigma_i = \gamma'\,\vec{\tau}_2'\sigma_i$ and $\gamma \notin \mathrm{FTN}(\vec{\tau}_2'\sigma_i)$, hence for all i we have
    $$\begin{array}{rcl}
    e\sigma_i & \equiv & \{\{\hat{\hat{e}}\sigma_i : \tau_1\sigma_i\}^{+}_{\gamma\approx\tau'\sigma_i} : \tau_2\sigma_i\}^{\pm}_{\gamma\approx\tau'\sigma_i} \\
    & \to & \{\hat{\hat{e}}\sigma_i : \tau_1\sigma_i\}^{+}_{\gamma\approx\tau'\sigma_i} \\
    & \equiv & (\{\hat{\hat{e}} : \tau_1\}^{+}_{\gamma\approx\tau'})\sigma_i
    \end{array}$$

Hence $e' \equiv \{\hat{\hat{e}} : \tau_1\}^+_{\gamma\approx\tau'}$, which is well-typed by inversion.

  - subcase $e \equiv \{\{x : \tau_1\}^+_{\gamma'\approx\tau''} : \tau_2\}^\pm_{\gamma\approx\tau'}$
    (with $\tau_1 = \gamma'\,\vec{\tau}'_1$ and $\tau_2 = \gamma'\,\vec{\tau}'_2$ and $\gamma \notin \mathrm{FTN}(\vec{\tau}'_2)$)
    Likewise.

- cases (4b)–(4e): Similarly. We mainly need to verify that each side condition is invariant under each substitution σ_i, which is easy to see.

□

A.6 Typing $\lambda^\omega_{\check{\mathrm{N}}}$

For $\lambda^\omega_{\check{\mathrm{N}}}$, the higher-order calculus extended with Ň, the typing rules from figure 12 must be lifted to pre-terms (by substituting all occurences of e by $\check{\check{e}}$) and a rule for Ň-binders must be added:

$$(\textsc{Snew})\ \frac{\Gamma \vdash \tau' : \kappa \qquad \Gamma, \gamma{\approx}\tau' \vdash e : \tau}{\Gamma \vdash \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau'.e : \tau}(\gamma \notin \mathrm{FTV}(\tau))$$

Proofs for most lemmas from the previous section scale to $\lambda^\omega_{\check{\mathrm{N}}}$ trivially when replacing $\check{\check{e}}$ for e. The cases for Ň proceed as for plain N. Inversion needs to be extended:

Lemma 23 (Ň Inversion). *If $\Gamma \vdash \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau'.e : \tau$, then $\Gamma \vdash \tau' : \kappa$ and $\Gamma, \gamma{\approx}\tau' \vdash e : \tau$ and $\gamma \notin \mathrm{FTN}(\tau)$.*

In the main text we have omitted some technical detail with respect to name substitution in terms: obviously, the notation $e[\gamma := \tau]$ substitutes free type names in proper type terms only. It does not substitute occurences of γ in coercion subscripts, i.e. the subscript left-hand side in $\{e : \tau'\}^\pm_{\gamma\approx\tau}$. The corresponding substitutions have been left out in the reduction rules (23) and (25) as given in figure 10. In order to make these rules precise, we need to define a *representation substitution*, written $e[\gamma :\approx \tau]$, that replaces coercion subscripts as defined by the closure of the equation

$$(\{e : \tau_1\}^\pm_{\gamma\approx\tau})[\gamma :\approx \tau'] \quad = \quad \{e : \tau_1[\gamma :\approx \tau']\}^\pm_{\gamma\approx\tau'}$$

The precise formulation of the reduction rules with representation substitution is given in figure 14. The following lemma holds:

Lemma 24 (Representation substitution). *If $\Gamma, \gamma{\approx}\tau_1 \vdash \gamma : \kappa$ and $\Gamma, \gamma{\approx}\tau_2 \vdash \gamma\,\vec{\tau} : \kappa$ for some κ, and $\Gamma, \gamma{\approx}\tau_1 \vdash \check{e} : \tau$, then $\Gamma, \gamma{\approx}\tau_2 \vdash \check{e}[\gamma :\approx \tau_2][\gamma := \gamma\,\vec{\tau}] : \tau[\gamma := \gamma\,\vec{\tau}]$.*

Proof. By induction on the derivation. □

Using the adapted lemmata and rules, preservation can be proved:

Theorem 10 ($\lambda^\omega_{\check{\mathrm{N}}}$ Preservation). *If $\Gamma \vdash \check{\check{e}} : \tau$ and $\check{\check{e}} \rightarrow \check{\check{e}}'$, then $\Gamma \vdash \check{\check{e}}' : \tau$.*

Proof. By straight-forward extension of the inductive proof for theorem 6. The only non-obvious new cases are:

- case $\check{\check{e}} \equiv \Lambda\alpha{:}\kappa'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}$
  and $\check{\check{e}}' \equiv \check{\mathrm{N}}\gamma{:}\kappa'{\rightarrow}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau).\Lambda\alpha{:}\kappa'.\check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau][\gamma := \gamma\,\alpha]$
  1. by inverting (Gen), $\tau = \forall\alpha{:}\kappa'.\tau'$ and $\Gamma, \alpha{:}\kappa' \vdash \check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} : \tau'$
  2. by inverting (Snew), $\Gamma, \alpha{:}\kappa' \vdash \tau : \kappa$ and $\Gamma, \alpha{:}\kappa', \gamma{\approx}\tau \vdash \check{e} : \tau'$ and $\gamma \notin \mathrm{FTN}(\tau')$
  3. by kinding, $\Gamma \vdash \lambda\alpha{:}\kappa'.\tau : \kappa'{\rightarrow}\kappa$
  4. by kinding, $\Gamma, \alpha{:}\kappa', \gamma{\approx}(\lambda\alpha{:}\kappa'.\tau) \vdash \gamma\ \alpha : \kappa$
  5. by representation substitution, $\Gamma, \alpha{:}\kappa', \gamma{\approx}(\lambda\alpha{:}\kappa'.\tau) \vdash \check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau][\gamma := \gamma\,\alpha] : \tau'$
  6. by rule (Gen), $\Gamma, \gamma{\approx}(\lambda\alpha{:}\kappa'.\tau) \vdash \Lambda\alpha{:}\kappa'.\check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau][\gamma := \gamma\,\alpha] : \forall\alpha{:}\kappa'.\tau'$
  7. by rule (Snew), $\Gamma \vdash \check{\check{e}}' : \forall\alpha{:}\kappa'.\tau'$
- case $\check{\check{e}} \equiv \mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e}$
  and $\check{\check{e}}' \equiv \check{\mathrm{N}}\gamma{:}\kappa'{\rightarrow}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau[\gamma'{:=}\alpha]).\mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau[\gamma'{:=}\alpha]][\gamma{:=}\gamma\,\gamma']$
  Similarly.

□

Theorem 11 ($\lambda^\omega_{\check{\mathrm{N}}}$ Progress). *Let Γ be an environment containing only type assertions (i.e. $\Gamma \equiv \gamma_1{\approx}\tau_1, \cdots, \gamma_n{\approx}\tau_n$). If $\Gamma \vdash \check{\check{e}} : \tau_0$, then either $\check{\check{e}} \equiv \hat{e}$ for some result $\hat{e}$, or $\check{\check{e}} \rightarrow \check{\check{e}}'$ for some expression $\check{\check{e}}'$. In the latter case, if $\check{\check{e}} \equiv e$ then $\check{\check{e}}' \equiv e'$.*

Proof. By straight-forward extension of the proof for $\lambda^\omega_{\mathrm{N}}$ with the cases for $\check{\check{e}} \not\equiv e$, using the slightly stronger induction hypothesis. For example,

- case $\check{\check{e}} \equiv \{\check{\check{e}}_1 : \tau_1\}^-_{\gamma\approx\tau'}$
  - subcases $\check{\check{e}}_1 \equiv \hat{\hat{e}}_1$
    as for $\lambda^\omega_{\mathrm{N}}$
  - subcase $\check{\check{e}}_1 \equiv \mathrm{N}\gamma'{\approx}\tau_2.\hat{e}_1$
    as for $\lambda^\omega_{\mathrm{N}}$
  - subcase $\check{\check{e}}_1 \equiv e_1$
    1. by inverting (Unseal), $\Gamma \vdash e_1 : \tau_1$
    2. by induction, $e_1 \rightarrow e'_1$
    3. by reduction rule (18), $\check{\check{e}} \rightarrow \{e'_1 : \tau_1\}^-_{\gamma\approx\tau'}$
  - subcase $\check{\check{e}}_1 \equiv \check{\mathrm{N}}\gamma'{\approx}\tau_2.\check{e}_1$
    1. by reduction rule (12), $\check{\check{e}} \rightarrow \check{\mathrm{N}}\gamma'{\approx}\tau_2.\{\check{e}_1 : \tau_1\}^-_{\gamma\approx\tau'}$
  - subcase $\check{\check{e}}_1 \not\equiv \check{e}_1$
    1. by inverting (Unseal), $\Gamma \vdash \check{\check{e}}_1 : \tau_1$
    2. by induction, $\check{\check{e}}_1 \rightarrow \check{\check{e}}'_1$
    3. by reduction rule (37), $\check{\check{e}} \rightarrow \{\check{\check{e}}'_1 : \tau_1\}^-_{\gamma\approx\tau'}$

□

A.7 Termination of Pre-term Reduction

Let W be a weight function on pre-terms inductively defined as follows:

$$\begin{array}{rcl}
W(x) & = & 0 \\
W(\mathrm{fix}\,x_1(x_2{:}\tau_2){:}\tau_1.\check{\check{e}}) & = & 2W(\check{\check{e}}) \\
W(\check{\check{e}}_1\ \check{\check{e}}_2) & = & 2(W(\check{\check{e}}_1) + W(\check{\check{e}}_2)) \\
W(\Lambda\alpha{:}\kappa.\check{\check{e}}) & = & 2W(\check{\check{e}}) \\
W(\check{\check{e}}\ \tau) & = & 2W(\check{\check{e}}) \\
W(\mathrm{N}\gamma{:}\kappa{\approx}\tau.\check{\check{e}}) & = & 2W(\check{\check{e}}) \\
W(\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{\check{e}}) & = & 1 + W(\check{\check{e}}) \\
W(\{\check{\check{e}} : \tau\}^\pm_{\gamma\approx\tau'}) & = & 2W(\check{\check{e}}) \\
W(\mathrm{tcase}\ \check{\check{e}}_1 : \tau_1\ \mathrm{of}\ x : \tau_2 & & \\
\mathrm{then}\ \check{\check{e}}_2\ \mathrm{else}\ \check{\check{e}}_3) & = & 2(W(\check{\check{e}}_1) + W(\check{\check{e}}_2) + W(\check{\check{e}}_3))
\end{array}$$

Lemma 25 (Term Weights). *A pre-term $\check{\check{e}}$ has weight $W(\check{\check{e}}) = 0$ if and only if $\check{\check{e}} \equiv e$ for some plain term e.*

$$\begin{array}{lrcl}
(23) & \Lambda\alpha{:}\kappa'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} & \rightarrow & \check{\mathrm{N}}\gamma{:}\kappa'{\rightarrow}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau).\Lambda\alpha{:}\kappa'.\check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau][\gamma := \gamma\,\alpha] \\
(25) & \mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{\mathrm{N}}\gamma{:}\kappa{\approx}\tau.\check{e} & \rightarrow & \check{\mathrm{N}}\gamma{:}\kappa'{\rightarrow}\kappa{\approx}(\lambda\alpha{:}\kappa'.\tau[\gamma' := \alpha]).\mathrm{N}\gamma'{:}\kappa'{\approx}\tau'.\check{e}[\gamma :\approx \lambda\alpha{:}\kappa'.\tau[\gamma' := \alpha]][\gamma := \gamma\,\alpha]
\end{array}$$

Figure 14: Revised reduction rules for applicative generation

PROOF. Both directions by induction on the structure of $\check{\check{e}}$. □

LEMMA 26 (WEIGHT REDUCTION). *For a pre-term $\check{\check{e}}$ with $\check{\check{e}} \rightarrow \check{\check{e}}'$, either $W(\check{\check{e}}) = W(\check{\check{e}}') = 0$, or $W(\check{\check{e}}) > W(\check{\check{e}}')$.*

PROOF. By induction on the derivation of $\check{\check{e}} \rightarrow \check{\check{e}}'$. □

LEMMA 27 (PHASE SEPARATION). *For a pre-term $\check{\check{e}}$ with $\check{\check{e}} \rightarrow \check{\check{e}}'$, if $\check{\check{e}} \equiv e$ then the reduction step involves only rules (1)-(19), otherwise reduction involves only rules (20)-(41).*

PROOF. By easy extension of the inductive proof for progress (theorem 11). □

THEOREM 12 (FINITE PRE-TERM REDUCTION). *Every well-typed pre-term $\check{\check{e}}$ reduces to a plain term e by a finite reduction sequence involving only rules (20)–(41).*

PROOF. If $\check{\check{e}}$ is a plain term then the conjecture holds immediately. Otherwise, $W(\check{\check{e}}) > 0$ by lemma 25. By progress, there is a reduction $\check{\check{e}} \rightarrow \check{\check{e}}'$. By phase separation, the reduction uses only rules (20)–(41). By weight reduction, $W(\check{\check{e}}') < W(\check{\check{e}})$, so that there cannot be an infinite reduction sequence without reaching a plain term. □