Meta-Programming with Names and Necessity

Aleksandar Nanevski

April 2002
CMU-CS-02-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Meta-programming languages provide infrastructure to generate and execute object programs at run-time. In a typed setting, they contain a modal type constructor which classifies object code. These code types generally come in two flavors: closed and open. Closed code expressions can be invoked at run-time, but the computations over them are more rigid, and typically produce less efficient residual object programs. Open code provides better inlining and partial evaluation of object programs, but once constructed, expressions of this type cannot be evaluated.
Recent work in this area has focused on combining the two notions into a sound system. We present a novel way to achieve this. It is based on adding the notion of *names* from the work on Nominal Logic and FreshML to the $\lambda^{\Box}$-calculus of proof terms for the *necessity* fragment of modal logic S4. The resulting language provides a more fine-grained control over free variables of object programs when compared to the existing languages for meta-programming. In addition, this approach lends itself well to addition of intensional code analysis, i.e. capability of meta programs to inspect and destruct object programs at run-time in a type-safe manner, which we also undertake.

Keywords: meta-programming, modal logic, nominal logic, intensional code analysis, staged computation, run-time code generation, symbolic computation

1 Introduction

Meta-programming is a paradigm referring to the ability to algorithmically compose programs of a certain object language, through a program written in a meta-language. A particularly intriguing instance of this concept, and the one we are interested in in this work, is when the meta and the object language are: (1) the *same*, or the object language is a subset of the meta language; and (2) *typed* functional languages. A language satisfying (1) adds the possibility to also invoke the generated programs at run-time. We refer to this setup as *homogeneous* meta-programming.

Among some of the advantages of meta-programming and of its homogeneous and typed variant we distinguish the following (and see [She01] for a comprehensive analysis).

1. **Efficiency.** Rather than using one general procedure to solve many different instances of a problem, a program can generate specialized (and hence more efficient) subroutines for each particular case. If the language is capable of executing thus generated procedures, the program can choose dynamically, depending on a run-time value of a certain variable or expression, which one is most suitable to invoke. In particular, this is the idea behind the functional programming concept of *staged computation*, and has been considered before in a typed setting (see [LL96], [WLP98], [WLPD98], [DP96]).

2. **Maintainability.** Instead of maintaining a number of specialized, but related, subprograms, it is easier to maintain their generator. In a language capable of invoking the generated code, there is an added bonus of being able to accentuate the relationship between the synthesized code and its producer; the subroutines can be generated and bound to their respective identifiers in the initialization stage of the program execution, rather then generated and saved into a separate file of the build tree for later compilation and linking.

Languages in which programs can not only be composed and executed but also have their structure inspected add further advantages. Efficiency benefits from various optimizations that can be performed knowing the structure of the code. For example, Griewank reports in [Gri89], on a way to reuse common subexpressions of a numerical function in order to compute its value at a certain point and the value of its n-dimensional gradient, but in such a way that the complexity of both evaluations performed together does not grow with n. Maintainability (and in general the whole program development process) benefits from the presence of types on both the level of synthesized code, and on the level of program generators. Finally, there are applications from various domains, which seem to call for the capability to execute a certain function as well as recurse over its structure: see [Roz93] for examples in computer graphics and numerical analysis, and [RP02] for an example in machine learning and probabilistic modeling.

Recent developments in type systems for meta-programming have been centered around two particular modal lambda calculi: $\lambda^{\Box}$ and $\lambda^{\bigcirc}$. The first is a language of proof terms for the modal logic S4, whose necessity constructor $\Box$ annotates *valid* propositions ([DP96], [PD01]). The second is the proof language for discrete linear temporal logic, whose modal operator $\bigcirc$ annotates the time-level separation between propositions [Dav96]. Both calculi provide a distinction between levels of terms, and this explains their use in meta-programming. The lowest, level 0, is the meta language, which is used to manipulate the terms on level 1 (terms of type $\Box A$ in $\lambda^{\Box}$ and $\bigcirc A$ in $\lambda^{\bigcirc}$). This first level is the meta language for the level 2 containing another stratum of boxed and circled types, etc. Functional programming interpretation of these two constructors assigns type $\Box A$ to *closed code* i.e. to closed terms of type A, while $\bigcirc A$ is the type of *postponed* code, i.e., it classifies terms of type A which are associated with the subsequent time moment. Postponed code in $\lambda^{\bigcirc}$ may refer to outside context variables, as long as they are on the same temporal level, and this has contributed to it frequently being associated with the notion of *open* code. For this exact reason, the concept of code in $\lambda^{\bigcirc}$ is obviously broader, allowing for more expressiveness and generation of better and more optimized residual programs (as already observed in [Dav96]), but, unlike $\lambda^{\Box}$, it has no language support for mixing of the code levels, and in particular, no language support for execution of the generated code.

There have been several proposed systems which incorporate the advantages from both languages, most notable being MetaML ([MTBS99], [Tah99b], [CMT00], [CMS01]). MetaML starts with the postponed/open code type of $\lambda^{\bigcirc}$ and strengthens the notion to introduce closed code as its refinement – as postponed code which happens to contain no variables declared outside of it. The approach of our work is the opposite.

Rather than refining the notion of open code, we relax the notion of closed code. We start with the system of $\lambda^{\Box}$, but provide the additional expressiveness by allowing the code to contain specified object variables as free (and rudiments of this idea have already been considered in [Nie01]). The fact that a given code expression depends on a set of free variables will be reflected in its type. The object variables themselves are represented by a separate semantic category of names (also called symbols or atoms), which admits equality. The treatment of names is adopted, with certain modifications, from the work on Nominal Logic and FreshML by Pitts and Gabbay ([GP01], [PG00], [Pit01], [Gab00]). This design choice lends itself well to the addition, in an *orthogonal way*, of intensional code analysis, which we also undertake for the simply-typed fragment of the language. Thus, we can also treat our simply-typed code expressions as data; they can not only be evaluated, but can also be compared for structural equality and destructed via pattern-matching, much in the same way as one would work with any abstract syntax tree. The binding constructs are handled through the *name abstraction* mechanism of the meta level, rather than through concrete representations or through *variable abstraction* (i.e. higher-order abstract syntax; see [PE88]). Using a separate meta-level binding construct to deal with bindings on the object level has been advocated in [She01], and a similar idea has already been used in [Mil90].

The rest of the document is organized as follows: Section 2 is a brief exposition of the previous work on $\lambda^{\Box}$, $\lambda^{\bigcirc}$, MetaML, and FreshML. Our type system is described in Section 3, while Section 4 contains the theoretical development behind it. In Section 5, we present the operational semantics of the language and prove the Type Preservation and Progress theorem. Intensional code analysis is introduced in Section 6. Finally, we illustrate the type system with example programs, before outlining the future work in Section 7.

2 Background

In this section we review the previous work on languages for meta-programming. The motivation is to quickly present the intuition behind the various calculi, not to give a detailed or rigorous treatise. For this reason, we limit ourselves to only operational exposition, because it demands the least amount of background theoretical machinery. Furthermore, we describe only the core part of a language, but in the presented examples we often assume presence of certain types and term constructs, like integers, conditionals or recursion, which are needed to illustrate the point. In any case, addition of these will never present any theoretical problems.

The example we will use throughout is the exponentiation function, presented below in a MinML-like notation.

```
pow = fix pow:int->int->int.
         λx:int. λn:int.
         if n = 0 then 1 else x * pow x (n-1)
```

2.1 $\lambda^{\Box}$

The functional programming motivation behind the $\lambda^{\Box}$ calculus is to ensure proper staging of programs. For example, consider the following equivalent of our exponentiation function.

```
powbox1 = fix pow:int->int->int.
          λn:int.
              if n = 0 then λx:int.1
              else
                 let val u = pow (n - 1)
                 in
                      λx:int. x * u(x)
                 end
```

One can argue that `powbox1` is preferred to `pow` because it allows a partial evaluation of the function when only n is known, but not x. Indeed, in such a situation, the expression `powbox1 n` produces a residual function specialized to computing the n-th power of its argument x. In particular, this function will not perform any operations or take decisions at run-time based on the value of n; in fact, it does not even depend on n – all the computation steps dependent on n have been taken during the partial evaluation.

The type system of $\lambda^{\Box}$ allows the programmer to specify the intended staging of operations by annotating subterms of the program which are to be *closed*, i.e. independent of the variables from the surrounding code. Then the type system can check whether the written code conforms to the staging specification, making staging errors into type errors.

$$
\begin{array}{llcl}
\textit{Types} & A & ::= & 1 \mid A_1 \to A_2 \mid \Box A \\
\textit{Terms} & e & ::= & x \mid * \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \\
\textit{Contexts} & \Delta, \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\textit{Values} & v & ::= & * \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e
\end{array}
$$

To declare that a subterm e of type A is closed, $\lambda^{\Box}$ provides the type constructor $\Box$ and its introduction term **box**, so that **box** e has type $\Box A$ (consult the typing rules in Figure 1). It is in this sense that the type constructor $\Box$ is associated with closed code. In the spirit of this "run-time code generation" interpretation, the operational semantics does not proscribe reductions under the box; boxed expressions are values.

The elimination form for $\Box$ is **let box** $u = e_1$ **in** e_2. Operationally, it evaluates e_1 to a boxed value, then binds the unreduced expression under that box to u in e_2. Notice that u is not an ordinary variable – it stands for an unevaluated *closed* expression, rather than a value. This fact motivates having two variable contexts in the typing judgment: Γ for ordinary value variables, and Δ for closed expression variables. In order to have proper staging, closed code expressions should not depend on value variables from Γ, but they can depend on expression variables from Δ.

The staging of `powbox1` can be made explicit in the following way.

```
powbox2 = fix pow:int -> □(int->int).
            λn:int.
              if n = 0 then box (λx:int. 1)
              else
                 let box u = pow (n - 1)
                 in
                    box (λx:int. x * u(x))
                 end
```

Application of `powbox2` at argument 2 produces a boxed function for squaring.

```
- sqbox = powbox2 2;
val sqbox = box (λx:int. x *
                 (λy:int. y *
                     (λz:int. 1) y ) x ) : □(int -> int)
```

It can then be evaluated in order to be applied itself.

```
- sq = (let box u = sqbox in u);
val sq = [fn] : int -> int
- sq 3;
val it = 9 : int
```

2.2 $\lambda^{\bigcirc}$

The $\lambda^{\Box}$ staging of `powbox2` which was presented in the previous section, leaves a lot to be desired. In particular, the residual programs that `powbox2` produces, e.g. `sqbox`, contain variable-for-variable redices, and hence are not as efficient as one would want. The reason, of course, is that boxed/code expressions are values; they completely suspend the evaluation of the enclosed term. As witnessed by the example of `sqbox`, it may be advantageous to have a general programming mechanism[1] whereby one could specify that certain reductions[2] in a code expression are to take place. Of course, $\lambda^{\Box}$ already contains mechanisms to

[1] Thus we are interested in something more than just devising an operational semantics which scans boxed expressions and actually reduces all variable-for-variable redices.

[2] And here, reductions are understood in the broader sense of λ-calculus, i.e. they can occur under a λ-abstraction.

Typechecking

$$\frac{}{\Delta;\Gamma \vdash * : 1} \qquad \frac{x{:}A \in \Gamma}{\Delta;\Gamma \vdash x : A} \qquad \frac{u{:}A \in \Delta}{\Delta;\Gamma \vdash u : A}$$

$$\frac{\Delta;\Gamma, x{:}A \vdash e : B}{\Delta;\Gamma \vdash \lambda x{:}A.\, e : A \to B} \qquad \frac{\Delta;\Gamma \vdash e_1 : A \to B \qquad \Delta;\Gamma \vdash e_2 : A}{\Delta;\Gamma \vdash e_1\ e_2 : B}$$

$$\frac{\Delta;\cdot \vdash e : A}{\Delta;\Gamma \vdash \mathbf{box}\ e : \Box A} \qquad \frac{\Delta;\Gamma \vdash e_1 : \Box A \qquad \Delta, u{:}A;\Gamma \vdash e_2 : B}{\Delta;\Gamma \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B}$$

Operational semantics

$$\frac{}{* \hookrightarrow *} \qquad \frac{}{\lambda x{:}A.\, e \hookrightarrow \lambda x{:}A.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x{:}A.\, e \qquad e_2 \hookrightarrow v_2 \qquad [v_2/x]e \hookrightarrow v}{e_1\ e_2 \hookrightarrow v}$$

$$\frac{}{\mathbf{box}\ e \hookrightarrow \mathbf{box}\ e} \qquad \frac{e_1 \hookrightarrow \mathbf{box}\ e \qquad [e/u]e_2 \hookrightarrow v}{\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \hookrightarrow v}$$

Figure 1: Typing and Evaluation rules for $\lambda^{\Box}$.

$$\frac{}{\Gamma \vdash^n * : 1} \qquad \frac{x{:}A^n \in \Gamma}{\Gamma \vdash^n x : A}$$

$$\frac{\Gamma, x{:}A^n \vdash^n e : B}{\Gamma \vdash^n \lambda x{:}A.\, e : A \to B} \qquad \frac{\Gamma \vdash^n e_1 : A \to B \qquad \Gamma \vdash^n e_2 : A}{\Gamma \vdash^n e_1\, e_2 : B}$$

$$\frac{\Gamma \vdash^{n+1} e : A}{\Gamma \vdash^n \mathbf{next}\ e : \bigcirc A} \qquad \frac{\Gamma \vdash^n e : \bigcirc A}{\Gamma \vdash^{n+1} \mathbf{prev}\ e : A}$$

Figure 2: Typing rules of $\lambda^{\bigcirc}$.

encode substitutions of closed code, but there is no way to perform substitutions of open code which is required in the `sqbox` example. The solution should be to extend the notion of code to include not only closed expressions, but also expressions which may contain free variables. The $\lambda^{\bigcirc}$-calculus [Dav96] provides some of the required flexibility.

$$\begin{array}{llcl}
\textit{Types} & A & ::= & 1 \mid A_1 \to A_2 \mid \bigcirc A \\
\textit{Terms} & e & ::= & x \mid * \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{next}\ e \mid \mathbf{prev}\ e \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A^n \\
\textit{Values} & v^0 & ::= & * \mid \lambda x{:}A.\ e \mid \mathbf{next}\ v^1 \\
 & v^{n+1} & ::= & * \mid \lambda x{:}A.\ v^{n+1} \mid v_1^{n+1}\ v_2^{n+1} \mid \mathbf{next}\ v^{n+2} \mid \mathbf{prev}\ v^n\ \ (n > 0)
\end{array}$$

The motivation behind $\lambda^{\bigcirc}$ is to ensure and maintain a temporal distinction between levels of computation, for the purposes of partial evaluation. So, for example, the computation marked to be on level 0 should be executed first in the scheme of partial evaluation; computation on level 1 is postponed and obtained as the residual of the level 0 evaluation, etc. The type system of $\lambda^{\bigcirc}$ allows the programmer to decorate terms with temporal annotations. Then the typechecking ensures that the program conforms to the level specifications, turning staging errors into type errors, just as in $\lambda^{\Box}$.

To mark that a certain term of type A is postponed, i.e. on the subsequent temporal code level, $\lambda^{\bigcirc}$ provides the modal type constructor $\bigcirc$, with its corresponding introduction term **next**. In contrast to $\lambda^{\Box}$, however, postponed terms in $\lambda^{\bigcirc}$ can contain free occurrences of variables from the the surrounding typing context, so long as they are marked to be on the same code level (see Figure 2). It is this property of the calculus that has associated the notion of *open* code with the type constructor $\bigcirc$, in contrast to the *closed* code of $\Box$ in $\lambda^{\Box}$.

Operationally, a postponed expression ($\mathbf{next}\ e$) is not generally a value, as ($\mathbf{box}\ e$) would be in $\lambda^{\Box}$. The $\lambda^{\bigcirc}$ calculus has separate evaluation relations for each code level; reductions may happen under **next** in $\mathbf{next}\ e$, albeit only at the subterms of e which are on lower code levels than e itself.

The elimination form for $\bigcirc$ is $\mathbf{prev}\ e$, where e is of code type. Intuitively, **prev** is used to compose code expressions, by splicing its argument into the surrounding term. More precisely, the operational behavior of $\mathbf{prev}\ e$ starts by evaluating e on the previous time level. If the previous is level 0, the reduction is bound to produce a postponed expression $\mathbf{next}\ e'$; then **prev** and **next** cancel each other, and e' is returned to be spliced into the surrounding term, which is itself of level 1. In cases of higher code levels, the evaluation of e may produce more general expressions, so no cancellation is prescribed by the operational semantics.

The exponentiation function in $\lambda^{\bigcirc}$ can be staged as the function `powercirc2` below. We hoist the helper function `powcirc'` outside of the main code for better readability.

$$\frac{}{\lambda x{:}A.\ e \overset{0}{\hookrightarrow} \lambda x{:}A.\ e} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x{:}A.\ e \quad e_2 \overset{0}{\hookrightarrow} v_2 \quad [v_2/x]e \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}$$

$$\frac{}{* \overset{n}{\hookrightarrow} *} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x{:}A.\ e \overset{n+1}{\hookrightarrow} \lambda x{:}A.\ v} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{next}\ e \overset{n}{\hookrightarrow} \mathbf{next}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \mathbf{next}\ v}{\mathbf{prev}\ e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{prev}\ e \overset{n+2}{\hookrightarrow} \mathbf{prev}\ v}$$

Figure 3: Operational Semantics of $\lambda^{\bigcirc}$.

```
powcirc' =
       fix pow':◯int->int->◯int.
           λe:◯int. λn:int.
               if n = 0 then next 1
               else
                   let val u = pow' e (n - 1)
                   in
                       next (prev e * prev u)
                   end

powcirc2 : int -> ◯(int->int) =
       λn:int.
           next λx:int. prev
               powcirc' (next x) n
```

Observe how the type of `powcirc2` indicates the staging of the function. Partial application of `powcirc2` to an integer n will produce a function for powering by n, of type `int->int`, but on the subsequent temporal level.

```
- sqcirc = powcirc2 2;
val sqcirc = next (λx:int. x * (x * 1)) : ◯(int->int)
```

The function `sqcirc` does not contain unnecessary redices as did `sqbox`. However, $\lambda^{\bigcirc}$ does not permit coercions of terms between different code levels, so it is not possible to demote the function `sqcirc` to code level 0 in order to evaluate it (the type system does not allow **prev** to occur on code level 0). This is only a sound behavior – operational semantics of level 0 proscribes a usual evaluation of the term, and this evaluation is defined only if the term in question is closed. Code values in $\lambda^{\bigcirc}$ *may* contain variables bound outside of them, so the type system, conservatively, forbids their evaluation.

2.3 *MetaML*

MetaML combines the notions of code from the two previous systems, so that programs can be manipulated in the style of $\lambda^{\bigcirc}$, but can also be made transcend their code levels (like code expressions of $\lambda^{\Box}$) and in particular be executed. To simplify the arguments here, we omit the MetaML features for coercing code expressions into higher code levels, and present only the fragment relevant to their execution. We follow [CMS01] in the exposition.

$$\frac{}{\Delta;\Gamma \vdash^n * : 1} \qquad \frac{x{:}A^n \in \Gamma}{\Delta;\Gamma \vdash^n x : A} \qquad \frac{u{:}A^n \in \Delta}{\Delta;\Gamma \vdash^n u : A}$$

$$\frac{\Delta;\Gamma, x{:}A^n \vdash^n e : B}{\Delta;\Gamma \vdash^n \lambda x{:}A.\, e : A \to B} \qquad \frac{\Delta;\Gamma \vdash^n e_1 : A \to B \qquad \Delta;\Gamma \vdash^n e_2 : A}{\Delta;\Gamma \vdash^n e_1\ e_2 : B}$$

$$\frac{\Delta;\Gamma \vdash^{n+1} e : A}{\Delta;\Gamma \vdash^n <e> : <A>} \qquad \frac{\Delta;\Gamma \vdash^n e : <A>}{\Delta;\Gamma \vdash^{n+1} {\sim}e : A}$$

$$\frac{\Delta;\Gamma \vdash^n e : T}{\Delta;\Gamma \vdash^n e : [T]} \qquad \frac{\Delta^{\leq n};\cdot \vdash^n e : A}{\Delta;\Gamma \vdash^n e : [A]} \qquad \frac{\Delta;\Gamma \vdash^n e : [A]}{\Delta;\Gamma \vdash^n e : A}$$

$$\frac{\Delta;\Gamma \vdash^n e_1 : T \qquad \Delta, u{:}T^n;\Gamma \vdash^n e_2 : A}{\Delta;\Gamma \vdash^n \mathbf{letc}\ u = e_1\ \mathbf{in}\ e_2 : A} \qquad \frac{\Delta;\Gamma \vdash^n e : [<A>]}{\Delta;\Gamma \vdash^n \mathbf{run}\ e : [A]}$$

Figure 4: Selected typing rules of MetaML.

$$\begin{array}{llcl}
\textit{Types} & A & ::= & 1 \mid A_1 \to A_2 \mid [A] \mid <A> \\
\textit{Closed types} & T & ::= & 1 \mid A \to T \mid [A] \\
\textit{Terms} & e & ::= & x \mid * \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid <e> \mid {\sim}e \mid \mathbf{run}\ e \mid \mathbf{letc}\ u = e_1\ \mathbf{in}\ e_2 \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A^n \\
\textit{Closed contexts} & \Delta & ::= & \cdot \mid \Delta, u{:}T^n \\
\textit{Values} & v^0 & ::= & * \mid \lambda x{:}A.\ e \mid <v^1> \\
 & v^{n+1} & ::= & * \mid \lambda x{:}A.\ v^{n+1} \mid v_1^{n+1}\ v_2^{n+1} \mid <v^{n+2}> \mid {\sim}v^n\ (n > 0) \mid \\
 & & & \mathbf{run}\ v^{n+1} \mid \mathbf{letc}\ u = v_1^{n+1}\ \mathbf{in}\ v_2^{n+1}
\end{array}$$

MetaML starts with the postponed/open code type of $\lambda^{\bigcirc}$, and introduces language constructs to refine the typing of those code instances which happen to be closed. The typing and operational semantics of MetaML are presented in Figures 4 and 5, respectively, and are similar to those of $\lambda^{\bigcirc}$, which they extend with the new term and type constructors.

The modal type constructor of MetaML is $<A>$ with introduction form $<e>$ and elimination form ${\sim}e$, corresponding to $\bigcirc A$, **next** e and **prev** e in $\lambda^{\bigcirc}$. However, MetaML adds a type refinement $[A]$ which classifies terms of type A with no free value variables. Note that it does not have corresponding term constructors. The typing judgment introduces implicit coercion from $[A]$ into A. The opposite coercion is also possible if the type A is "closed". Closed types are essentially those types whose values cannot refer to outside variables from higher code levels (e.g. base types, []-annotated code types, or functions with closed codomains). Expressions that can be assigned a code type which is closed in this sense, can be evaluated using the language constructor **run**.

Similarly to $\lambda^{\Box}$, MetaML splits the typing context into two parts: Γ for ordinary value variables, and Δ for variables of closed types. The ordinary value context is cleared when checking a term against a []-annotated type. A term constructor $\mathbf{letc}\ u = e_1\ \mathbf{in}\ e_2$ serves to introduce variables into the closed type context, so that they are not erased when typing closed code.

Our example with the exponentiation function can be coded in MetaML as shown below.

$$\frac{}{\lambda x{:}A.\ e \overset{0}{\hookrightarrow} \lambda x{:}A.\ e} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x{:}A.\ e \qquad e_2 \overset{0}{\hookrightarrow} v_2 \qquad [v_2/x]e \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}$$

$$\frac{}{* \overset{n}{\hookrightarrow} *} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x{:}A.\ e \overset{n+1}{\hookrightarrow} \lambda x{:}A.\ v} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \qquad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\texttt{<}e\texttt{>} \overset{n}{\hookrightarrow} \texttt{<}v\texttt{>}} \qquad \frac{e \overset{0}{\hookrightarrow} \texttt{<}v\texttt{>}}{\tilde{}\,e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\tilde{}\,e \overset{n+2}{\hookrightarrow} \tilde{}\,v}$$

$$\frac{e \overset{0}{\hookrightarrow} \texttt{<}v^1\texttt{>} \qquad v^1 \overset{0}{\hookrightarrow} v^0}{\mathbf{run}\ e \overset{0}{\hookrightarrow} v^0} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad [v_1/u]e_2 \overset{0}{\hookrightarrow} v}{\mathbf{letc}\ u = e_1\ \mathbf{in}\ e_2 \overset{0}{\hookrightarrow} v}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{run}\ e \overset{n+1}{\hookrightarrow} \mathbf{run}\ v} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \qquad e_2 \overset{n+1}{\hookrightarrow} v_2}{\mathbf{letc}\ u = e_1\ \mathbf{in}\ e_2 \overset{n+1}{\hookrightarrow} \mathbf{letc}\ u = v_1\ \mathbf{in}\ v_2}$$

Figure 5: Operational Semantics of (a selected fragment of) MetaML.

```
powmeta' =
       fix pow':[<int> -> int -> <int>].
           λe:<int>. λn:int.
               if n = 0 then <1>
               else
                   let val u = pow' e (n - 1)
                   in
                       <~e * ~u>
                   end

powmeta2 : int -> [<int->int>] =
       λn:int.
           letc n = n
                powmeta' = powmeta'
           in
                <λx:int. ~(powmeta' <x> n)>
           end
```

Notice how the variables `n` and `powmeta'` in the body of the function `powmeta2` are transferred into the closed type context by `letc`, so that they are not erased when the code expression `<λx:int. ~(powmeta' <x> n)>` is checked against the closed type `[<int->int>]`.

Application of `powmeta2` at argument 2 produces code containing a function for squaring.

```
- sqmeta = powmeta2 2;
val sqmeta = <λx:int. x * (x * 1)> : [<int->int>]
```

It can then be "ran" and applied itself.

```
- sq = run sqmeta;
val sq = [fn] : [int -> int]
- sq 3;
val it = 9 : int
```

2.4 *FreshML*

The FreshML language [PG00] has a slightly different flavor from the meta-languages presented before. It is not intended to manipulate open and closed code. Rather, its defining feature is its unique handling of representation and manipulation of abstract syntax trees of object languages with binding constructs. The syntax of the object language is represented via a user-defined datatype, like in ML, but the binding constructs of the object language are encoded modulo alpha-equivalence, using the concept of names and name abstraction.[3] Names (or symbols, atoms) are a separate semantic category which admits equality; there is a countably infinite supply of names, so that we can always pick another, fresh one, to place in a position of an object-level bound variable. Operation of name abstraction provides a mechanism to encode all the α-variants of a term with respect to the abstracted name.

The syntax of FreshML is given by the following context-free grammar. In the rules below, K is a finite set of names.

$$
\begin{array}{llcl}
\textit{Types} & A & ::= & atm \mid A_1 \to A_2 \mid [atm]A \\
\textit{Terms} & e & ::= & x \mid a \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
 & & & \mathbf{new}\ a\ \mathbf{in}\ e \mid a\,.\,e \mid e\,@\,a \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A\#K \\
\textit{Store} & S & ::= & \cdot \mid a, S \\
\textit{Values} & v & ::= & a \mid \lambda x{:}A.\ e \mid a\,.\,v
\end{array}
$$

Operationally, names can be though of as locations. They have their separate type atm, and are introduced into the computation through the combinator **new**; each introduced name is fresh/unique, i.e. different from all the previously introduced ones, and is bound in **new**. One can think of the expression

$$\mathbf{new}\ a\ \mathbf{in}\ e$$

as operationally analogous to ML's

$$\mathbf{let\ val}\ a = \mathbf{ref}\ ()\ \mathbf{in}\ e$$

except that the type system of FreshML imposes restrictions on the occurrences of a in e which are not required of references. In particular, names are not allowed to freely leave the scope of the defining **new**. If such a thing would happen, the escaped name will occur in the residual term, but the programmer will have no handle on it, and thus no capability to manipulate it in any way. Consider for example the (ill-typed) term

$$\mathbf{new}\ a\ \mathbf{in}\ \langle 1+1, a\rangle$$

This term is supposed to introduce a new name a into the run-time environment, then evaluate $1+1 \longmapsto 2$, and return $\langle 2, a\rangle$, which contains an occurrence of a about to escape the scope of its defining **new**. The way FreshML deals with this problem is to require that the reduct $\langle 2, a\rangle$, before returning, be coupled with a to form a closure, or name abstraction. The responsibility to create name abstractions is on the programmer, but the type system will check whether all the terms are appropriately closed when leaving a scope of **new**. Name abstraction construct of FreshML is $a\,.\,e$, and it has a corresponding type constructor $[atm](-)$. Thus, the term $\mathbf{new}\ a\ \mathbf{in}\ \langle 1+1, a\rangle$ is not well-typed in FreshML, but the terms $\mathbf{new}\ a\ \mathbf{in}\ a\,.\,\langle 1+1, a\rangle$ and $\mathbf{new}\ a\ \mathbf{in}\ \langle 1+1, a\,.\,a\rangle$ both are, with types $\mathbf{int} \times [atm]atm$ and $[atm](\mathbf{int} \times atm)$ respectively. Note that name abstraction is a binder, but the binding occurs only after the body of the abstraction has been evaluated.

The operation opposite from abstraction is concretion. Its syntax is $e\,@\,a$ and its operational semantics is to separate the term in the closure e from its abstracted name by *swapping* that name with a throughout the closure body (see the operational semantics in Figure 7). For example, if E denotes the term $E =$

[3] The same can be done with DeBruijn indices, but they have their drawbacks.

$$\frac{(x{:}A\#M) \in \Gamma \qquad K \subseteq M}{\Gamma \vdash x{:}A \;\#\; K} \qquad \frac{\Gamma, x{:}A \vdash e : B \;\#\; K \qquad \Gamma \vdash x_i{:}A_i \;\#\; K \quad \text{for all } x_i \in \mathbf{fv}(e) \setminus \{x\}}{\Gamma \vdash \lambda x{:}A.\, e : A \to B \;\#\; K}$$

$$\frac{\Gamma \vdash e_1 : A \to B \;\#\; K \qquad \Gamma \vdash e_2 : A \;\#\; K}{\Gamma \vdash e_1\; e_2 : B \;\#\; K} \qquad \frac{\Gamma \vdash e_1 : A \;\#\; M \qquad \Gamma, x{:}A\#M \vdash e_2 : B \;\#\; K}{\Gamma \vdash \textbf{let val } x = e_1 \textbf{ in } e_2 : B \;\#\; K}$$

$$\frac{\Gamma\#a, a{:}atm \vdash e : A \;\#\; (K \cup \{a\})}{\Gamma \vdash \textbf{new } a \textbf{ in } e : A \;\#\; K}$$

$$\frac{\Gamma \vdash e : A \;\#\; K \setminus \{a\} \qquad \Gamma(a) = atm}{\Gamma \vdash a \,.\, e : [atm]A \;\#\; K} \qquad \frac{\Gamma \vdash e : [atm]A \;\#\; (K \cup \{a\}) \qquad \Gamma \vdash a : atm \;\#\; K}{\Gamma \vdash e\,@\,a : A \;\#\; K}$$

Figure 6: Typing rules for for FreshML. Consult [PG00] for an explanation of the lambda rule.

$$\frac{}{S, a \vdash a \hookrightarrow a} \qquad \frac{}{S \vdash \lambda x{:}A.\, e \hookrightarrow \lambda x{:}A.\, e}$$

$$\frac{S \vdash e_1 \hookrightarrow \lambda x{:}A.\, e \qquad S \vdash e_2 \hookrightarrow v_2 \qquad S \vdash [v_2/x]e \hookrightarrow v}{S \vdash e_1\; e_2 \hookrightarrow v} \qquad \frac{S \vdash e_1 \hookrightarrow v_1 \qquad S \vdash [v_1/x]e_2 \hookrightarrow v}{S \vdash \textbf{let val } x = e_1 \textbf{ in } e_2 \hookrightarrow v}$$

$$\frac{S, a \vdash e \hookrightarrow v}{S \vdash \textbf{new } a \textbf{ in } e \hookrightarrow v} \qquad \frac{S \vdash e \hookrightarrow v}{S \vdash a \,.\, e \hookrightarrow a \,.\, v} \qquad \frac{S \vdash e \hookrightarrow a' \,.\, v' \qquad v = (a\; a')v'}{S \vdash e\,@\,a \hookrightarrow v}$$

Figure 7: Operational Semantics of FreshML. The operation $(a\; a')v'$ swaps the names a and a' in v'.

$(a \,.\, b \,.\, \langle 1+1, a\rangle)$, then $E\,@\,b$ evaluates to $a \,.\, \langle 1+1, b\rangle$. In a sense, the operation of name swapping is a simplification of the concept of alpha-renaming. When the name b is substituted for a in E, then name b in the second abstraction of E has to be renamed so as not to incur an accidental abstraction in the subsequent subterm. Since abstraction is intended to *hide* the abstracted name, it does not matter exactly which abstracting name is used (as long as it does not incur unintended abstractions), so FreshML might as well use the already available a.

As illustrated above, the FreshML mechanism of names makes a distinction between the notions of name introduction (construct **new**) and name abstraction (construct $a \,.\, e$), which is not the case in λ-calculi, where lambda expressions serve the role of both. The name introduction is an effectfull operation; every evaluation of a given term with name introduction results with a different name. But notice that due to the restrictions placed on the occurrences of names, these effects cannot be observed in the language itself, thus making FreshML purely functional.

The relevant set of rules of the typechecking and evaluation judgments of FreshML is presented in Figures 6 and 7. The typechecking judgment not only associates types to term, but it also keeps track of names occurring in the subterms and ensures that none of them leaves the scope of its defining **new** unguarded by a name abstraction. The judgment has the form $\Gamma \vdash e : A \;\#\; K$ and reads: in context Γ, the term e has type A, and the set of names K is fresh for e (i.e, no names from K will occur unguarded by name abstraction in the value of e). Notice that the context Γ contains not only typing, but also freshness annotations.

As an illustration we present an example in FreshML with an object language of untyped lambda calculus. The object language syntax is defined in ML-like datatype declaration, but it uses meta-level name abstraction to represent object-level lambda construct. The function `subst` takes an abstracted lambda term e, and another term t, and substitutes the term t in for the occurrences of the abstraction variable in e.

```
datatype lam = Var of atm
             | Lam of [atm]lam
             | App of lam * lam

fun subst e t : [atm]lam -> lam -> lam =
    new a in
      case e@a of
         Var a => t
       | Var e' => Var e'
       | Lam u =>
           new b in
              Lam b.(subst (a.(u @ b)) t)
       | App (e1, e2) =>
           (subst e1 t, subst e2 t)
```

The approach of FreshML towards representing binding constructs of the object languages is similar in design and goals to that of *higher order abstract syntax*, HOAS [PE88]. They both use a meta-level binding construct (name abstraction in FreshML, and lambda abstraction in HOAS) to encode the α-equivalence class of the object term. However, the notion of name abstraction in FreshML is weaker from HOAS in that it does not hardwire into the representation the capability to substitute for bound variables – substitution in FreshML has to be programmed by recursion on the structure of the object language terms. That way FreshML achieves the adequacy of representation, which can be lost in HOAS in the presence of additional term and type constructors (specifically, constructors for disjoint sums, as observed in [DPS97] in a logic programming setup). But, remark that even if the two approaches may differ in their abstraction mechanisms, they both will require some notion of names in order to recurse over the structure of the encoded term.

3 The Core Language

In this section we present the syntax and static semantics of our core language, deferring the exposition of intensional code analysis for Section 6. Taken in itself, the core language contains constructs for unifying the notions of closed and open code – a problem which motivated the earlier development behind the extension of $\lambda^{\bigcirc}$ into MetaML. However, our notion of code differs from that of the last two calculi. Both $\lambda^{\bigcirc}$ and MetaML allow code expressions to contain free variables, and in order to evaluate a code expression, MetaML has to prove it closed. Here, we adopt an opposite approach: each code expression is allowed to contain only those free variables which have been listed as *dependencies* in its type, and only the expressions with no dependencies are executable.

As a first development, we decided to disassociate the notion of "variables bound in lambda abstractions", from the notion of "free variables of a code expression", which are equated in $\lambda^{\bigcirc}$ and MetaML. The main reason is the following: intensional code analysis ought to provide a test whether two free variables in a code expression are different or equal. The result of this test is obviously not preserved under substitution, so it looks questionable to tie the free code variables to outside lambda abstractions. This is not to say that it is impossible to add intensional code analysis to $\lambda^{\bigcirc}$ or MetaML. Indeed, the distinction between code levels on which the code comparison and the substitution occur gives some leeway to believe that this setup can be given a sound operational semantics. Code analysis is a meta level operation, substitution of lambda bound variables in code expressions is an object level operation. The first is immediately executable, the second is postponed, so the two will never interfere. However, having them both tied to the same mechanism of variable binding will almost certainly disturb the theory of the underlying core language (in particular, the beta rule [Tah99a], but also the parametricity of λ-abstraction may be in question).

This argument motivates the addition of a second binding mechanism which will be used to create and abstract free variables of code expressions. But, we want to compare our free variables for intensional equality, and thus need to resort to *names* (see for example [Ode94]). Additionally, we opt to separate the operation of name creation (hiding of a name), from the name abstraction (renameability), because that seems to provide strictly more expressiveness in manipulation of code and names, than if the two are combined into a single constructor. This is where we employ the mechanisms of Nominal Logic and FreshML, which were designed with exactly that purpose in mind (see [Pit01] and [PG00]). We introduce a new semantic category of *names* which are to stand for free variables in boxed expressions. Thus, boxed expression, as before in $\lambda^{\Box}$, cannot contain free variables, but we allow them to contain *names*, under the provision that the occurring free names are listed in the type of the expression. Correspondingly, the boxed types are now of the form $\Box(A[C])$ where C stands for a finite set of names and name parameters (to be explained later) that the boxed term may depend on. The syntax of our language is presented below.

Simple types	P	$::=$	$1 \mid P_1 \to P_2$
Types	A	$::=$	$1 \mid A_1 \to A_2 \mid \Box(A[C]) \mid \underset{a:P}{\text{И}} A \mid \forall p\#K.\, A$
Terms	e	$::=$	$x \mid a \mid * \mid \lambda x{:}A.\, e \mid e_1\, e_2 \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \mid$
			$a\,.\,e \mid e\,@\,a \mid \mathbf{new}\ a{:}P\ \mathbf{in}\ e \mid \Lambda p\#K.\, e \mid e\,[\![C]\!] \mid \{a \doteq e_1\}\ e_2 \mid$
			$\mathbf{fix}\ x{:}A.\, e$
Variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A \mid \Gamma, t{\div}A[C] \mid \Gamma, u{::}A[C]$
Name contexts	S	$::=$	$\cdot \mid S, a{:}P$
Parameter contexts	Δ	$::=$	$\cdot \mid \Delta, p\#K$

Before explaining the term constructors, let us first dispense with the conventions we will assume in the rest of the text. Similarly to $\lambda^{\Box}$, our language makes a distinction between ordinary (value) variables and expression variables. We further distinguish between expression variables and expressions that have empty name dependencies, and those that may depend on some name; the first kind can be executed, and the second cannot. In analogy with Kripke semantics for Modal Logic, we will also call the first kind *reflexive* or *reflectable*, and the second kind *nonreflexive* or *nonreflectable* variables and expressions. Thus, a variable context Γ may contain three forms of variable typings: $x{:}A$ for value variables, and $u{::}A[C]$ and $t{\div}A[C]$ for, respectively, reflexive and nonreflexive expression variables with name dependency C. We call the type A with a dependency C an *annotated type*. Here, we use A, B and variants to vary over arbitrary types, P to vary over simple types, a, b, c and the variants to vary over names; x, y, z stand for ordinary variables and u, v and t for expression variables. Further, we use p and q for unknown (i.e. parametric) sets of names, K, M, L for finite sets of names, and C, D for name dependencies, i.e. for finite sets containing both names and name-set parameters. The parameter context Δ associates parameters with *disjointness annotations*. For example, $p\#K \in \Delta$ would mean that the parameter p stands for an unknown dependency set C such that: (1) C contains no names from the set K, and (2) if $q \in C$ is a parameter, then $q\#K \in \Delta$, too. Enlarging an appropriate context by a new variable, name or parameter, is subject to Barendregt's Variable Convention: the new variables are assumed distinct, or are renamed in order not to clash with already existing ones. Terms which differ only in names of their bound variables or parameters are considered equal. But notice that *types and terms with abstracted names are subject to more complicated rules*, which will be explained with the typing judgment. As usual, capture avoiding substitution is defined to rename variables, parameters and names when descending into their scope. Free parameters of a type A are denoted by $\mathbf{fp}(A)$, free variables of a term e by $\mathbf{fv}(e)$, and its free names are $\mathbf{fn}(e)$.

We are now ready to describe various new term and type constructors that our language introduces. Similarly to FreshML, we have a term construct $a\,.\,e$ for name abstraction and $e\,@\,a$ for name concretion. However, the type of name abstraction in our case has to be more general than the $[atm]A$ of FreshML. Since names take part as dependency annotations in types, the name abstraction type has to be a binder $\underset{a:P}{\text{И}} A$, abstracting the occurrences of the name a in the type A. For example, assuming for a moment that our language has a type constructor for pairs, the terms

$$\mathbf{new}\ a{:}P\ \mathbf{in}\ a\,.\,\langle 1+1, \mathbf{box}\ a\rangle \quad \text{and} \quad \mathbf{new}\ a{:}P\ \mathbf{in}\ \langle 1+1, a\,.\,\mathbf{box}\ a\rangle$$

would both be well-typed with types $\mathbf{int} \times \underset{a:P}{\text{И}} \Box(P[a])$ and $\underset{a:P}{\text{И}}(\mathbf{int} \times \Box(P[a]))$, respectively. The quantifier

$И$ has already been investigated in [Gab00] and [Pit01], but it has not been used explicitly in the definition of FreshML.

Yet further difference between our names and FreshML is that we do not have a separate type of names, but rather can assign them an arbitrary (simple) type P. As a consequence, names cannot be values in the operational semantics we intended for the language, because it would not make sense to compute with them. For example, if n is as a *name* of integer type, the operation $n + 1$ cannot be evaluated unless n is provided with a definition. Thus, while FreshML names can be thought of as references, our names are *null references*, i.e. references without an extension. We must impose that the they not occur without definition on the level 0 – they find their use only when placed in a code expression (i.e. under a box) to stand for some piece of code, or as placeholders for free variables, where they can be used for intensional code analysis.

The limitation to simple types may be somewhat arbitrary. We felt compelled to first understand the language better before we extend it and generalize, and the simply-typed fragment is the most obvious choice for a first attempt at intensional code analysis. The unfortunate consequence (at least for now) is that it becomes impossible for code expressions to contain free names of code types, and thus our language, at least when open code expressions are concerned, is a two-level, rather than multi-level language like $\lambda^{\bigcirc}$ and MetaML. However, even with this limitation the language has enough expressive power to encode many, if not all, interesting examples from meta-programming practice.

Another feature we consider is explicit name polymorphism. A program may want to manipulate code expressions no matter what their name dependencies are, or code expressions whose name dependencies are unknown at compile time. A typical example would be any recursive function which scans over a boxed term. When it encounters a lambda expression, it has to place a *fresh* name instead of the bound variable, and recursively continue scanning the body of the lambda, which is itself a boxed expression, but depending on this newly introduced name. For such uses, our language has a term construct $\Lambda p\#K.\ e$ of type $\forall p\#K.\ A$ which is a polymorphic abstraction of an unknown set of name dependencies p disjoint from a set of names K. When K is empty, we abbreviate the constructs into $\Lambda p.\ e$ and $\forall p.\ A$. The term $e\ [\![C]\!]$ is the polymorphic instantiation, substituting a name dependency set C for a bound parameter in e.

Finally, we have a term construct for name substitution $\{a \doteq e_1\}\ e_2$. It substitutes the value of e_1 for name a in all the occurrences of a in e_2 on the *current code level*. In particular, the substitution will not take place under boxes. This is consistent with the notion of preservation of code levels; eventual free variables from the environment of e_1 should not be permitted to creep under other boxes. This construct also gives us a way to provide *extensions*, i.e. definitions for names, while still using names for the *intensional* information of their identity.

Example 1
To illustrate our language constructs and motivate the further development, we present a version of the staged exponentiation function that we could write in our system. In this example we assume that the language is extended with the base type of integers.

```
pow' =
    fix pow':∀p. □(int[p])->int->□(int[p]).
        Λp. λe:□(int[p]). λn:int.
           if n = 0 then box 1
           else
             let box e1 = pow' ⟦p⟧ e (n - 1)
                 box e2 = e
             in
                 box (e1 * e2)
             end
```

```
pow : int -> □(int -> int) =
    λn:int.
       new a:int in
          let box e = pow' ⟦a⟧ (box a) n
          in
              box (λx:int. {a = x} e)
          end

- sqcode = pow 2;
val sqcode =
   box (λx:int. x * (x * 1)):□(int->int)
```

The function `pow` takes an integer `n` and generates an integer name `a`. Then it calls the helper function `pow'` to build the term $e = \mathbf{box}\ (\underbrace{a * \cdots * a}_{n} * 1)$ of type $\Box$(`int[a]`). Finally, it substitutes the name `a` with a newly introduced bound variable `x`, before returning. The helper function `pow'` is name-polymorphic; its name parameter p is instantiated with the relevant dependency set as part of the application.

Notice that the generated residual code for `sqcode` does not contain any unnecessary redices, in contrast to the $\lambda^{\Box}$ version of the program from Section 2.1.

■

3.1 Auxiliary judgments

In order to state the typechecking rules, we need a couple of auxiliary judgments. We start with the judgment for *disjointness* (interchangeably referred to as *freshness*) of name dependencies. It has the form

$$\Delta \vdash C \ \# \ K$$

where Δ is a context storing parameters with their freshness annotations, C is a name dependency set, and K is a set of names. The judgment is satisfied if none of the names from K appears in C, and if all the parameters from C are declared fresh for K in the context Δ.

$$\frac{}{\Delta \vdash \cdot \ \# \ K} \qquad \frac{\Delta \vdash C \ \# \ K \qquad a \notin K}{\Delta \vdash a, C \ \# \ K} \qquad \frac{\Delta \vdash C \ \# \ K \qquad p \# M \in \Delta \qquad K \subseteq M}{\Delta \vdash p, C \ \# \ K}$$

Figure 8: Disjointness judgment for name dependencies.

We extend the concept of disjointness for name dependencies to disjointness for types and annotated types. We will use the same notation for all three of them, as the distinction will always be clear from the context. The two new, mutually recursive judgments have the form

$$\Delta \vdash A \ \# \ a \qquad \text{and} \qquad \Delta \vdash A[C] \ \# \ a$$

where Δ is a context of parameters and their freshness annotations, $A[C]$ is an *annotated type*, and a is a name. The judgments are satisfied if a does not appear free in the type A, nor in the parameters of type A, and the name dependency C is disjoint from the name set $\{a\}$, as determined by the previously defined judgment for disjointness of dependencies. Observe that the rule for disjointness of name abstraction types requires that the bound name c is different from the name we test against. That can always be achieved by alpha-renaming the bound name c to some fresh name, as described by Barendregt's Variable Convention. The rule for universally quantified types inserts the parameter p into Δ, but with the freshness annotation

Type disjointness

$$\frac{}{\Delta \vdash 1 \;\#\; a} \qquad \frac{\Delta \vdash A \;\#\; a \qquad \Delta \vdash B \;\#\; a}{\Delta \vdash (A \to B) \;\#\; a} \qquad \frac{\Delta \vdash A[C] \;\#\; a}{\Delta \vdash \Box(A[C]) \;\#\; a}$$

$$\frac{\Delta \vdash A \;\#\; a \qquad a \neq c}{\Delta \vdash (\underset{c:P}{\text{И}} A) \;\#\; a} \qquad \frac{\Delta, p\#(K,a) \vdash A \;\#\; a \qquad a \notin K}{\Delta \vdash (\forall p\#K.\, A) \;\#\; a}$$

Annotated Type disjointness

$$\frac{\Delta \vdash A \;\#\; a \qquad \Delta \vdash C \;\#\; a}{\Delta \vdash A[C] \;\#\; a}$$

Figure 9: Disjointness judgments for types and annotated types.

(K, a), rather than just K. The idea is that a name a is disjoint/fresh from a type A if it does not occur in A, and it is fresh for all the free parameters of A. To preserve this interpretation, new parameters must be introduced into the context with extended freshness annotation.

We also require a judgment to decide if a given type A is well-formed in the name context S and parameter context Δ, i.e. whether all the free names and parameters of A are declared in S or Δ, respectively. The judgment reads

$$S; \Delta \vdash A \ \text{wf}$$

and its rules are defined below. Notice that only the names of particular symbols, and not their types of freshness annotations are relevant for the judgment.

$$\frac{}{S; \Delta \vdash 1 \ \text{wf}} \qquad \frac{S; \Delta \vdash A \ \text{wf} \qquad S; \Delta \vdash B \ \text{wf}}{S; \Delta \vdash A \to B \ \text{wf}} \qquad \frac{S, a{:}P; \Delta \vdash A \ \text{wf}}{S; \Delta \vdash (\underset{a:P}{\text{И}} A) \ \text{wf}}$$

$$\frac{S; \Delta \vdash A \ \text{wf} \qquad C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Delta \vdash \Box(A[C]) \ \text{wf}} \qquad \frac{K \subseteq \mathbf{dom}(S) \qquad S; \Delta, p\#K \vdash A \ \text{wf}}{S; \Delta \vdash \forall p\#K.\, A \ \text{wf}}$$

Figure 10: Well-formedness judgment for types.

We also need to define weakening on types: if a type contains a certain set of name dependencies, we can always pass it as a type with a superset of dependencies instead. As in the previous judgments, here too it may be needed to alpha-rename the bound names when comparing two И-types. It can always be done by choosing a fresh name c.

Finally, two types A and B are *equivalent* if $A \leqslant: B$ and $B \leqslant: A$. In the future text, we will implicitly equate types which are equivalent. It is justified by the fact that two types are equivalent iff they differ only in the ordering of names and parameters in their name dependencies. But name dependencies are considered sets, so this ordering should not matter.

The following lemmas lead to establishment of parametricity properties of the typing judgment. This, in

$$\frac{}{1 \leqslant: 1} \qquad \frac{B_1 \leqslant: A_1 \qquad A_2 \leqslant: B_2}{A_1 \to A_2 \leqslant: B_1 \to B_2} \qquad \frac{A \leqslant: B}{\underset{c:P}{\text{И}}\, A \leqslant: \underset{c:P}{\text{И}}\, B}$$

$$\frac{A \leqslant: B \qquad C \subseteq D}{\Box(A[C]) \leqslant: \Box(B[D])} \qquad \frac{A \leqslant: B \qquad K \subseteq M}{\forall p\#K.\, A \leqslant: \forall p\#M.\, B}$$

Figure 11: Subtyping judgment.

turn, would be instrumental in proving the progress and type preservation for name-polymorphic instantiation and name concretion constructs of our language.

Lemma 1

1. *if* $\Delta \vdash C \;\#\; M$ *and* $K \subseteq M$*, then* $\Delta \vdash C \;\#\; K$.
2. $\Delta \vdash C_1 \;\#\; K$ *and* $\Delta \vdash C_2 \;\#\; K$ *if and only if* $\Delta \vdash (C_1 \cup C_2) \;\#\; K$.

Proof: In each case, by a straightforward induction on the cardinality of the appropriate dependency set. ■

Lemma 2 (Substitution and auxiliary judgments)

1. *if* $\Delta, p\#K \vdash C \;\#\; M$ *and* $\Delta \vdash D \;\#\; K$*, then* $\Delta \vdash ([D/p]C) \;\#\; M$,
2. *if* $\Delta, p\#K \vdash A[C] \;\#\; a$ *and* $\Delta \vdash D \;\#\; K$*, then* $\Delta \vdash ([D/p]A)[[D/p]C] \;\#\; a$,
3. *if* $S; \Delta, p\#K \vdash A$ *wf, and* $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$*, and* $\Delta \vdash D \;\#\; K$*, then* $S; \Delta \vdash ([D/p]A)$ *wf,*
4. *if* $C_1 \subseteq C_2$*, then* $[D/p]C_1 \subseteq [D/p]C_2$,
5. *if* $A \leqslant: B$*, then* $[D/p]A \leqslant: [D/p]B$.

Proof: Straightforward induction on the derivation of the appropriate relation, using Lemma 1. ■

Lemma 3
If $\Delta, p\#K \vdash \mathbf{fp}(A) \;\#\; a$ *and* $\Delta \vdash D \;\#\; K$*, then* $\Delta \vdash \mathbf{fp}([D/p]A) \;\#\; a$.

Proof: We first show that $\mathbf{fp}([D/p]A) \subseteq [D/p]\,\mathbf{fp}(A)$. The case when $p \notin \mathbf{fp}(A)$ is trivial, so we assume that $p \in \mathbf{fp}(A)$. Then $\mathbf{fp}([D/p]A) = (\mathbf{fp}(A) \setminus \{p\}) \cup \mathbf{fp}(D)$ and $[D/p]\,\mathbf{fp}(A) = (\mathbf{fp}(A) \setminus \{p\}) \cup D$. But, $\mathbf{fp}(D) \subseteq D$, and thus $\mathbf{fp}([D/p]A) \subseteq [D/p]\,\mathbf{fp}(A)$. Now, instantiating Lemma 2.1, with $C = \mathbf{fp}(A)$ and $M = \{a\}$, we obtain $\Delta \vdash ([D/p]\,\mathbf{fp}(A)) \;\#\; a$. From here, using Lemma 1.2 and the just established inequality, we obtain the required $\Delta \vdash \mathbf{fp}([D/p]A) \;\#\; a$. ■

3.2 The Type System

The typing judgment of our language has the form

$$S; \Gamma \vdash_\Delta e : A[C]$$

It reads: in the presence of name context S, variable context Γ and parameter context Δ, the term e has type A with name dependency C. As customary, we presuppose that all involved contexts are well-formed. In particular, all the variables, names and parameters are distinct, and all their types are well-formed. The

Constants, variables and names

$$\frac{C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta * : 1\,[C]} \qquad \frac{x{:}A \in \Gamma \qquad C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta x : A\,[C]}$$

$$\frac{u{::}A[C] \in \Gamma \qquad C \subseteq D \qquad D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta u : A\,[D]} \qquad \frac{a{:}P \in S \qquad C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta a : P\,[a, C]}$$

λ-calculus and recursion

$$\frac{S; \Delta \vdash A \ \text{wf} \qquad S; \Gamma, x{:}A \vdash_\Delta e : B\,[C]}{S; \Gamma \vdash_\Delta \lambda x{:}A.\, e : A \to B\,[C]} \qquad \frac{S; \Gamma \vdash_\Delta e_1 : A \to B\,[C] \qquad S; \Gamma \vdash_\Delta e_2 : A\,[C]}{S; \Gamma \vdash_\Delta e_1\ e_2 : B\,[C]}$$

$$\frac{S; \Delta \vdash A \ \text{wf} \qquad S; \Gamma, x{:}A \vdash_\Delta e : A\,[C]}{S; \Gamma \vdash_\Delta \mathbf{fix}\ x{:}A.\, e : A\,[C]}$$

Modality

$$\frac{S; \Gamma^\triangledown \vdash_\Delta e : A\,[C] \qquad D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta \mathbf{box}\ e : \Box(A\,[C])\,[D]}$$

$$\frac{S; \Gamma \vdash_\Delta e_1 : \Box(A[\,])\,[C] \qquad S; \Gamma, u{::}A \vdash_\Delta e_2 : B\,[C]}{S; \Gamma \vdash_\Delta \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B\,[C]}$$

$$\frac{S; \Gamma \vdash_\Delta e_1 : \Box(A[D])\,[C] \qquad S; \Gamma, t{\div}A[D] \vdash_\Delta e_2 : B\,[C] \qquad D \neq \emptyset}{S; \Gamma \vdash_\Delta \mathbf{let\ box}\ t = e_1\ \mathbf{in}\ e_2 : B\,[C]}$$

Figure 12: Typing rules of the core language (Part 1).

name dependency C deserves further explanation, because it goes to the hart of our treatment of names. As already hinted, we do not operationally treat names as values. If an expression contains an occurrence of a name, it cannot be evaluated. Rather, names are employed as placeholders for free variables in code expressions (i.e., under a box), where there will be no attempts to evaluate them, since evaluation of boxed expressions is postponed. In this sense, one may say that a dependency annotation of some term *is any set of names such that the term can be safely evaluated once all of the listed names are provided with extensions.* Thus, if C is a name dependency for a given term e in some context, any set $D \supseteq C$ which is well-formed with respect to the context, is a valid name dependency for e as well. We will build this idea into the type system by allowing weakening (i.e. enlarging) of name dependencies at specific typing rules.

Before proceeding further, we define an operation $\Gamma^\triangledown$ on variable contexts. It erases the ordinary variables

Names

$$\frac{S, a{:}P; \Gamma \vdash_\Delta e : A\,[C] \qquad \Delta \vdash \mathbf{fp}(A) \ \#\ a}{S, a{:}P; \Gamma \vdash_\Delta a\,.\,e : (\mathsf{И}_{a:P} A)\,[C]} \qquad \frac{S, a{:}P; \Gamma \vdash_\Delta e : (\mathsf{И}_{a:P} A)\,[C] \qquad \Delta \vdash \mathbf{fp}(A) \ \#\ a}{S, a{:}P; \Gamma \vdash_\Delta e \mathbin{@} a : A\,[C]}$$

$$\frac{S, a{:}P; \Gamma \vdash_{\Delta\#a} e : A\,[C] \qquad \Delta\#a \vdash A[C] \ \#\ a}{S; \Gamma \vdash_\Delta \mathbf{new}\ a{:}P\ \mathbf{in}\ e : A\,[C]}$$

Name polymorphism

$$\frac{S; \Gamma \vdash_{\Delta,p\#K} e : A\,[C]}{S; \Gamma \vdash_\Delta \Lambda p\#K.\, e : \forall p\#K.\, A\,[C]}$$

$$\frac{S; \Gamma \vdash_\Delta e : \forall p\#K.\, A\,[C] \qquad \Delta \vdash D \ \#\ K \qquad D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \vdash_\Delta e\,[\![D]\!] : ([D/p]A)\,[C]}$$

Name substitutions

$$\frac{S, a{:}P; \Gamma \vdash_\Delta e_1 : P\,[C] \qquad S, a{:}P; \Gamma \vdash_\Delta e_2 : B\,[a, C]}{S, a{:}P; \Gamma \vdash_\Delta \{a \doteq e_1\}\ e_2 : B\,[C]}$$

Subtyping

$$\frac{S; \Gamma \vdash_\Delta e : A\,[C] \qquad A \leqslant: B}{S; \Gamma \vdash_\Delta e : B\,[C]}$$

Figure 13: Typing rules of the core language (Part 2).

from Γ and changes nonreflexive expression hypothesis $t{\div}A$ into reflexive ones $t{::}A$.

$$\begin{array}{rcl}
(\cdot)^\triangledown & = & \cdot \\
(\Gamma, x{:}A)^\triangledown & = & \Gamma^\triangledown \\
(\Gamma, t{\div}A[C])^\triangledown & = & \Gamma^\triangledown, t{::}A[C] \\
(\Gamma, u{::}A[C])^\triangledown & = & \Gamma^\triangledown, u{::}A[C]
\end{array}$$

The typing rules of our language are presented in Figures 12 and 13. In the following text, we try to explain some of the intricacies, decisions and interdependencies behind the design of the type system.

Consider first the rule for λ-abstraction. Notice that it relies on one of the auxiliary judgments from the previous section to check whether the type A of the bound variable is well-formed. That is because types can mention names and parameters, and it has to ensure that the ones actually occurring in A have already been declared in the name context S and the parameter context Δ, before the hypothesis $x{:}A$ is placed into the variable context. The synthesized type B does not have to be checked for well-formedness, because the typing rules guarantee it, provided all the contexts are well-formed themselves.

Next consider the rule for box. Similarly to $\lambda^{\Box}$, it checks the boxed expression e against a variable context $\Gamma^{\triangledown}$ from which the value variables have been erased. In addition $\Gamma^{\triangledown}$ changes the status of all the nonreflexive expression variables into reflexive ones. This is because the free expression variables in e are already enclosed by a box; their occurrences are not for purpose of evaluation/reflection, but rather for composing pieces of code into a larger one. Since boxing suspends evaluation, the term can be assigned any well-formed dependency set D.

Observe that we have two different typings for the **let box** expression: one handles reflexive expressions, i.e. expressions with empty name dependencies, and the other is for the nonreflexive ones. Alternatively, we could have introduced two syntactically different constructs.

The construct **new** generates a fresh name, and then checks, using the auxiliary disjointness judgment, if the synthesized type and name dependency do not contain free occurrences of this new name. The operation $\Delta\#a$ extends with a the freshness annotation of every parameter in Δ. This is only sound, since a is a new name. It is necessary in order to type possible abstractions with name a in the body of **new**. The operation is defined recursively as:

$$\begin{aligned}(\cdot)\#a &= \cdot \\ (\Delta, p\#K)\#a &= (\Delta\#a), p\#(K, a)\end{aligned}$$

As explained before, the name abstraction construct $a\,.\,e$ creates a closure with a of the *value* of e, that way abstracting the eventual occurrences of a in it. Notice, however, that the side condition $\Delta \vdash \mathbf{fp}(A)\ \#\ a$ in the typing judgment is crucial. It ensures that the parameters occurring in e could not be substituted with a set containing the name a. If that were possible, the new occurrence of a would be abstracted on the level of terms, but there would be no binding in the corresponding type, thus causing unsound behavior. The reason for that is that the quantifier in $\underset{a:p}{\text{И}}\ A$ is itself a binder, and two name abstraction types which differ only in the names of bound atoms, are considered equal.

For example, consider the following term, ill-typed in the presence of the side-condition on the typing rule for name abstraction.

```
new a:int in
   let val F = Λp. λx:□(int[p]). (a.x)
   in
       F ⟦a⟧ (box a)
   end
```

The term assigned to F is typed as $\Box(int[p]) \rightarrow \underset{a:int}{\text{И}}\ \Box(int[p])$, but because the quantifier И is a binder, this type is the same as $\Box(int[p]) \rightarrow \underset{c:int}{\text{И}}\ \Box(int[p])$, for some fresh name $c \neq a$. The two types, however, even though supposedly equal, behave differently under name-polymorphic instantiation. In the first case, the term $F\ [\![a]\!]$ receives the type $\Box(int[a]) \rightarrow \underset{a:int}{\text{И}}\ \Box(int[a])$, while in the second case, it is typed as $\Box(int[a]) \rightarrow \underset{c:int}{\text{И}}\ \Box(int[a])$. Resorting for a moment to the still undefined operational semantics of our language, the whole term $F\ [\![a]\!]$ (**box** a) is supposed to evaluate to $a\,.$ (**box** a), and hence only the first, but not the second typing above will be sound.

The side-condition $\Delta \vdash \mathbf{fp}(A)\ \#\ a$ on the name abstraction and concretion rules is imposed exactly to avoid this kind of problematic behavior. It prevents the parameter p in our previous example to be instantiated with the name set $\{a\}$, so that the name abstraction on the level of terms will be reflected by a sound И-abstraction on the level of types. A correctly typed equivalent of the above code would then be the following.

```
new a:int in
  let val F = Λp#a. λx:□(int[p,a]). (a.x)
  in
      F ⟦a⟧ (box a)
  end
```

As a yet another peculiarity of the typing rule for name abstraction, observe that that it does not change in any way the name dependency C of the involved term; in particular, it does not attempt to remove the abstracted name from it. This is justified by the intended interpretation of name abstraction $(a \,.\, e)$: it first *evaluates* its body e before creating the closure with a. Thus, the set of names that need to be provided with definitions in order to evaluate $(a \,.\, e)$ is the same set required for the evaluation of e itself. In other words, the two expressions have the same dependency set. Similar considerations motivate the typing rule for concretion as well.

On a related note, the above example is also illustrative of some flexibilities of our language (compared to FreshML) which result from having name dependencies be part of types, rather than just be part of the typing judgment. Namely, in FreshML, the function $\lambda x.\ (a \,.\, x)$ must be typed as dependent on a (i.e. a is not fresh for it), even though a is always fresh for its body $(a \,.\, x)$. In other words, as already remarked in [PG00], freshness is not a logical relation, and that forces a somewhat unjustified typing rule for lambda abstractions (see Figure 6). Obviously, a system like ours, which incorporates name dependencies into types, will remedy that, and will be able to give a more precise typing to $\lambda x.\ (a \,.\, x)$. For example (as already shown) we can express that the result of applying this function will always be disjoint from a, independently of the status with respect to a of the application argument.

Returning to the typing rules of our core language, the last rule we consider is that for subtyping. Due to the nature of name dependencies which can be arbitrarily weakened, we need an explicit subtyping rule to coerce types into types of same structure but larger name dependencies on various code levels.

4 Theory

This section explores the theoretical properties of our type system, which will be used to justify the operational semantics we ascribe to it, and ultimately prove the Progress and Type Preservation theorems of our language. We begin with the basic:

Lemma 4 (Structural Properties of Contexts)

1. **Exchange** *If* S_1, Γ_1, Δ_1, C_1 *are obtained by permuting the elements of* S_2, Γ_2, Δ_2, C_2 *respectively, and* $S_1; \Gamma_1 \vdash_{\Delta_1} e : B\,[C_1]$ *then* $S_2; \Gamma_2 \vdash_{\Delta_2} e : B\,[C_2]$.

2. **Weakening** *If* S_1, Γ_1, Δ_1, C_1 *are subsets or equal to* S_2, Γ_2, Δ_2, C_2 *respectively, and* $S_1; \Gamma_1 \vdash_{\Delta_1} e : B\,[C_1]$ *then* $S_2; \Gamma_2 \vdash_{\Delta_2} e : B\,[C_2]$.

3. **Variable Contraction** *Let* $\star$ *stand for any of* $:$, $::$, *or* $\div$ *variable typings. Then* $S; \Gamma, x{\star}A[D], y{\star}A[D] \vdash_\Delta e : B\,[C]$ *implies* $S; \Gamma, x \star A[D] \vdash_\Delta [x/y]e : B\,[C]$.

4. **Name for Variable Substitution** *If* $S; \Gamma, x{:}P \vdash_\Delta e : A\,[C]$ *and* $a \notin \mathbf{dom}(S)$, *then* $S, a{:}P; \Gamma \vdash_\Delta [a/x]e : A\,[C, a]$

Proof: By straightforward induction on the structure of the typing derivation, using similarly formulated exchange, weakening and contraction properties of the auxiliary judgments. ■

Next step is to define two new operations on contexts, $\Gamma^\ominus$ and $\Gamma^\vartriangle$, which, together with the already defined $\Gamma^\triangledown$, will be important for stating the substitution principles for our language. $\Gamma^\ominus$ removes the ordinary value variables from Γ, leaving only expression variables in it. $\Gamma^\vartriangle$ changes the reflexive expression variables with nonempty name dependencies into nonreflexive ones.

$$
\begin{array}{rcl}
(\cdot)^\ominus & = & \cdot \\
(\Gamma, x{:}A)^\ominus & = & \Gamma^\ominus \\
(\Gamma, t{\div}A[C])^\ominus & = & \Gamma^\ominus, t{\div}A[C] \\
(\Gamma, u{::}A[C])^\ominus & = & \Gamma^\ominus, u{::}A[C]
\end{array}
\qquad
\begin{array}{rcll}
(\cdot)^\vartriangle & = & \cdot & \\
(\Gamma, x{:}A)^\vartriangle & = & \Gamma^\vartriangle, x{:}A & \\
(\Gamma, t{\div}A[C])^\vartriangle & = & \Gamma^\vartriangle, t{\div}A[C] & \\
(\Gamma, u{::}A[\emptyset])^\vartriangle & = & \Gamma^\vartriangle, u{::}A[\emptyset] & \\
(\Gamma, u{::}A[C])^\vartriangle & = & \Gamma^\vartriangle, u{\div}A[C] & \text{if } C \neq \emptyset
\end{array}
$$

Before we formulate and prove the substitution principles, we need a couple of intermediate steps. The next lemma states that a context with nonreflexive variables is weaker than a corresponding context in which these variables are given typing. Every term that can be typed in the first context can also be typed in the second.

Lemma 5
If $S;\Gamma_1^{\ominus},\Gamma_2 \vdash_\Delta e : A\,[C]$, *then* $S;\Gamma_1^{\nabla},\Gamma_2 \vdash_\Delta e : A\,[C]$.

Proof: By induction on the typing derivation of e. We present only the case when $e = \mathbf{box}\ e'$ and $A = \Box(A'[C'])$.

(1) By typing derivation, $S;\Gamma_1^{\ominus\nabla},\Gamma_2^{\nabla} \vdash_\Delta e' : A'\,[C']$.

(2) Because $\Gamma_1^{\ominus\nabla} = \Gamma_1^{\nabla\nabla\ominus}$, we have $S;\Gamma_1^{\nabla\nabla\ominus},\Gamma_2^{\nabla} \vdash_\Delta e' : A'\,[C']$.

(3) By induction hypothesis, $S;\Gamma_1^{\nabla\nabla},\Gamma_2^{\nabla} \vdash_\Delta e' : A'\,[C']$.

(4) Now by typing rule for **box**, $S;\Gamma_1^{\nabla},\Gamma_2 \vdash_\Delta \mathbf{box}\ e' : \Box(A'[C'])\,[C]$.

∎

We also require another meta operation – that of name substitution $\{a/e\}e'$. It differs from the usual variable substitution in the fact that it substitutes the name a by e only on the current code level in e'; the occurrences of a on higher code levels (i.e., under boxes) will not be touched. This operation and its corresponding substitution principles will be used to justify the operational semantics of the term construct for name substitution $\{a \doteq e\}\ e'$.[4] The operation is capture avoiding – all the variables, atoms and parameters are renamed when descending into the scope of the term construct which created them.

Definition 6 (Name Substitution)
Given terms e *and* e' *and a name* a, *the operation* $\{e/a\}e'$ *of substituting* e *for name* a *in* e' *is defined recursively over the structure of* e' *as follows.*

$$
\begin{array}{rcl}
\{e/a\}* & = & * \\
\{e/a\}x & = & x \\
\{e/a\}u & = & u \\
\{e/a\}a & = & e \\
\{e/a\}b & = & b \qquad (a \neq b) \\
\{e/a\}(\lambda x{:}A.\ e_1) & = & \lambda x{:}A.\ \{e/a\}e_1 \\
\{e/a\}(\mathbf{fix}\ x{:}A.\ e_1) & = & \mathbf{fix}\ x{:}A.\ \{e/a\}e_1 \\
\{e/a\}(e_1\ e_2) & = & (\{e/a\}e_1\ )(\{e/a\}e_2) \\
\{e/a\}(\mathbf{box}\ e_1) & = & \mathbf{box}\ e_1 \\
\{e/a\}(\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let\ box}\ u = \{e/a\}e_1\ \mathbf{in}\ \{e/a\}e_2 \\
\{e/a\}(b\,.\,e_1) & = & b\,.\,\{e/a\}e_1 \qquad \textit{(a may be equal to b)} \\
\{e/a\}(e_1 \,@\, b) & = & (\{e/a\}e_1) \,@\, b \qquad \textit{(a may be equal to b)} \\
\{e/a\}(\mathbf{new}\ b{:}Q\ \mathbf{in}\ e_1) & = & \mathbf{new}\ b{:}Q\ \mathbf{in}\ \{e/a\}e_1 \qquad (a \neq b) \\
\{e/a\}(\Lambda p\#K.\ e_1) & = & \Lambda p\#K.\ \{e/a\}e_1 \\
\{e/a\}(e_1\ [\![D]\!]) & = & (\{e/a\}e_1)\ [\![D]\!] \\
\{e/a\}(\{a \doteq e_1\}\ e_2) & = & \{a \doteq \{e/a\}e_1\}\ e_2 \\
\{e/a\}(\{b \doteq e_1\}\ e_2) & = & \{b \doteq \{e/a\}e_1\}\ (\{e/a\}e_2) \qquad (a \neq b)
\end{array}
$$

In the similar spirit, we define the set $\mathbf{fn}_0(e)$ of names occurring in the term e on the current code level. This gives us a way to compute the minimal set of names which must appear as a name dependency when typing the term e.

[4] Notice the difference in the notation.

Definition 7
Given a term e, the set $\mathbf{fn}_0(e)$ is defined recursively by the equations below. We refer to $\mathbf{fn}_0(e)$ as the set of free names *of the term e.*

$$
\begin{array}{rcl}
\mathbf{fn}_0(a) & = & \{a\} \\
\mathbf{fn}_0(*) & = & \emptyset \\
\mathbf{fn}_0(x) & = & \emptyset \\
\mathbf{fn}_0(u) & = & \emptyset \\
\mathbf{fn}_0(\lambda x{:}A.\, e) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(\mathbf{fix}\ x{:}A.\, e) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(e_1\ e_2) & = & \mathbf{fn}_0(e_1) \cup \mathbf{fn}_0(e_2) \\
\mathbf{fn}_0(\mathbf{box}\ e) & = & \emptyset \\
\mathbf{fn}_0(\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) & = & \mathbf{fn}_0(e_1) \cup \mathbf{fn}_0(e_2) \\
\mathbf{fn}_0(a\,.\,e) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(e\ @\ a) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(\mathbf{new}\ a{:}P\ \mathbf{in}\ e) & = & \mathbf{fn}_0(e) \setminus \{a\} \\
\mathbf{fn}_0(\Lambda p\#K.\, e) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(e\ [\![C]\!]) & = & \mathbf{fn}_0(e) \\
\mathbf{fn}_0(\{a \doteq e_1\}\ e_2) & = & \mathbf{fn}_0(e_1) \cup (\mathbf{fn}_0(e_2) \setminus \{a\})
\end{array}
$$

Notice that the free names of a name abstraction $a\,.\,e$ do not exclude a. The reason is closely related to the already stated property that name abstraction abstracts only after the body of the abstraction e is evaluated. Thus, if the name a occurs on the present code level in e, we need to provide an extension for a before e is ran, and hence e depends on a. Similar comment applies to name concretion and to name-polymorphic instantiation.

The following definition is adopted from Nominal Logic and FreshML [PG00].

Definition 8 (Name Transposition)
The operation of interchanging all the occurrences of names a and b in the argument name dependency/context/type/expression, is called name transposition *or* name swapping, *and is denoted by $(a\ b)(-)$.*

Name transposition is different from name substitution: the former swaps two names throughout the given term or type, no matter the code level on which any of the names occur, while the later only works on the current code level.

Lemma 9 (Strengthening)

1. *if $S; \Gamma, x{:}A \vdash_\Delta e : B\,[C]$ and $x \notin \mathbf{fv}(e)$, then $S; \Gamma \vdash_\Delta e : B\,[C]$*
2. *if $S; \Gamma, u{::}A[D] \vdash_\Delta e : B\,[C]$ and $u \notin \mathbf{fv}(e)$, then $S; \Gamma \vdash_\Delta e : B\,[C]$*
3. *if $S; \Gamma, t{\div}A[D] \vdash_\Delta e : B\,[C]$ and $t \notin \mathbf{fv}(e)$, then $S; \Gamma \vdash_\Delta e : B\,[C]$*
4. *if $S; \Gamma^\Delta \vdash_\Delta e : A\,[C, a]$ and $a \notin \mathbf{fn}_0(e)$ then $S; \Gamma^\Delta \vdash_\Delta e : A\,[C]$*
5. *if $S; \Gamma^\Delta \vdash_\Delta e : A\,[C, p]$ then $S; \Gamma^\Delta \vdash_\Delta e : A\,[C]$*

Proof: In each case, by a straightforward induction on the first typing derivation. ■

Lemma 10 (Substitution Principles)

1. *if $S; \Gamma \vdash_\Delta e_1 : A\,[C]$ and $S; \Gamma, x{:}A \vdash_\Delta e_2 : B\,[C]$, then $S; \Gamma \vdash_\Delta [e_1/x]e_2 : B\,[C]$.*
2. *if $S; \Gamma_1^\ominus \vdash_\Delta e_1 : A\,[D]$ and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_2 : B[C]$, then $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e_2 : B[C]$.*

3. *if* $S; \Gamma_1^{\triangledown} \vdash_\Delta e_1 : A\,[D]$ *and* $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_2 : B[C]$, *then* $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B[C]$.

4. *if* $S, a{:}P; \Gamma_1 \vdash_\Delta e_1 : P[C]$ *and* $S, a{:}P; \Gamma_2^{\triangle} \vdash_\Delta e_2 : B[a, C]$, *then* $S, a{:}P; \Gamma_1, \Gamma_2^{\triangle} \vdash_\Delta \{e_1/a\}e_2 : B[C]$.

The premises in the formulation of the substitution principles deserve further elaboration. Principle 10.2 requires that the substituted term e_1 is typable in a context $\Gamma_1^{\ominus}$, i.e. that it does not contain any free value variables. The intuition behind this is that e_1 substitutes an *expression* variable u. Expression variables may occur on multiple code levels, so the substitution will copy e_1 to multiple code levels too. But the ordinary value variables are anchored by the type system to only the current code level, and thus e_1 must contain none of them.

Principle 10.3 requires that the substituting term e_1 be typed in a context $\Gamma_1^{\triangledown}$, which contains no value variables, and in which all the nonreflexive expression variables are turned into reflexive ones. The reasons for the first requirement is analogous to the one in the previous principle. The second requirement is actually a weakening of the context, since turning a nonreflexive variable into a reflexive one allows more terms to be typed (because now the variable can be used on the current code level as well). It is justified because e_1 substitutes a *nonreflexive expression variable* t. The variable t only occurs guarded by a box, i.e. on code levels strictly higher than the current one. Thus any typing of e_1 in the term $[e_1/t]e_2$ will happen in a context in which the nonreflexive expression variables relevant for e_1 have already been turned into reflexive ones by the typing rule for **box** (see Figure 12).

Finally, Principle 10.4 requires that the context in both the second premise and in the conclusion be of special form $\Gamma_2^{\triangle}$, i.e. that its reflexive variables only have empty name dependency. Note that the principle describes a way to reduce the name dependency of a term e_2 by substituting away the name a. But, the way the operation of name substitution is defined, it may not necessarily change the expression e_2 itself. For example, consider the case when $e_2 = u$ in the context $\Gamma = u{::}A[a]$. The substitution $\{a/e_1\}u$ produces a term u itself, but there is no typing $S; \Gamma \vdash_\Delta u : A\,[\,]$. That is why we require that the involved reflexive variables have no dependencies. In retrospect, the need to distinguish between expression variables with and without dependencies, which arises from this principle, was the main reason why we introduced nonreflexive variables into the design of the type system at all, instead of staying with only the reflexive variables of $\lambda^{\Box}$.

Another observation of crucial importance is that the local variables of a boxed expression form a context Γ, which is exactly of the form the name substitution principle 10.4 requires, i.e. $\Gamma = \Gamma^{\triangle}$. This can easily be seen, as all the reflexive variables which will be put into the context have empty name dependencies (see the typing rules for **let box** in Figure 12. This would allow us to use the meta operation of name substitution $\{e_1/a\}e_2$ to define the operational semantics of the language construct for name substitution $\{e_1 \doteq a\}\ e_2$. The idea is to use this construct to perform substitutions within box-annotated expression, and the principle 10.4 ensures that these substitutions can be carried out without the postponement of evaluation which is the usual operational semantics associated with boxed expressions in $\lambda^{\Box}$.

Proof: All the substitution principles are proved by induction on the typing derivation for e_2. We present below some of the more interesting cases. The complete proof can be found in the Appendix.

Principle 2. if $S; \Gamma_1^{\ominus} \vdash_\Delta e_1 : A\,[D]$ and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_2 : B[C]$, then $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e_2 : B[C]$.

case $e_2 = \textbf{box}\ e$, where $B = \Box(B'[C'])$.

1. By typing derivation, $S; \Gamma_2^{\triangledown}, u{::}A[D] \vdash_\Delta e : B'\,[C']$.
2. Because $\Gamma_1^{\ominus} = \Gamma_1^{\ominus\ominus}$, we have $S; \Gamma_1^{\ominus\ominus} \vdash_\Delta e_1 : A\,[D]$.
3. From (1), (2) and the induction hypothesis, $S; \Gamma_1^{\ominus}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/x]e : B'\,[C']$.
4. By Lemma 5, $S; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/x]e : B'\,[C']$,
5. and finally, we can reassemble $S; \Gamma_1, \Gamma_2 \vdash_\Delta \textbf{box}\ ([e_1/x]e) : \Box(B'[C'])\,[C]$,

Principle 3. if $S; \Gamma_1^{\triangledown} \vdash_\Delta e_1 : A\,[D]$ and $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_2 : B[C]$, then $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B[C]$.

case $e_2 = \textbf{box}\ e'$, where $B = \Box(B'[C'])$.

1. By typing derivation, $S; \Gamma_2^{\triangledown}, t{::}A[D] \vdash_\Delta e' : B' [C']$.
2. Because $\Gamma_1^{\triangledown} = \Gamma_1^{\triangledown\ominus}$, by the previously proved substitution principle (Lemma 10.2), $S; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/t]e' : B' [C']$.
3. Now assemble back into $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{box}\ ([e_1/t]e') : \Box(B'[C'])\,[C]$.

Principle 4. if $S, a{:}P; \Gamma_1 \vdash_\Delta e_1 : P[C]$ and $S, a{:}P; \Gamma_2^{\vartriangle} \vdash_\Delta e_2 : B[a, C]$, then $S, a{:}P; \Gamma_1, \Gamma_2^{\vartriangle} \vdash_\Delta \{e_1/a\}e_2 : B[C]$.

case $e_2 = u$.

By the definition of $\Gamma_2^{\vartriangle}$, it is only possible that the variable $u \in \mathbf{dom}(\Gamma_2^{\vartriangle})$ if its name annotation is empty, i. e. if $u{::}B[\emptyset] \in \Gamma_2^{\vartriangle}$. Thus, by name dependency weakening, $S, a{:}P; \Gamma_2^{\vartriangle} \vdash_\Delta u : B\,[C]$. Now, by hypothesis weakening, and because $u = \{e_1/a\}u$, we get the required $S, a{:}P; \Gamma_1, \Gamma_2^{\vartriangle} \vdash_\Delta \{e_1/a\}u : B\,[C]$.

case $e_2 = \mathbf{box}\ e'$, and $B = \Box(B'[C'])$.

In this case, the typing $\mathbf{box}\ e' : B\,[a, C]$ is obtained by weakening inherent in the box rule. Thus, we can also derive $S, a{:}P; \Gamma_2^{\vartriangle} \vdash_\Delta \mathbf{box}\ e' : \Box(B'[C'])\,[C]$. Considering that $\{e_1/a\}\mathbf{box}\ e' = \mathbf{box}\ e'$, weaken the hypothesis context to get $S, a{:}P; \Gamma_1, \Gamma_2^{\vartriangle} \vdash_\Delta \{e_1/a\}\mathbf{box}\ e' : \Box(B'[C'])\,[C]$.

■

Lemma 11 (Parametricity)

1. if $S; \Gamma \vdash_{\Delta, p\#K} e : A\,[C]$ and D is a well-formed dependency set, i.e. $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$, and is fresh for K, i.e. $\Delta \vdash D \;\#\; K$, then

$$S; [D/p]\Gamma \vdash_\Delta [D/p]e : ([D/p]A)\,[[D/p]C]$$

2. if $S; \Gamma \vdash_\Delta e : A\,[C]$, and $a, b{:}P$ are names (not necessarily in S), then

$$(a\ b)S; (a\ b)\Gamma \vdash_{(a\ b)\Delta} (a\ b)e : (a\ b)A\,[(a\ b)C]$$

Proof: First notice that the transposition property (property 2) is trivial to prove by induction on the typing derivation for e. Namely, all the typing rules, as well as the rules for auxiliary judgments are are obviously insensitive to swapping the names throughout, in all the contexts, types, terms and dependencies. Thus the judgment itself must be insensitive to swapping names.

The proof of the first property is somewhat less trivial, but still rather straightforward by induction on the typing derivation of e. We present here only the cases for name abstraction and name-polymorphic instantiation, and leave the rest for the Appendix.

case $e = a\,.\,e'$, where $A = \underset{a:P}{\text{И}}\, A'$ and $a{:}P \in S$.

Assume $a \notin D$ to ensure capture avoiding. This can always be achieved by alpha-renaming a into some other fresh name.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e' : A'\,[C]$ and $\Delta, p\#K \vdash \mathbf{fp}(A') \;\#\; a$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_\Delta ([D/p]A')\,[[D/p]C]$.
3. By Lemma 3, $\Delta \vdash \mathbf{fp}([D/p]A') \;\#\; a$.
4. Assemble back into $S; [D/p]\Gamma \vdash_\Delta (a\,.\,[D/p]e') : (\underset{a:P}{\text{И}}([D/p]A'))\,[[D/p]C]$.

case $e = e'\ [\![D']\!]$.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e' : (\forall q\#M.\ A')\,[C]$ where $\Delta, p\#K \vdash D' \;\#\; M$ and $D' \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta, p\#K)$, and $A = [D'/q]A'$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_\Delta [D/p]e' : (\forall q\#M.\ [D/p]A')\,[[D/p]C]$.

3. By Lemma 2.1, $\Delta \vdash ([D/p]D') \ \# \ M$.
4. Next, obviously, $([D/p]D') \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
5. Thus conclude, $S; [D/p]\Gamma \vdash_\Delta ([D/p]e') \, [\![[D/p]D']\!] : ([D/p]A) \, [[D/p]C]$.

∎

5 Operational Semantics

In this section we define the structured operational semantics for our core language, and prove the appropriate Progress and Type Preservation theorem. We start by introducing the notion of *contraction*, which will be instrumental in defining the *values* of our language. The idea is that we do not consider, like in $\lambda^\square$, that all boxed expressions are values. Rather, in order to be values, boxed expressions have to be "contracted", i.e. not reduced completely, but only freed of (some) name substitution they may contain. The name substitutions that are carried out (i.e. contracted) under a box in a given expression satisfy two properties: (1) They occur on the current code level. This is in accord with the previously made observation about the substitution principle 10.4 that the variable context Γ of variables encountered when traversing the current code level of a boxed term, *and not descending into further and further boxes*, is always of a form $\Gamma = \Gamma^\Delta$. Thus, the said substitution principle is applicable, and the encountered name substitutions can actually be carried out without postponing. (2) The substituted name should be created outside of the boxed term, rather than being local to it.

The judgment for contraction is defined in Figure 14. It has the form

$$e \xrightarrow{S} w$$

and means: if the name substitutions in the expression e of names *other than those* in S are carried out, we obtain w. The "protected" set S carries the locally defined names of e (see the contraction rule for **new**), and is introduced in order to comply with the requirement (2) from above. An expression e is *S-contracted* if $e \xrightarrow{S} e$. It is *contracted* if it is $\emptyset$-contracted. We use the letter w to range over S-contracted expressions.

Lemma 12 (Contraction Termination and Type Preservation)
If $S_1, S_2; \Gamma^\Delta \vdash_\Delta e : A \, [C]$ then there exists unique term w, such that $e \xrightarrow{S_2} w$. Furthermore, w is S_2-contracted and $S_1, S_2; \Gamma^\Delta \vdash_\Delta w : A \, [C]$.

Proof: By induction on the derivation $S_1, S_2; \Gamma^\Delta \vdash_\Delta e : A \, [C]$. The full proof is in the Appendix.

case $e = \mathbf{let\ box}\ u = e'\ \mathbf{in}\ e''$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash e' : \square(A'[C']) \, [C]$, and,
2. either $S_1, S_2; \Gamma^\Delta, u{::}A' \vdash e' : \square(A'[C']) \, [C]$, or $S_1, S_2; \Gamma^\Delta, u{\div}A'[C'] \vdash e' : \square(A'[C']) \, [C]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. Also notice that $(\Gamma^\Delta, u{::}A') = (\Gamma, u{::}A')^\Delta$, and if $C' \neq \emptyset$, then $(\Gamma^\Delta, u{\div}A'[C']) = (\Gamma, u{\div}A'[C'])^\Delta$.
4. Then, by induction hypothesis, we have w' and w'' satisfying the prescribed properties. Combine them into $w = \mathbf{let\ box}\ u = w'\ \mathbf{in}\ w''$.

case $e = \{a \doteq e'\} \, e''$, where $a{:}P \in S_1, S_2$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash_\Delta e' : P \, [C]$, and $S_1, S_2; \Gamma^\Delta \vdash_\Delta e'' : A \, [a, C]$.
2. By induction hypothesis, there are unique w' and w'' such that $e' \xrightarrow{S_2} w'$ and $e'' \xrightarrow{S_2} w''$, plus they are contracted and preserve the types.
3. Now, distinguish two cases: (1) $a \in \mathbf{dom}(S_2)$, and (2) $a \notin \mathbf{dom}(S_2)$.
4. In the first case, pick $w = \{a \doteq w'\} \, w''$. It is contracted and has the correct typing.

$$\frac{}{x \xrightarrow{S} x} \qquad \frac{}{u \xrightarrow{S} u} \qquad \frac{}{a \xrightarrow{S} a} \qquad \frac{}{* \xrightarrow{S} *}$$

$$\frac{e \xrightarrow{S} w}{\lambda x.\, e \xrightarrow{S} \lambda x.\, w} \qquad \frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2}{e_1\ e_2 \xrightarrow{S} w_1\ w_2} \qquad \frac{e \xrightarrow{S} w}{\mathbf{fix}\ x.\, e \xrightarrow{S} \mathbf{fix}\ x.\, w}$$

$$\frac{}{\mathbf{box}\ e \xrightarrow{S} \mathbf{box}\ e} \qquad \frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2}{\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \xrightarrow{S} \mathbf{let\ box}\ u = w_1\ \mathbf{in}\ w_2}$$

$$\frac{e \xrightarrow{S} w}{a\,.\,e \xrightarrow{S} a\,.\,w} \qquad \frac{e \xrightarrow{S} w}{e\,@\,a \xrightarrow{S} w\,@\,a} \qquad \frac{e \xrightarrow{S, a:P} w}{\mathbf{new}\ a{:}P\ \mathbf{in}\ e \xrightarrow{S} \mathbf{new}\ a{:}P\ \mathbf{in}\ w}$$

$$\frac{e \xrightarrow{S} w}{\Lambda p\#K.\, e \xrightarrow{S} \Lambda p\#K.\, w} \qquad \frac{e \xrightarrow{S} w}{e\,[\![C]\!] \xrightarrow{S} w\,[\![C]\!]}$$

$$\frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2 \qquad a \in \mathbf{dom}(S)}{\{a \doteq e_1\}\ e_2 \xrightarrow{S} \{a \doteq w_1\}\ w_2} \qquad \frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2 \qquad a \notin \mathbf{dom}(S)}{\{a \doteq e_1\}\ e_2 \xrightarrow{S} \{w_1/a\}w_2}$$

Figure 14: Contraction rules for expressions.

5. In the second case, pick $w = \{w'/a\}w''$. By the contraction rules, $e \xrightarrow{S_2} w$. By Lemma 13.1, it is contracted. By substitution principle (Lemma 10.4), it also has the correct typing $S_1, S_2; \Gamma^{\Delta} \vdash_{\Delta} \{w'/a\}w'' : A\,[C]$.

∎

Lemma 13 (Substitution and Transposition of Contracted Expressions)

1. *If w_1 and w_2 are S-contracted, then $\{w_1/a\}w_2$ is S-contracted.*

2. *If w is S-contracted and $a, b \notin S$, then $(a\ b)w$ is S-contracted.*

Proof:

1. By induction on the derivation $w_2 \xrightarrow{S} w_2$. Base cases are $w_2 = x, u, a, *, \mathbf{box}\ e$ and they clearly satisfy the requirements of the lemma. The rest of the induction cases are also easy. The most interesting is when $w_2 = \{a \doteq w_1'\}\ w_2'$. In that case, w_1' and w_2' are S-contracted and $a \in \mathbf{dom}(S)$. By induction hypothesis, $\{w_1/a\}w_1'$ is also S-contracted, and so $\{w_1/a\}w_2 = \{a \doteq \{w_1/a\}w_1'\}\ w_2'$ must be too.

2. By a straightforward induction on the derivation $w \xrightarrow{S} w$. In case $w = \mathbf{new}\ c{:}P\ \mathbf{in}\ w'$ (we assume by Barendregt's Variable Convention that $c \neq a, b$), then w' is $(S, c{:}P)$-contracted. By induction hypothesis, so is $(a\ b)w'$, and the conclusion follows.

∎

We can now define our syntactic category of values.

$$v \quad ::= \quad * \mid \lambda x.\, e \mid a\,.\,v \mid \Lambda p\#K.\, e \mid \mathbf{box}\ w \quad (w \text{ contracted})$$

$$\frac{\mu, e_1 \longmapsto \mu', e_1'}{\mu, (e_1\ e_2) \longmapsto \mu', (e_1'\ e_2)} \qquad \frac{\mu, e_2 \longmapsto \mu', e_2'}{\mu, (v_1\ e_2) \longmapsto \mu', (v_1\ e_2')}$$

$$\frac{}{\mu, ((\lambda x{:}A.\ e)\ v) \longmapsto \mu, [v/x]e} \qquad \frac{}{\mu, \mathbf{fix}\ x{:}A.\ e \longmapsto \mu, [\mathbf{fix}\ x{:}A.\ e/x]e}$$

$$\frac{e \rightarrow w \qquad e \text{ not contracted}}{\mu, \mathbf{box}\ e \longmapsto \mu, \mathbf{box}\ w} \qquad \frac{\mu, e_1 \longmapsto \mu', e_1'}{\mu, (\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) \longmapsto \mu', (\mathbf{let\ box}\ u = e_1'\ \mathbf{in}\ e_2)}$$

$$\frac{}{\mu, (\mathbf{let\ box}\ u = \mathbf{box}\ w\ \mathbf{in}\ e_2) \longmapsto \mu, [w/u]e_2} \qquad \frac{}{\mu, (\mathbf{new}\ a{:}P\ \mathbf{in}\ e) \longmapsto (\mu, a), e}$$

$$\frac{\mu, e \longmapsto \mu', e'}{\mu, (a\ .\ e) \longmapsto \mu', (a\ .\ e')} \qquad \frac{\mu, e \longmapsto \mu', e'}{\mu, (e \mathbin{@} a) \longmapsto \mu', (e' \mathbin{@} a)} \qquad \frac{}{\mu, (b\ .\ v) \mathbin{@} a \longmapsto \mu, (a\ b)v}$$

$$\frac{\mu, e \longmapsto \mu', e'}{\mu, (e\ [\![C]\!]) \longmapsto \mu', (e'\ [\![C]\!])} \qquad \frac{}{\mu, ((\Lambda p\#K.\ e')\ [\![C]\!]) \longmapsto \mu, [C/p]e'}$$

$$\frac{\mu, e_1 \longmapsto \mu', e_1'}{\mu, (\{a \doteq e_1\}\ e_2) \longmapsto \mu', (\{a \doteq e_1'\}\ e_2)} \qquad \frac{}{\mu, (\{a \doteq v\}\ e_2) \longmapsto \mu, \{v/a\}e_2}$$

Figure 15: Structured operational semantics of the core language.

Lemma 14 (Name Transposition Preserves Values)
If an expression v is syntactically a value, as defined by the above grammar, in the name context S containing names $a, b{:}P \in S$, then $(a\ b)v$ is also a value.

Proof: By induction on the structure of v. The only interesting case is when $v = \mathbf{box}\ w$. Then $(a\ b)v = \mathbf{box}\ (a\ b)w$. But, by Lemma 13.2, $(a\ b)w$ is contracted, and hence $\mathbf{box}\ (a\ b)w$ is a value. ∎

At last, we are in position to define a small-step operational semantics (see Figure 15), and prove the Type Preservation and Progress theorem for the core part of the language. We use μ and variants to denote a store of names created during the evaluation. A store of names corresponding to a name context S will be denoted by μ_S. Note that the theorem requires empty variable and parameter contexts and name dependency.

Theorem 15 (Progress and Type Preservation)
If $S; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists $S' \supseteq S$ such that $\mu_S, e \longmapsto \mu_{S'}, e'$; furthermore e' is unique and $S'; \cdot \vdash e' : A\,[\,]$.*

Proof: By induction on the derivation $S; \cdot \vdash e : A\,[\,]$. We present the more important cases below. The rest can be found in the Appendix.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$.

Assume that e_1 is a value (otherwise trivial). In that case $e_1 = \mathbf{box}\ w_1$, where w_1 is contracted, and $\mu_S, e \longmapsto \mu_S, [w_1/u]e_2$.

1. By typing derivation, $S; \cdot \vdash w_1 : A' [C']$, and,
2. either $S; u{::}A' \vdash e_2 : A [\,]$, or $S; u{\div}A'[C'] \vdash e_2 : A [\,]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. Conclude, using either Lemma 10.2, or Lemma 10.3, that $S; \cdot \vdash [w_1/u]e_2 : A [\,]$.

case $e = e' \,@\, a$, where $a{:}P \in S$.

Assume that e' is a value (otherwise, trivial).

1. By typing derivation, $S; \cdot \vdash e' : (\underset{a:P}{\text{И}} A) [\,]$, and $\cdot \vdash \mathbf{fp}(A) \ \# \ a$ (because $\mathbf{fp}(A) = \emptyset$).
2. Then $e' = b \,.\, v'$, where $b{:}P \in S$, and $S; \cdot \vdash v' : (a\ b)A [\,]$.
3. By reduction rules, $\mu_S, e \longmapsto \mu_S, (a\ b)v'$.
4. By Lemma 11.2, $(a\ b)S; \cdot \vdash (a\ b)v' : (a\ b)(a\ b)A [\,]$.
5. Now, both $a, b \in \mathbf{dom}(S)$, so $(a\ b)S = S$.
6. By idempotency of swapping, $S; \cdot \vdash (a\ b)v' : A [\,]$, and the typing is preserved.

case $e = e' \ [\![D]\!]$.

Assume e' is a value (otherwise trivial).

1. By typing derivation, $S; \cdot \vdash e' : \forall p \# K.\ A'$, where $\vdash D \ \# \ K$, and $A = [D/p]A'$.
2. Since e' is a value, it is of the form $e' = \Lambda p \# M.\ e''$, where $M \subseteq K$ and thus $\vdash D \ \# \ M$.
3. By typing rules, $S; \cdot \vdash_{p \# M} e'' : A' [\,]$.
4. By Lemma 11.1, $S; \cdot \vdash [D/p]e'' : ([D/p]A') [\,]$,
5. Since $\mu_S, e \longmapsto \mu_S, [D/p]e''$, we have just shown that the typing is preserved.

case $e = \{a \doteq e_1\} \ e_2$, where $a{:}P \in S$.

Assume both e_1 is a value (otherwise trivial).

1. By typing derivation, $S; \cdot \vdash e_1 : P [\,]$, and $S; \cdot \vdash e_2 : A [a]$.
2. By Lemma 10.4, $S; \cdot \vdash \{e_1/a\}e_2 : A [\,]$.
3. By reduction rules, $\mu_S, e \longmapsto \mu_S, \{e_1/a\}e_2$, so the statement is proved.

■

6 Intensional Code Analysis

This section presents the definition and theory of pattern-matching on code expressions, which is used to inspect the structure of an object program and destruct it into its component parts. For the purposes of this work, we limit ourselves to intensional analysis of only the simply typed λ-calculus fragment of our language. Thus, admittedly, our current results are far from complete, but nevertheless, we present them here as a first step towards a stronger and more robust system.

$$\textit{Patterns} \quad \pi \quad ::= \quad [E\ x_1 \cdots x_n] \mid x \mid a \mid * \mid \lambda x{:}P.\ \pi \mid (\pi_1)\ (\pi_2{:}P) \mid \mathbf{fix}\ x{:}P.\ \pi \mid \pi{:}P[C]$$

The pattern $[E\ x_1 \cdots x_n]$ declares a pattern variable E which matches a code expression subject to condition that the expression's free variables are among $x_1, \ldots, x_n$. We will denote pattern variables with capital E and its variants. Patterns $\lambda x{:}P.\ \pi$ and $\mathbf{fix}\ x{:}P.\ \pi$, match respectively a lambda expression and a fixpoint expression of domain type P. They declare a variable x which is local to the pattern, and demand that the body of the matched expression conforms to the pattern π. Bound variables, like x above, are to be distinguished from pattern variables, like $[E\ x_1 \cdots x_n]$. The later provides a placeholder for the matching process; upon execution of a successful matching, it will be bound to a certain expression. The former is just

$$\frac{x_i{:}P_i \in \Gamma \qquad C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \Vdash_\Delta [E\ x_1 \ldots x_n] : P\,[C] \Longrightarrow E{:}\ \underset{a_1:P_1}{\text{И}} \cdots \underset{a_n:P_n}{\text{И}} \Box(P\,[C, a_1, \ldots, a_n])}$$

$$\frac{S; \Gamma \Vdash_\Delta \pi : P\,[D] \Longrightarrow \Gamma_1 \qquad D \subseteq C}{S; \Gamma \Vdash_\Delta (\pi{:}P[D]) : P\,[C] \Longrightarrow \Gamma_1}$$

$$\frac{C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma, x{:}P \Vdash_\Delta x : P\,[C] \Longrightarrow \cdot} \qquad \frac{C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S, a{:}P; \Gamma \Vdash_\Delta a : P\,[a, C] \Longrightarrow \cdot} \qquad \frac{C \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)}{S; \Gamma \Vdash_\Delta * : 1\,[C] \Longrightarrow \cdot}$$

$$\frac{S; \Gamma, x{:}P_1 \Vdash_\Delta \pi : P_2\,[C] \Longrightarrow \Gamma_1}{S; \Gamma \Vdash_\Delta \lambda x{:}P_1.\ \pi : P_1 \to P_2\,[C] \Longrightarrow \Gamma_1} \qquad \frac{S; \Gamma, x{:}P \Vdash_\Delta \pi : P\,[C] \Longrightarrow \Gamma_1}{S; \Gamma \Vdash_\Delta \mathbf{fix}\ x{:}P.\ \pi : P\,[C] \Longrightarrow \Gamma_1}$$

$$\frac{S; \Gamma \Vdash_\Delta \pi_1 : P_2 \to P\,[C] \Longrightarrow \Gamma_1 \qquad S; \Gamma \Vdash_\Delta \pi_2 : P_2\,[C] \Longrightarrow \Gamma_2}{S; \Gamma \Vdash_\Delta (\pi_1)\ (\pi_2 : P_2) : P\,[C] \Longrightarrow \Gamma_1, \Gamma_2}$$

Figure 16: Typing rules for patterns.

a syntactic constant, which is introduced by a pattern for lambda expressions, and can match only itself. Pattern a matches a name a from the global name context. Pattern $(\pi_1)(\pi_2{:}P)$ matches an application; in order to avoid polymorphic types in patterns, we require that the this pattern proscribes the exact type of the argument in the application. The pattern $\pi{:}P[C]$ serves to specifically limit the allowed dependencies of the matched expression to only C.

The judgment for typechecking patterns has the form

$$S; \Gamma \Vdash_\Delta \pi : P\,[C] \Longrightarrow \Gamma_1$$

and reads: in the context of global names S, global parameters Δ, and a context of locally declared variables Γ, the pattern π has the type P, name dependency C and produces a residual context Γ_1 of pattern variables and their typings. This residual context is to be passed to subsequent computations. The rules of this judgment are presented in Figure 16. Note that, because we are limited to only simply-typed fragment, the local variables that the typing rules deposit in Γ will always be ordinary value variables, and always simply typed. On the other hand, we do allow a bit more generality in the case of pattern variables $[E\ x_1 \cdots x_n]$; they still can match only terms of simple types, but these terms can have subterms of more general typing.

In order to incorporate pattern matching into the core language, we enlarge the syntax with a new term constructor.

$$\mathit{Terms}\quad e \quad ::= \quad \ldots \mid \mathbf{case}\ e_0\ \mathbf{of}\ \mathbf{box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2$$

The intended operational interpretation of **case** is to evaluate the argument e_0 to obtain a boxed expression **box** w, then match w to the pattern π. If the matching is successful, it creates an environment with bindings for the pattern variables, and then evaluates e_1 in this environment. If the matching fails, the branch e_2 is taken. The typing rule for **case** is:

$$\frac{S; \Gamma \vdash_\Delta e_0 : \Box(P[D])\,[C] \qquad S; \cdot \Vdash_\Delta \pi : P\,[D] \Longrightarrow \Gamma_1 \qquad S; \Gamma, \Gamma_1 \vdash_\Delta e_1 : B\,[C] \qquad S; \Gamma \vdash_\Delta e_2 : B\,[C]}{S; \Gamma \vdash_\Delta \mathbf{case}\ e_0\ \mathbf{of}\ \mathbf{box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2 : B\,[C]}$$

Observe that the second premise of **case** requires an empty variable context, so that patterns cannot contain outside value or expression variables.

The meta operations of name substitution and name transposition, as well as the $\mathbf{fn}_0$ function on terms, readily extend.

$$\begin{aligned}\{e/a\}(\textbf{case } e_0 \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) &= (\textbf{case } \{e/a\}e_0 \textbf{ of box } \pi \Rightarrow \{e/a\}e_1 \textbf{ else } \{e/a\}e_2) \\ \mathbf{fn}_0(\textbf{case } e_0 \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) &= \mathbf{fn}_0(e_0) \cup \mathbf{fn}_0(e_1) \cup \mathbf{fn}_0(e_2)\end{aligned}$$

Example 2
To illustrate intensional code analysis, the following examples presents a generalization of our old exponentiation function. Instead of powering only integers, we can power functions too, i.e. have a functional computing $f \mapsto \lambda x.\ (fx)^n$. The functional is passed the code for f, and an integer n, and returns the code for $\lambda x.\ (fx)^n$. The idea is to have this residual code be as optimized as possible, while still computing the extensionally same result.

For comparison, we first present a $\lambda^{\Box}$ version of the function-powering functional.

```
fpowbox : □(int->int) -> int -> □(int->int) =
    λf:□(int->int). λn:int.
        let box F = f
            box P = powbox2 n
        in
            box (λv:int. (P (F v)))
        end

- fpowbox (box λw:int. w + 1) 2;
val it = box (λv:int. (λx.x*(λy.y*(λz.1)y)x) ((λw.w+1)v)) : □(int->int)
```

Observe that the residual program contains a lot of unnecessary redices. As could be expected, $\lambda^{\bigcirc}$ (and for that matter, MetaML as well), provides a better way to stage the code.

```
fpowcirc1 : ◯(int->int) -> int -> ◯(int->int) =
    λf:◯(int->int). λn:int.
        let val P = powcirc2 n
        in
            next (λv:int. (prev P) ((prev f) v))
        end

- fpowcirc1 (next λw:int. w + 1) 2;
val it = next (λv:int. (λx.x*(x*1)) ((λw.w+1) v)) : ◯(int->int)
```

In fact, there is at least one other way to program this functional in $\lambda^{\bigcirc}$: we can eliminate the outer beta-redex from the residual code, at the price of duplicating the inner one.

```
fpowcirc2 : ◯(int->int) -> int -> ◯(int->int) =
    λf:◯(int->int). λn:int.
        next (λv:int.
              prev (let val b = next ((prev f) v)
                    in
                        powcirc' b n
                    end))

- fpowcirc2 (next (λw:int. w + 1)) 2;
val it = next (λv:int. ((λw.w+1) v) * ((λw.w+1) v) * 1) : ◯(int->int)
```

All three of the above programs can be encoded in the new language as well. The first program `fpowbox` is simply copied line-for-line. The $\lambda^{\bigcirc}$ version `fpowcirc1` and `fpowcirc2` will require translations `fpow1` and `fpow2`, which we show below.

```
fpow1 : □(int->int) -> int -> □(int->int) =
  λf:□(int->int). λn:int.
      let box p = pow n
          box g = f
      in
          box (λv:int. p (g v))
      end

-fpow1 (box λw:int. w + 1) 2;
val it = box (λv:int. (λx.x*(x*1)) ((λw.w+1) v)) : □(int->int)


fpow2 : □(int->int) -> int -> □(int->int) =
  λf:□(int->int). λn:int.
    new a:int in
      let box f' = f
          box e = pow' ⟦a⟧ (box (f' a)) n
      in
          box (λv:int. {a = v} e)
      end

- fpow2 (box (λw:int. w + 1)) 2;
val it = box (λv:int. ((λw.w+1) v) * ((λw.w+1) v) * 1) : □(int->int)
```

However, neither of the above implementations is quite satisfactory, since, evidently, the residual code in all the cases contains unnecessary redices. The reason is that we do not utilize the *intensional* information that the passed argument is actually a boxed *lambda* abstraction, rather than a more general expression of a functional type. In a language with intensional code analysis, we can do a bit better. We can test the argument at run-time and output a more optimized result if the argument is a lambda expression. This way we can obtain the most simplified, if not the most efficient residual code.

```
fpow : □(int->int) -> int -> □(int->int) =
    λf:□(int->int). λn:int.
      case f of
          box (λw:int. [E w]) =>
              new a:int in
                 let box F = pow' ⟦a⟧ (E @ a) n
                 in
                     box (λw:int. {a = w} F)
                 end
          else fpow1 f n

- fpow (box λw:int. w + 1) 2;
val it = box(λw:int.(w + 1) * (w + 1) * 1): □(int->int)
```

∎

Lemma 16
If $S; \Gamma \Vdash_{\Delta} \pi : P\,[C] \Longrightarrow \Gamma_1$, *then* $\Gamma_1 = \Gamma_1^{\Delta}$.

Proof: Trivial. ∎

$$\frac{\mathbf{fv}(w) \subseteq \{x_1, \ldots, x_n\}}{\mu, w \rhd [E\ x_1 \cdots x_n] \Longrightarrow (\mu, a_1, \ldots, a_n), [E \mapsto (a_1 \ldots a_n \,.\, \mathbf{box}\ [a_1, \ldots, a_n/x_1, \ldots, x_n]w)]}$$

$$\frac{\mu, w \rhd \pi \Longrightarrow \mu', \Theta \qquad \mathbf{fn}_0(w) \subseteq D}{\mu, w \rhd \pi{:}P[D] \Longrightarrow \mu', \Theta}$$

$$\frac{}{\mu, x \rhd x \Longrightarrow \mu, \cdot} \qquad \frac{}{\mu, a \rhd a \Longrightarrow \mu, \cdot} \qquad \frac{}{\mu, * \rhd * \Longrightarrow \mu, \cdot}$$

$$\frac{\mu, w \rhd \pi \Longrightarrow \mu', \Theta}{\mu, \lambda x{:}P.\ w \rhd \lambda x{:}P.\ \pi \Longrightarrow \mu', \Theta} \qquad \frac{\mu, w \rhd \pi \Longrightarrow \mu', \Theta}{\mu, \mathbf{fix}\ x{:}P.\ w \rhd \mathbf{fix}\ x{:}P.\ \pi \Longrightarrow \mu', \Theta}$$

$$\frac{\mu, w_1 \rhd \pi_1 \Longrightarrow \mu_1, \Theta_1 \qquad \mu_1, w_2 \rhd \pi_2 \Longrightarrow \mu_2, \Theta_2}{\mu, (w_1\ w_2) \rhd (\pi_1)\ (\pi_2{:}P_2) \Longrightarrow \mu_2, (\Theta_1 \circ \Theta_2)}$$

Figure 17: Operational semantics for pattern matching.

Lemma 17 (Parametricity of Pattern Matching)

1. *if* $S; \Gamma \Vdash_{\Delta, p\#K} \pi : P\,[C] \Longrightarrow \Gamma_1$, *and* D *is well-formed dependency set, i.e.* $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$, *and is fresh for* K, *i.e.* $\Delta \vdash D \ \#\ K$, *then*

$$S; [D/p]\Gamma \Vdash_\Delta \pi : P\,[[D/p]C] \Longrightarrow [D/p]\Gamma_1$$

2. *if* $S; \Gamma \Vdash_\Delta \pi : P\,[C] \Longrightarrow \Gamma_1$ *then*

$$(a\ b)S; (a\ b)\Gamma \Vdash_{(a\ b)\Delta} (a\ b)\pi : P\,[(a\ b)C] \Longrightarrow (a\ b)\Gamma_1$$

Proof: By a straightforward induction on the structure of π. ■

Using the previous two lemmas, we can augment the theory of the core language with pattern matching and the new construct **case**. In particular, the Substitution Principles (Lemma 10), and the Parametricity Properties (Lemma 11), are easily extended with the additional inductive cases resulting from this addition. We present the completed proofs of these lemmas in the Appendix.

The operational semantics for patterns is given through the new judgment

$$\mu, w \rhd \pi \Longrightarrow \mu', \Theta$$

which reads: in a global store of names μ, the matching of *contracted* expression w to the pattern π extends the global store to μ' and generates a substitution Θ for the pattern-variables of π. The rules for this judgment are given in Figure 17. Notice that, by the nature of pattern matching, the substitution Θ is of a very simple structure. In particular, the terms from its range never contain variables from its domain.

As already explained, the pattern variable $[E\ x_1 \cdots x_n]$ should match an expression w provided that w depends only on variables $x_1, \ldots, x_n$. Thus, the rule for pattern variables explicitly provides the required check. Similarly, the pattern $\pi{:}P[C]$ has to ensure that all the free names of a matched expression w are in the dependency set C, and the corresponding rule reflects that. That this checks are sound with respect to the type system is the motivation for the following definition and lemma.

Definition 18 (Types for Substitutions)
The judgment $S \vdash_\Delta \Theta : \Gamma$ *denotes that* Θ *is a substitution for the variables in* Γ, *and that the substituting terms allow occurrences of only the names in* S. *In other words* $S \vdash_\Delta \Theta : \Gamma$ *if for every pattern-variable* $E{:}A \in \Gamma$ *we have* $S; \cdot \vdash_\Delta \Theta(E) : A\,[\,]$.

Lemma 19 (Type Preservation for pattern-matching)
If $S; \Gamma_1^\Delta \Vdash \pi : P\,[C] \Longrightarrow \Gamma_2$ *and* $S; \Gamma_1^\Delta \vdash_\Delta w : P\,[C]$ *and* $\mu_S, w \rhd \pi \Longrightarrow \mu', \Theta$, *then there exists* $S' \supseteq S$ *such that* $\mu' = \mu_{S'}$ *and* $S' \vdash_\Delta \Theta : \Gamma_2$.

Proof: By induction on the structure of the pattern π. We present the interesting cases.

case $\pi = [E\ x_1 \cdots x_n]$.

1. By assumption,
$$S; \Gamma_1^\Delta \Vdash_\Delta [E\ x_1 \cdots x_n] : P\,[C] \Longrightarrow E{:}\ \mathop{\text{И}}_{a_i:P_i} \Box(P[a_i, C])$$
In other words, $\Gamma_2 = E{:}\ \mathop{\text{И}}_{a_i:P_i} \Box(P[a_i, C])$.
2. Also by assumption, $S; \Gamma_1^\Delta \vdash_\Delta w : P\,[C]$.
3. By (2) and name-for-variable substitution (Lemma 4.4),
$$S, a_i{:}P_i; \Gamma_1^\Delta \setminus \{x_i{:}P_i\} \vdash_\Delta [a_i/x_i]w : P\,[a_i, C]$$
4. By operational semantics of pattern-matching,
$$\mu, \pi \rhd w \Longrightarrow (\mu, a_1, \ldots, a_n), [E \mapsto (a_1 \ldots a_n\ .\ \mathbf{box}\ [a_1, \ldots, a_n/x_1, \ldots, x_n]w)]$$
In other words, the residual substitution Θ is in this case defined as
$$\Theta = [E \mapsto (a_1 \ldots a_n\ .\ \mathbf{box}\ [a_1, \ldots, a_n/x_1, \ldots, x_n]w)]$$
for some fresh names $a_i{:}P_i$.
5. By the same rule for evaluation of patterns, $\mathbf{fv}(w) \subseteq \{x_1 \ldots, x_n\}$.
6. By (5) and context strengthening (Lemma 9.1), $S, a_i{:}P_i; \cdot \vdash_\Delta [a_i/x_i]w : P\,[a_i, C]$,
7. Then (6) implies, by typing rules
$$S, a_i{:}P_i; \cdot \vdash_\Delta (a_1 \ldots a_n\ .\ \mathbf{box}\ [a_1, \ldots, a_n/x_1, \ldots, x_n]w) : \mathop{\text{И}}_{a_i:P_i} \Box(P[a_i, C])$$
8. Hence, taking $S' = S, a_i{:}P_i$ satisfies $S' \vdash_\Delta \Theta : \Gamma_2$, which was required.

case $\pi = (\pi'{:}P[D])$ where $D \subseteq C$.

1. By assumption, $S; \Gamma_1^\Delta \vdash_\Delta w : P\,[C]$.
2. By operational semantics for patterns, $\mathbf{fn}_0(w) \subseteq D$.
3. From (1) and (2), by strengthening (Lemmas 9.4 and 9.5), we know that
$$S; \Gamma_1^\Delta \vdash_\Delta w : P\,[D]$$
4. The result follows from (3) by induction hypothesis.

■

The last piece to be added is the operational semantics for the **case** statement. First we extend the judgment for contraction.

$$\frac{e_0 \xrightarrow{S} w_0 \qquad e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2}{(\mathbf{case}\ e_0\ \mathbf{of\ box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2) \xrightarrow{S} (\mathbf{case}\ w_0\ \mathbf{of\ box}\ \pi \Rightarrow w_1\ \mathbf{else}\ w_2)}$$

The additional cases which arise in the lemmas 12, 13 and 14 are easy to prove. We also extend the small-step semantics (see the rules below). Notice that the premise of last rule makes use of the fact that the operational semantics for patterns is decidable, i.e. it is always possible to to find out, for given μ, w and π which unique μ' and Θ, if any, satisfy the relation $\mu, w \triangleright \pi \Longrightarrow \mu', \Theta$.

$$\frac{\mu, e_0 \longmapsto \mu', e_0'}{\mu, (\textbf{case } e_0 \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) \longmapsto \mu', (\textbf{case } e_0' \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2)}$$

$$\frac{\mu, w \triangleright \pi \Longrightarrow \mu', \Theta}{\mu, (\textbf{case box } w \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) \longmapsto \mu', \Theta(e_1)}$$

$$\frac{\nexists \mu', \Theta. \quad \mu, w \triangleright \pi \Longrightarrow \mu', \Theta}{\mu, (\textbf{case box } w \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) \longmapsto \mu, e_2}$$

Finally, using the lemmas established in this section, it is possible to augment the proof of the Progress and Type Preservation theorem (Theorem 15) to handle the extended language. The complete proof is presented in the Appendix.

Example 3
In this example we encode a function that beta-reduces the expressions passed to it as an argument. At present, we do not have parametric type polymorphism in our language, so we restrict this function to only expressions of type `real`, but we do allow arbitrary number of names in them.

```
fix breduce:∀p. □(real[p])->□(real[p]).
    Λp. λe:□(real[p]).
        case e of
           box((λx. [E1 x]) [E2]:real) => (* E1: И_{a:real} □(real[a]), E2:□ real *)
              new a:real in
                  let box e1 = breduce ⟦p, a⟧ (E1 @ a)
                      box e2 = breduce ⟦p, a⟧ (E2)
                  in
                      box ({a = e2} e1)
                  end
           else e
```

■

Example 4
This example is a (segment) of the meta function for symbolic differentiation. The function takes a name abstraction as an argument: the body of the abstraction is a boxed term encoding the expression to be differentiated; the abstracted name represents the variable with respect to which the differentiation takes place. When the boxed expression is a sum of two subexpressions, the function just recurses over them. When the boxed expression is a beta-redex (of a limited form), it first reduces it before recursing. Other names and constants are matched in the default case, which thus returns the derivative 0.

Notice that the present lack of polymorphic patterns prevents us from recognizing, let alone reducing all the beta redices that could possibly occur in the argument; although, admittedly, it is wrong, we currently let them pass through the default case.

```
diff : ∀p. (Иa:real.□real[a, p]) -> (Иa:real.□real[a, p]) =
  fix diff.
    Λp. λe:(Иa:real.□real[a, p]).
       new a:real  (* the differentiating name *)
       in
         case (e @ a) of
          box a => a.(box 1)
          | box ([E1] + [E2]) =>
             let box e1 = (diff ⟦p⟧ (a.E1)) @ a
                 box e2 = (diff ⟦p⟧ (a.E2)) @ a
             in
                 a.box (e1 + e2)
             end
          | box ((λx:real. [E1 x]) [E2]:real) =>
             new b:real in
              let box e1 = E1 @ b
                  box e2 = E2
              in
                  diff ⟦p⟧ (a.box ({b = e2} e1))
              end
         else a.(box 0)
```

■

7 Future Work

There are numerous directions in which the system we presented here can be extended. We list some of the more important ones:

1. **Arbitrary, not just simple types for names.** With the limitation that names can only be simply-typed, our language can encode only object programs with simply-typed free variables. This makes it a two-level, rather than a multi-level language like $\lambda^{\bigcirc}$ and MetaML. It would be interesting to investigate how generalizing the typing for names, if possible at all, will influence the rest of the language, in particular the operations of name abstraction and concretion. One can imagine retaining the simple typing for the bound name in the И quantifier, for the *predicative* variant, or allowing arbitrary names for *impredicative* И-quantification.

2. **Modal type of names.** In the present version of the system, names used in abstraction and concretion must always be constants, in order to account for them in the dependency annotations. In other words, we cannot compute with names; they can be passed around as part of boxed/code expressions, but once unboxed, they cannot be used for abstraction and concretion. For this purpose, it may be beneficial to add a separate type modality, say $\triangle(A[C])$, to classify names of type A listed in the dependency C. The new modal type should be a subtype of $\Box(A[C])$.

3. **Type polymorphism and type-polymorphic recursion.** In a meta-programming language, the typing of object programs is made part of the typing of the meta programs. Consequently, such a language has a lot of types to care of and thus needs strong notions of type polymorphism. This was already evident from our example programs for beta reduction and symbolic differentiation in Section 6.

4. **Existential name and type abstraction.** The motivation for this comes from automatic code generation. Say that we have a datatype `absyn`, representing the abstract syntax of the language, and that we have synthesized an abstract syntax tree for a certain object program. We would like to invoke that program in run-time, i.e. transform it into its boxed code representation and then evaluate

it. A program that performs this transformation is often referred to as a "visible compiler". In a meta-programming language we ought be able to implement a function `vcomp` representing a visible compiler, but the question is what its type should be. Obviously, the types of boxed expressions that `vcomp` is supposed to produce will depend on the abstract syntax tree supplied as an argument, and thus the range type of the function will have to be existentially quantified; something like `vcomp : absyn -> ∃p.∃A.□(A[p])`. Intensional code analysis should incorporate a certain form of intensional type analysis as well, and that would enable the object programs resulting from the visible compiler to be used in non-trivial ways in the rest of the program.

5. **Generalized forms of recursion.** In all the meta-programming languages considered in this work, fixpoint variables have been treated as ordinary *value* variables. In such a setup, a a function can make a recursive call to itself only on the current code level; the call can never be "postponed", i.e. be on a higher code level (under a **box** or **next**). It would be interesting to investigate if it is possible to have fixpoint *expression* variables.

6. **Adding references.** While it should be relatively straightforward to add references to our language, it may be possible to give them a much stronger role in the modal setup. For example, a reference can be used as a name with a definition – carrying both the extensional information of its referent, as well as the intensional information of its name. Much like names, it should be sound with respect to intensional analysis to endow the references with cross-stage persistence, i.e. allow them to cross the code-level boundaries.

   Related to problem (5) above, we would also like to investigate recursive references. These would be references whose extension can refer to the reference name itself, but perhaps in postponed positions (i.e. under a box).

7. **Enriching the language of patterns.** Is it possible to extend the pattern-matching mechanism to analyze and destruct expressions under more than one layers of boxes? How to match against binding constructs like **let val** and **let box**?

8. **Model theory of the language.** Last, but probably most important, we should build models for our type system and put it on a sound logical footing. Interaction between names and modal logic has been of interest to philosophical investigations for quite some time (see [Kri80] and [FM99]). We hope to draw on this work for the future developments.

8 Conclusions

In this paper we presented a typed functional language for meta-programming, employing a novel way to define a modal type of code. We formulated its static and dynamic semantics and proved the corresponding Progress and Type Preservation theorem. The system is based on adding the notion of names, as developed by Pitts and Gabbay in [PG00], [GP01], [Pit01] and [Gab00], to the $\lambda^{\square}$-calculus of proof terms for the necessitation fragment of modal logic S4 [PD01]. The motivation for combining the two systems comes from the long-recognized need of meta-programming to handle code expressions containing free variables ([Dav96], [Tah99b], [MTBS99]). In our language, the free variables of a code expression are represented by *names*. Names can be operationally thought of as references, except that the type system puts certain limitations on their occurrence. First, a term can be evaluated only if it contains no names on the lowest code level – this is in sharp distinction with the semantics of names in FreshML [PG00]. Second, names cannot escape the scope of their creating **new** unguarded by an appropriate construct for *name abstraction*.

The set of names on which a term depends is reflected in its typing. We hope that this design would allow a more fine-grained control over the occurrence of free variables, and thus make the language more expressive, easier to program in, and more amenable to future extensions.

On a related note, it may be of interest here to draw a parallel with with a related phenomenon which occurs in the extension of MetaML with references [CMS01]. A reference in MetaML must not be assigned a postponed/code value which contains variables bound on the *outside* of the code constructor. Indeed, if such a thing occured, than the "free" variable may escape the scope of the *outside* λ-construct which introduced it. For technical reasons, however, this actually cannot be prohibited, so the authors resort to a hygienic handling of scope extrusion by annotating a term with the list of free variables it contains. Obviously, this very much reminds of the operation of name abstraction. In particular, one can imagine that a similar scenario would happen if we tried to extend our language with references, too. We would need to require that the type of an expression assigned to a reference does not contain any unbound names, in order to avoid scope extrusion. In such a setup, the operation of name abstraction would correspond exactly to the mentioned MetaML annotations.

That aside, we also considered constructs for polymorphism in names and for intensional code analysis. The former provides for writing programs which are parametric in their name dependencies. The later is a pattern-matching mechanism to compare, inspect and destruct code values at run-time, which we currently restricted to the simply typed fragment of the language. Taken together, they allow recursing over a source of (a simply-typed) object program – a feature that we hope will find its use in programming code optimizations in a setup of scientific and symbolic computation.

References

[CMS01] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2001. to appear.

[CMT00] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2000.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan 1996*, pages 258–270. ACM Press, New York, 1996.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[FM99] Melvin Fitting and Richard L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Publishers, 1999.

[Gab00] Murdoch J. Gabbay. *A Theory of Inductive Definitions with α-Equivalence*. PhD thesis, Cambridge University, August 2000.

[GP01] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 2001. Special issue in honour of Rod Burstall. To appear.

[Gri89] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[Kri80] Saul A. Kripke. *Naming and Necessity*. Harvard University Press, 1980.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures. In *Proceedings of the Logical Frameworks BRA Workshop*, May 1990.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, pages 193–207, 1999.

[Nie01] Michael Florentin Nielsen. Combining close and open code. Unpublished, 2001.

[Ode94] Martin Odersky. A functional theory of local names. In *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 48–59, New York, NY, USA, 1994. ACM Press.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[Pit01] Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In Naoki Kobayashi and Benjamin C. Pierce, editors, *TACS*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer, 2001.

[Roz93] Guillermo J. Rozas. Translucent procedures, abstraction without opacity. Technical Report AITR-1427, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1993.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conf. Record 29th ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'02, Portland, OR, USA*, pages 154–165, New York, 2002. ACM Press.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *SAIG*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.

[Tah99a] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival. *ACM SIGPLAN Notices*, 34(11):34–43, 1999.

[Tah99b] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–235, 1998.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), 1998.

A Proofs

Lemma 10 (Substitution Principles)

1. *if* $S; \Gamma \vdash_\Delta e_1 : A\,[C]$ *and* $S; \Gamma, x{:}A \vdash_\Delta e_2 : B\,[C]$*, then* $S; \Gamma \vdash_\Delta [e_1/x]e_2 : B\,[C]$.
2. *if* $S; \Gamma_1^{\ominus} \vdash_\Delta e_1 : A\,[D]$ *and* $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_2 : B[C]$*, then* $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e_2 : B[C]$.

3. *if* $S; \Gamma_1^{\triangledown} \vdash_\Delta e_1 : A\,[D]$ *and* $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_2 : B[C]$*, then* $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B[C]$.

4. *if* $S, a{:}P; \Gamma_1 \vdash_\Delta e_1 : P[C]$ *and* $S, a{:}P; \Gamma_2^{\triangle} \vdash_\Delta e_2 : B[a, C]$*, then* $S, a{:}P; \Gamma_1, \Gamma_2^{\triangle} \vdash_\Delta \{e_1/a\}e_2 : B[C]$.

Proof: All the proofs are by induction on the structure of the typing derivation for e_2.

Principle 1. if $S; \Gamma \vdash_\Delta e_1 : A\,[C]$ and $S; \Gamma, x{:}A \vdash_\Delta e_2 : B\,[C]$, then $S; \Gamma \vdash_\Delta [e_1/x]e_2 : B\,[C]$.

case $e_2 = *$, $e_2 = y$, $e_2 = u$ or $e_2 = a$.

These are trivial, since the substitution is $[e_1/x]e_2$ is vacuous.

case $e_2 = x$.

Reduces to one of the assumptions.

case $e_2 = \lambda y{:}B_1.\, e$, and $B = B_1 \to B_2$.

1. By typing derivation, $S; \Gamma, x{:}A, y{:}B_1 \vdash_\Delta e : B_2\,[C]$.
2. By induction hypothesis, $S; \Gamma, y{:}B_1 \vdash_\Delta [e_1/x]e : B_2\,[C]$.
3. This leads to the required $S; \Gamma \vdash_\Delta \lambda y{:}B_1.\, [e_1/x]e : B_1 \to B_2\,[C]$.

case $e_2 = e'\ e''$.

From the typing derivations for e' and e'', by using the induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e' : B' \to B\,[C]$ and $S; \Gamma \vdash_\Delta [e_1/x]e'' : B'\,[C]$. Thus follows the result.

case $e_2 = \textbf{fix}\ y{:}B.\, e$.

1. By typing derivation, $S; \Gamma, x{:}A, y{:}B \vdash_\Delta e : B\,[C]$.
2. By induction hypothesis, $S; \Gamma, y{:}B \vdash_\Delta [e_1/x]e : B\,[C]$.
3. This leads to the required $S; \Gamma \vdash_\Delta \textbf{fix}\ y{:}B.\, [e_1/x]e : B\,[C]$.

case $e_2 = (\textbf{box}\ e)$.

Trivial because x does not occur in e, and so the substitution $[e_1/x]\textbf{box}\ e$ is vacuous.

case $e_2 = (\textbf{let box}\ u = e'\ \textbf{in}\ e'')$.

1. From typing derivation, by induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e' : \Box(B'[C'])\,[C]$.
2. Also notice that, $S; \Gamma, u{::}B'\,[\emptyset] \vdash_\Delta [e_1/x]e'' : B\,[C]$ or $S; \Gamma, u{\div}B'\,[C'] \vdash_\Delta [e_1/x]e'' : B\,[C]$, depending on whether $C' = \emptyset$ or not.
3. In either case, we have the required $S; \Gamma \vdash_\Delta (\textbf{let box}\ u = [e_1/x]e'\ \textbf{in}\ [e_1/x]e'') : B\,[C]$.

case $e_2 = a\,.\,e$, where $B = \underset{a:P}{\text{И}}\ B'$ and $a{:}P \in S$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_\Delta e : B'\,[C]$ and $\Delta \vdash \mathbf{fp}(B')\ \#\ a$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e : B'\,[C]$,
3. so we just reassemble $S; \Gamma \vdash_\Delta a\,.\,[e_1/x]e : (\underset{a:P}{\text{И}}\ B')\,[C]$.

case $e_2 = e \mathbin{@} a$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_\Delta e : (\underset{a:P}{\text{И}}\ B)\,[C]$ and $\Delta \vdash \mathbf{fp}(B)\ \#\ a$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e : (\underset{a:P}{\text{И}}\ B)\,[C]$,
3. so just reassemble $S; \Gamma \vdash_\Delta ([e_1/x]e) \mathbin{@} a : B\,[C]$.

case $e_2 = \textbf{new}\ a{:}P\ \textbf{in}\ e$.

1. By typing derivation, $S, a{:}P; \Gamma, x{:}A \vdash_{\Delta\#a} e : B\,[C]$ and $\Delta\#a \vdash B[C] \ \# \ a$.
2. By induction hypothesis, $S, a{:}P; \Gamma \vdash_{\Delta\#a} [e_1/x]e : B\,[C]$,
3. so assemble back into $S; \Gamma \vdash_\Delta \textbf{new}\ a{:}P\ \textbf{in}\ [e_1/x]e : B\,[C]$,

case $e_2 = \Lambda p\#K.\ e$, where $B = \forall p\#K.\ B'$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_{\Delta, p\#K} e : B'\,[C]$.
2. By induction hypothesis, $S; \Gamma \vdash_{\Delta, p\#K} [e_1/x]e : B'\,[C]$,
3. so we derive the required $S; \Gamma \vdash_\Delta \Lambda p\#K.\ [e_1/x]e : (\forall p\#K.\ B')\,[C]$.

case $e_2 = e\ [\![D]\!]$ where $B = [D/p]B'$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_\Delta e : (\forall p\#K.\ B')\,[C]$, $\Delta \vdash D \ \# \ K$, and $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e : (\forall p\#K.\ B')\,[C]$,
3. and follows the required $S; \Gamma \vdash_\Delta [e_1/x]e\ [\![D]\!] : ([D/p]B')\,[C]$.

case $e_2 = \{a \doteq e'\}\ e''$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_\Delta e' : P\,[C]$ and $S; \Gamma, x{:}A \vdash_\Delta e'' : B\,[a, C]$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e' : P\,[C]$ and $S; \Gamma \vdash_\Delta [e_1/x]e'' : B\,[a, C]$,
3. so follows the needed result $S; \Gamma \vdash_\Delta \{a \doteq ([e_1/x]e')\}\ ([e_1/x]e'')$.

case subtyping from $B' \leqslant: B$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_\Delta e_2 : B'\,[C]$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e_2 : B'\,[C]$.
3. Use subtyping to conclude, $S; \Gamma \vdash_\Delta [e_1/x]e_2 : B\,[C]$.

case $e_2 = (\textbf{case}\ e_0\ \textbf{of box}\ \pi \Rightarrow e'\ \textbf{else}\ e'')$.

1. By typing derivation, $S; \Gamma \vdash_\Delta e_0 : \Box(P[D])\,[C]$, and $S; \cdot \Vdash_\Delta \pi : P\,[D] \Longrightarrow \Gamma_1$ and $S; \Gamma, \Gamma_1 \vdash_\Delta e' : B\,[C]$, and $S; \Gamma \vdash_\Delta e'' : B\,[C]$.
2. By induction hypothesis, $S; \Gamma \vdash_\Delta [e_1/x]e_0 : \Box(P[D])\,[C]$, and $S; \Gamma, \Gamma_1 \vdash_\Delta [e_1/x]e' : B\,[C]$, and $S; \Gamma \vdash_\Delta [e_1/x]e'' : B\,[C]$.
3. Now just assemble back into the required result, using the typing rule for **case**.

Principle 2. if $S; \Gamma_1^\ominus \vdash_\Delta e_1 : A\,[D]$ and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_2 : B[C]$, then $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e_2 : B[C]$.

case $e_2 = *$, $e_2 = x$, $e_2 = u'$, or $e_2 = a$.

Trivial, since the substitution $[e_1/u]e_2$ is vacuous.

case $e_2 = u$, where $B = A$ and $D \subseteq C$.

1. By hypothesis weakening, $S; \Gamma_1^\ominus \vdash_\Delta e_1 : A\,[D]$ implies $S; \Gamma_1 \vdash_\Delta e_1 : A\,[D]$
2. and then $S; \Gamma_1, \Gamma_2 \vdash_\Delta e_1 : A\,[C]$, using both hypothesis and dependency weakening.

case $e_2 = \lambda y{:}B_1.\ e$, where $B = B_1 \to B_2$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D], y{:}B_1 \vdash_\Delta e : B_2\,[C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2, y{:}B_1 \vdash_\Delta [e_1/u]e : B_2\,[C]$.
3. Now assemble back $S; \Gamma_1, \Gamma_2 \vdash_\Delta \lambda y{:}B_1.\ [e_1/u]e : B_1 \to B_2\,[C]$.

case $e_2 = e' \ e''$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e' B' \to B\,[C]$ and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e'' B'\,[C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' : B' \to B\,[C]$ and $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e'' : B'\,[C]$.
3. We now just assemble back $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' \ [e_1/u]e''\,[B]\,[C]$.

case $e_2 = \mathbf{fix}\ y{:}B.\ e$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D], y{:}B \vdash_\Delta e : B\,[C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2, y{:}B \vdash_\Delta [e_1/u]e : B\,[C]$.
3. Now assemble back $S; \Gamma_1, \Gamma_2 \vdash_\Delta \lambda y{:}B.\ [e_1/u]e : B\,[C]$.

case $e_2 = \mathbf{box}\ e$, where $B = \Box(B'[C'])$.

1. By typing derivation, $S; \Gamma_2^{\triangledown}, u{::}A[D] \vdash_\Delta e : B'\,[C']$.
2. Because $\Gamma_1^{\ominus} = \Gamma_1^{\ominus\ominus}$, we have $S; \Gamma_1^{\ominus\ominus} \vdash_\Delta e_1 : A\,[D]$.
3. From (1), (2) and the induction hypothesis, $S; \Gamma_1^{\ominus}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/x]e : B'\,[C']$.
4. By Lemma 5, $S; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/x]e : B'\,[C']$,
5. and finally, we can reassemble $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{box}\ ([e_1/x]e) : \Box(B'[C'])\,[C]$,

case $e_2 = \mathbf{let\ box}\ v = e'\ \mathbf{in}\ e''$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e' : \Box(B'[C'])\,[C]$
2. Also by typing derivation, either $S; \Gamma_2, u{::}A[D], v{::}B' \vdash_\Delta e'' : B\,[C]$ or $S; \Gamma_2, u{::}A[D], v{\div}B'[C'] \vdash_\Delta e'' : B\,[C]$, depending whether $C = \emptyset$ or $C \neq \emptyset$.
3. By (1) and induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' : \Box(B'\,[C'])\,[C]$
4. By (2) and induction hypothesis, $S; \Gamma_1, \Gamma_2, v{::}B' \vdash_\Delta [e_1/u]e'' : B\,[C]$ or $S; \Gamma_1, \Gamma_2, v{\div}B'[C'] \vdash_\Delta [e_1/u]e'' : B\,[C]$.
5. Either way, just assemble back $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{let\ box}\ v = [e_1/u]e'\ \mathbf{in}\ [e_1/u]e'' : B\,[C]$.

case $e_2 = a\,.\,e'$, where $B = \underset{a:P}{\text{И}}\ B'$ and $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e' : B'\,[C]$ and $\Delta \vdash \mathbf{fp}(B')\ \#\ a$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' : B\,[C]$.
3. Thus, reassemble, $S; \Gamma_1, \Gamma_2 \vdash_\Delta a\,.\,([e_1/u]e') : B\,[C]$.

case $e_2 = e' \mathbin{@} a$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e : (\underset{a:P}{\text{И}}\ B)\,[C]$, and $\Delta \vdash \mathbf{fp}(B)\ \#\ a$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e : (\underset{a:P}{\text{И}}\ B)\,[C]$,
3. so follows the required $S; \Gamma_1, \Gamma_2 \vdash_\Delta ([e_1/u]e) \mathbin{@} a : B\,[C]$.

case $e_2 = \mathbf{new}\ a{:}P\ \mathbf{in}\ e'$.

1. By typing derivation, $S, a{:}P; \Gamma_2, u{::}A[D] \vdash_{\Delta\#a} e' : B\,[C]$, and $\Delta\#a \vdash B[C]\ \#\ a$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2 \vdash_{\Delta\#a} [e_1/u]e' : B\,[C]$,
3. and then the required $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{new}\ a{:}P\ \mathbf{in}\ [e_1/u]e' : B\,[C]$.

case $e_2 = \Lambda p\#K.\ e'$, where $B = \forall p\#K.\ B'$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_{\Delta, p\#K} e' : B' [C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_{\Delta, p\#K} [e_1/u]e' : B' [C]$,
3. and then the required, $S; \Gamma_1, \Gamma_2 \vdash_\Delta \Lambda p\#K. [e_1/u]e' : (\forall p\#K. B') [C]$.

case $e_2 = e' \llbracket D' \rrbracket$, where $B = [D'/p]B'$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e' : (\forall p\#K. B') [C]$, $\Delta \vdash D' \ \# \ K$ and $D' \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' : (\forall p\#K. B') [C]$,
3. so assemble back into $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' \llbracket D' \rrbracket : ([D'/p]B') [C]$.

case $e_2 = \{a \doteq e'\} \ e''$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e' : P [C]$ and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e'' : B [a, C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e' : P [C]$, and $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e'' : B [a, C]$.
3. Assemble back into the required $S; \Gamma_1, \Gamma_2 \vdash_\Delta \{a \doteq [e_1/u]e'\} \ [e_1/u]e'' : B [C]$.

case subtyping from $B' \leqslant: B$.

1. By typing derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_2 : B' [C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/u]e_2 : B' [C]$.
3. Use subtyping to conclude, $S; \Gamma_1, \Gamma \vdash_\Delta [e_1/u]e_2 : B [C]$.

case $e_2 = (\textbf{case } e_0 \textbf{ of box } \pi \Rightarrow e' \textbf{ else } e'')$.

1. By derivation, $S; \Gamma_2, u{::}A[D] \vdash_\Delta e_0 : \Box(P[D']) [C]$, and $S; \cdot \Vdash_\Delta \pi : P [D'] \Longrightarrow \Gamma'$ and $S; \Gamma_2, u{::}A[D], \Gamma' \vdash_\Delta e' : B [C]$, and $S; \Gamma_2, u{::}A[D] \vdash_\Delta e'' : B [C]$.
2. By induction hypothesis, $S; \Gamma_1^\ominus, \Gamma_2 \vdash_\Delta [e_1/u]e_0 : \Box(P[D']) [C]$, and $S; \Gamma_1^\ominus, \Gamma_2, \Gamma' \vdash_\Delta [e_1/u]e' : B [C]$, and $S; \Gamma_1^\ominus, \Gamma_2 \vdash_\Delta [e_1/u]e'' : B [C]$.
3. By typing rule for **case**, $S; \Gamma_1^\ominus, \Gamma_2 \vdash_\Delta [e_1/u](\textbf{case } e_0 \textbf{ of box } \pi \Rightarrow e' \textbf{ else } e'') : B [C]$.

Principle 3. if $S; \Gamma_1^\triangledown \vdash_\Delta e_1 : A [D]$ and $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_2 : B[C]$, then $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B[C]$.

case $e_2 = *$, $e_2 = x$, $e_2 = u$ or $e_2 = a$.

Trivial since the substitution $[e_1/t]e_2$ is vacuous.

case $e_2 = \lambda y{:}B_1. \ e$, where $B = B_1 \to B_2$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D], y{:}B_1 \vdash_\Delta e : B_2 [C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2, y{:}B_1 \vdash_\Delta [e_1/t]e : B_2 [C]$,
3. and thus $S; \Gamma_1, \Gamma_2 \vdash_\Delta \lambda y{:}B_1. [e_1/t]e : B_1 \to B_2 [C]$.

case $e_2 = e' \ e''$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : B' \to B [C]$ and $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e'' : B' [C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : B' \to B [C]$ and $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e'' : B' [C]$.
3. Thus follows the result $S; \Gamma_1, \Gamma_2 \vdash_\Delta ([e_1/t]e') \ ([e_1/t]e'') : B [C]$.

case $e_2 = \textbf{fix } y{:}B. \ e$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D], y{:}B \vdash_\Delta e : B [C]$.

2. By induction hypothesis, $S; \Gamma_1, \Gamma_2, y{:}B \vdash_\Delta [e_1/t]e : B\,[C]$,
3. and thus $S; \Gamma_1, \Gamma_2 \vdash_\Delta \lambda y{:}B.\ [e_1/t]e : B\,[C]$.

case $e_2 = \mathbf{box}\ e'$, where $B = \Box(B'[C'])$.

1. By typing derivation, $S; \Gamma_2^{\triangledown}, t{::}A[D] \vdash_\Delta e' : B'\,[C']$.
2. Because $\Gamma_1^{\triangledown} = \Gamma_1^{\triangledown\ominus}$, by the previously proved substitution principle (Lemma 10.2), $S; \Gamma_1^{\triangledown}, \Gamma_2^{\triangledown} \vdash_\Delta [e_1/t]e' : B'\,[C']$.
3. Now assemble back into $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{box}\ ([e_1/t]e') : \Box(B'[C'])\,[C]$.

case $e_2 = \mathbf{let\ box}\ v = e'\ \mathbf{in}\ e''$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : \Box(B'[C'])\,[C]$
2. By typing derivation, either $S; \Gamma_2, t{\div}A[D], v{::}B' \vdash_\Delta e'' : B\,[C]$ or $S; \Gamma_2, t{\div}A[D], v{\div}B'[C'] \vdash_\Delta e'' : B\,[C]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$,
3. By (1) and induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : \Box(B'\,[C'])\,[C]$
4. By (2) and induction hypothesis, $S; \Gamma_1, \Gamma_2, v{::}B' \vdash_\Delta [e_1/t]e'' : B\,[C]$ or $S; \Gamma_1, \Gamma_2, v{\div}B'[C'] \vdash_\Delta [e_1/t]e'' : B\,[C]$.
5. In either case, we get the required $S; \Gamma_1, \Gamma_2 \vdash \mathbf{let\ box}\ v = [e_1/t]e'\ \mathbf{in}\ [e_1/t]e'' : B\,[C]$.

case $e_2 = a\,.\,e'$, where $B = \mathop{\text{И}}_{a:P} B'$ and $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : B'\,[C]$ and $\Delta \vdash \mathbf{fp}(B')\ \#\ a$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : B'\,[C]$.
3. Reassemble into, $S; \Gamma_1, \Gamma_2 \vdash_\Delta a\,.\,[e_1/t]e' : (\mathop{\text{И}}_{a:P} B')\,[C]$.

case $e_2 = e'\,@\,a$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : (\mathop{\text{И}}_{a:P} B)\,[C]$ and $\Delta \vdash \mathbf{fp}(B)\ \#\ a$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : (\mathop{\text{И}}_{a:P} B)\,[C]$
3. and then $S; \Gamma_1, \Gamma_2 \vdash_\Delta ([e_1/t]e')\,@\,a : B\,[C]$.

case $e_2 = \mathbf{new}\ a{:}P\ \mathbf{in}\ e'$.

1. By typing derivation, $S, a{:}P; \Gamma_2, t{\div}A[D] \vdash_{\Delta\#a} e' : B\,[C]$ and $\Delta\#a \vdash B[C]\ \#\ a$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2 \vdash_{\Delta\#a} [e_1/t]e' : B\,[C]$.
3. Conclude $S; \Gamma_1, \Gamma_2 \vdash_\Delta \mathbf{new}\ a{:}P\ \mathbf{in}\ [e_1/t]e' : B\,[C]$.

case $e_2 = \Lambda p\#K.\ e'$, where $B = \forall p\#K.\ B'$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_{\Delta, p\#K} e' : B'\,[C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_{\Delta, p\#K} [e_1/t]e' : B'\,[C]$.
3. Conclude $S; \Gamma_1, \Gamma_2 \vdash_\Delta \Lambda p\#K.\ [e_1/t]e' : (\forall p\#K.\ B')\,[C]$.

case $e_2 = e'\ [\![D']\!]$, where $B = [D'/p]B'$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : (\forall p\#K.\ B')\,[C]$, $\Delta \vdash D'\ \#\ K$, and $D' \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : (\forall p\#K.\ B')\,[C]$.
3. From here, we can reassemble, $S; \Gamma_1, \Gamma_2 \vdash_\Delta ([e_1/t]e')\ [\![D']\!] : ([D'/p]B')\,[C]$.

case $e_2 = \{a \doteq e'\}\ e''$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e' : P\,[C]$ and $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e'' : B\,[a, C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e' : P\,[C]$, and $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e'' : B\,[a, C]$.
3. Assemble back into $S; \Gamma_1, \Gamma_2 \vdash_\Delta \{a \doteq ([e_1/t]e')\}\ ([e_1/t]e'') : B\,[C]$

case subtyping from $B' \leqslant: B$.

1. By typing derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_2 : B'\,[C]$.
2. By induction hypothesis, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B'\,[C]$.
3. Use subtyping to conclude, $S; \Gamma_1, \Gamma_2 \vdash_\Delta [e_1/t]e_2 : B\,[C]$.

case $e_2 = (\textbf{case}\ e_0\ \textbf{of box}\ \pi \Rightarrow e'\ \textbf{else}\ e'')$.

1. By derivation, $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e_0 : \Box(P[D'])\,[C]$, and $S; \cdot \Vdash_\Delta \pi : P\,[D'] \Longrightarrow \Gamma'$ and $S; \Gamma_2, t{\div}A[D], \Gamma' \vdash_\Delta e' : B\,[C]$, and $S; \Gamma_2, t{\div}A[D] \vdash_\Delta e'' : B\,[C]$.
2. By induction hypothesis, $S; \Gamma_1^\triangledown, \Gamma_2 \vdash_\Delta [e_1/t]e_0 : \Box(P[D'])\,[C]$, and $S; \Gamma_1^\triangledown, \Gamma_2, \Gamma' \vdash_\Delta [e_1/t]e' : B\,[C]$, and $S; \Gamma_1^\triangledown, \Gamma_2 \vdash_\Delta [e_1/t]e'' : B\,[C]$.
3. By the typing rule for **case**, $S; \Gamma_1^\triangledown, \Gamma_2 \vdash_\Delta [e_1/t](\textbf{case}\ e_0\ \textbf{of box}\ \pi \Rightarrow e'\ \textbf{else}\ e'') : B\,[C]$.

Principle 4. if $S, a{:}P; \Gamma_1 \vdash_\Delta e_1 : P[C]$ and $S, a{:}P; \Gamma_2^\vartriangle \vdash_\Delta e_2 : B[a, C]$, then $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta \{e_1/a\}e_2 : B[C]$.

case $e_2 = *$, $e_2 = x$.

Trivial since the name substitution $\{e_1/a\}e_2$ is vacuous, and the typing $e_2 : B\,[a, C]$ must have been derived by weakening from $e_2 : B\,[\emptyset]$, and thus we can also weaken it into the typing $e_2 : B\,[C]$.

case $e_2 = a$.

Trivially obtained by weakening the hypothesis with $\Gamma_2^\vartriangle$ in the premise $S, a{:}P; \Gamma_1 \vdash e_1 : A\,[C]$.

case $e_2 = b$.

Also trivial since the substitution is vacuous, $b \in C$ and the typing $B[a, C]$ must have been obtained by weakening from $b{:}B[b]$, and can thus be weakened into $b{:}B[C]$ as well.

case $e_2 = u$.

By the definition of $\Gamma_2^\vartriangle$, it is only possible that the variable $u \in \mathbf{dom}(\Gamma_2^\vartriangle)$ if its name annotation is empty, i. e. if $u{::}B[\emptyset] \in \Gamma_2^\vartriangle$. Thus, by name dependency weakening, $S, a{:}P; \Gamma_2^\vartriangle \vdash_\Delta u : B\,[C]$. Now, by hypothesis weakening, and because $u = \{e_1/a\}u$, we get the required $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta \{e_1/a\}u : B\,[C]$.

case $e_2 = \lambda y{:}B_1.\ e'$, where $B = B_1 \to B_2$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\vartriangle, y{:}B_1 \vdash_\Delta e' : B_2\,[a, C]$.
2. By induction hypothesis, and because $\Gamma_2^\vartriangle, y{:}B_1 = (\Gamma_2, y{:}B_1)^\vartriangle$, we have $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle, y{:}B_1 \vdash_\Delta \{e_1/a\}e' : B_2\,[C]$,
3. and from here, the required, $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta \lambda y{:}B_1.\ \{e_1/a\}e' : B_1 \to B_2 : C$.

case $e_2 = e'\ e''$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\vartriangle \vdash_\Delta e' : B' \to B\,[a, C]$ and $S, a{:}P; \Gamma_2^\vartriangle \vdash_\Delta e'' : B'\,[a, C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta \{e_1/a\}e' : B' \to B\,[C]$ and also $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta \{e_1/a\}e'' : B'\,[C]$.
3. Now just assemble back into $S, a{:}P; \Gamma_1, \Gamma_2^\vartriangle \vdash_\Delta (\{e_1/a\}e')\ (\{e_1/a\}e'') : B\,[C]$.

case $e_2 = \mathbf{fix}\ y{:}B.\ e'$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\Delta, y{:}B \vdash_\Delta e' : B\,[a, C]$.
2. Because $\Gamma_2^\Delta, y{:}B = (\Gamma_2, y{:}B)^\Delta$, by induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta, y{:}B \vdash_\Delta \{e_1/a\}e' : B\,[C]$,
3. and from here, the required, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \lambda y{:}B.\ \{e_1/a\}e' : B : C$.

case $e_2 = \mathbf{box}\ e'$, and $B = \Box(B'[C'])$.

In this case, the typing $\mathbf{box}\ e' : B\,[a, C]$ is obtained by weakening inherent in the box rule. Thus, we can also derive $S, a{:}P; \Gamma_2^\Delta \vdash_\Delta \mathbf{box}\ e' : \Box(B'[C'])\,[C]$. Considering that $\{e_1/a\}\mathbf{box}\ e' = \mathbf{box}\ e'$, weaken the hypothesis context to get $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \{e_1/a\}\mathbf{box}\ e' : \Box(B'[C'])\,[C]$.

case $e_2 = \mathbf{let\ box}\ v = e'\ \mathbf{in}\ e''$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\Delta \vdash_\Delta e' : \Box(B'[C'])\,[a, C]$,
2. By typing derivation, either $S, a{:}P; \Gamma_2^\Delta, v{::}B' \vdash_\Delta e'' : B\,[a, C]$ or $S, a{:}P; \Gamma_2^\Delta, v{\div}B'[C'] \vdash_\Delta e'' : B\,[a, C]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \{e_1/a\}e' : \Box(B'[C'])\,[C]$.
4. Also notice that $\Gamma_2^\Delta, v{::}B' = (\Gamma_2, v{::}B')^\Delta$, and if $C' \neq \emptyset$, $\Gamma_2^\Delta, v{\div}B'[C'] = (\Gamma_2, v{\div}B'[C'])^\Delta$.
5. From these two equations and induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta, v{::}B' \vdash_\Delta \{e_1/a\}e'' : B\,[C]$ or $S, a{:}P; \Gamma_1, \Gamma_2^\Delta, v{\div}B'[C'] \vdash_\Delta \{e_1/a\}e'' : B\,[C]$.
6. Now, just assemble back into the required $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \mathbf{let\ box}\ v = \{e_1/a\}e'\ \mathbf{in}\ \{e_1/a\}e'' : B\,[C]$.

case $e_2 = b\,.\,e'$, where $b{:}Q \in S$ and $B = \underset{b:Q}{\text{И}}\ B'$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\Delta \vdash_\Delta e' : B'\,[a, C]$, and $\Delta \vdash \mathbf{fp}(B')\ \#\ b$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \{e_1/a\}e' : B'\,[C]$,
3. and thus $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta b\,.\,(\{e_1/a\}e') : (\underset{b:Q}{\text{И}}\ B')\,[C]$.

case $e_2 = e' \mathbin{@} b$, where $b{:}Q \in S$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\Delta \vdash_\Delta e' : (\underset{b:Q}{\text{И}}\ B)\,[a, C]$ and $\Delta \vdash \mathbf{fp}(B)\ \#\ a$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \{e_1/a\}e' : (\underset{b:Q}{\text{И}}\ B)\,[C]$.
3. Conclude $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta (\{e_1/a\}e') \mathbin{@} b : B\,[C]$.

case $e_2 = \mathbf{new}\ b{:}Q\ \mathbf{in}\ e'$.

1. By typing derivation, $S, a{:}P, b{:}Q; \Gamma_2^\Delta \vdash_{\Delta\#b} e' : B\,[a, C]$.
2. Also, $\Delta\#b \vdash B[a, C]\ \#\ b$, and thus $\Delta\#b \vdash B[C]\ \#\ b$.
3. By induction hypothesis, $S, a{:}P, b{:}Q; \Gamma_1, \Gamma_2^\Delta \vdash_{\Delta\#b} \{e_1/a\}e' : B\,[C]$,
4. leading to the required $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \mathbf{new}\ b{:}Q\ \mathbf{in}\ \{e_1/a\}e' : B\,[C]$.

case $e_2 = \Lambda p\#K.\ e'$, where $B = \forall p\#K.\ B'$.

1. By typing derivation, $S, a{:}P; \Gamma_2^\Delta \vdash_{\Delta, p\#K} e' : B'\,[a, C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_{\Delta, p\#K} \{e_1/a\}e' : B'\,[C]$.
3. Thus follows, $S, a{:}P; \Gamma_1, \Gamma_2^\Delta \vdash_\Delta \Lambda p\#K.\ \{e_1/a\}e' : (\forall p\#K.\ B')\,[C]$.

case $e_2 = e'\ [\![D]\!]$, where $B = [D/p]B'$.

1. By typing derivation, $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e' : (\forall p\#K.\ B')\,[a, C]$, $\Delta \vdash D \ \#\ K$, and $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e' : (\forall p\#K.\ B')\,[C]$.
3. Conclude, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta (\{e_1/a\}e')\ [\![D]\!] : ([D/p]B')\,[C]$.

case $e_2 = \{a \doteq e'\}\ e''$.

1. By typing derivation, $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e' : P\,[a, C]$ and $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e'' : B\,[a, a, C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e' : P\,[C]$,
3. and so $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{a \doteq \{e_1/a\}e'\}\ e'' : B\,[C]$.

case $e_2 = \{b \doteq e'\}\ e''$, where $b{:}Q \in S$.

1. By typing derivation, $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e' : Q\,[a, C]$, and $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e'' : B\,[a, b, C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e' : Q\,[C]$, and $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e'' : B\,[b, C]$.
3. Assemble back into the required $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{b \doteq \{e_1/a\}e'\}\ (\{e_1/a\}e'') : B\,[C]$.

case subtyping from $B' \leqslant: B$.

1. By typing derivation, $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e_2 : B'\,[C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e_2 : B'\,[C]$.
3. Use subtyping to conclude, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}e_2 : B\,[C]$.

case $e_2 = (\mathbf{case}\ e_0\ \mathbf{of\ box}\ \pi \Rightarrow e'\ \mathbf{else}\ e'')$.

1. By derivation, $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash e_0 : \Box(P[D])\,[a, C]$, and $S, a{:}P; \cdot \Vdash_\Delta \pi : P\,[D] \Longrightarrow \Gamma'$ and $S, a{:}P; \Gamma_2^{\blacktriangle}, \Gamma' \vdash_\Delta e' : B\,[a, C]$, and $S, a{:}P; \Gamma_2^{\blacktriangle} \vdash_\Delta e'' : B\,[a, C]$.
2. By induction hypothesis, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash \{e_1/a\}e_0 : \Box(P[D])\,[C]$, and $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash \{e_1/a\}e'' : B\,[C]$.
3. By Lemma 16, $\Gamma' = \Gamma'^{\blacktriangle}$.
4. By (1) and (3), $S, a{:}P; (\Gamma_2, \Gamma')^{\blacktriangle} \vdash_\Delta e' : B\,[a, C]$.
5. By (4) and induction hypothesis, and then (3) again, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle}, \Gamma' \vdash_\Delta \{e_1/a\}e' : B\,[C]$.
6. Finally, by (2) and (5), and the typing rule for **case**, $S, a{:}P; \Gamma_1, \Gamma_2^{\blacktriangle} \vdash_\Delta \{e_1/a\}(\mathbf{case}\ e_0\ \mathbf{of\ box}\ \pi \Rightarrow e'\ \mathbf{else}\ e'') : B\,[C]$.

■

Lemma 11 (Parametricity)

1. *if* $S; \Gamma \vdash_{\Delta, p\#K} e : A\,[C]$ *and* D *is a well-formed dependency set, i.e.* $D \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$, *and is fresh for* K, *i.e.* $\Delta \vdash D \ \#\ K$, *then*

$$S; [D/p]\Gamma \vdash_\Delta [D/p]e : ([D/p]A)\,[[D/p]C]$$

2. *if* $S; \Gamma \vdash_\Delta e : A\,[C]$, *and* $a, b{:}P$ *are names (not necessarily in* S*), then*

$$(a\ b)S; (a\ b)\Gamma \vdash_{(a\ b)\Delta} (a\ b)e : (a\ b)A\,[(a\ b)C]$$

Proof: First notice that the transposition (the second) property is trivial to prove by induction on the typing derivation for e. Namely, all the typing rules, as well as the rules for auxiliary judgments are obviously insensitive to swapping the names throughout, in all the contexts, types, terms and dependencies. Thus the judgment itself must be insensitive to swapping names. This remains true even when the language is extended with the construct **case** for pattern-matching, as shown in Lemma 17.2.

The proof of the first property is somewhat less trivial, but still rather straightforward by induction on the typing derivation of e.

case $e = *$, $e = x$, $e = u$ or $e = a$.

Substitution is vacuous on these terms, but it still may change the types. However, the given derivations are obtained using a hypothesis or a constant rule, and the substitution will change the types in the contexts as well as in the judgment.

case $e = \lambda x{:}A'.\ e'$, where $A = A' \to A''$.

1. By typing derivation, $S; \Gamma, x{:}A' \vdash_{\Delta, p\#K} e' : A''\,[C]$. and $S; \Delta, p\#K \vdash A'$ wf.
2. By induction hypothesis, $S; [D/p]\Gamma, x{:}[D/p]A' \vdash_{\Delta} [D/p]e' : [D/p]A''\,[[D/p]C]$,
3. By Lemma 2.3, $S; \Delta \vdash ([D/p]A')$ wf.
4. Combining (2) and (3), $S; [D/p]\Gamma \vdash_{\Delta} (\lambda x{:}([D/p]A').\ [D/p]e') : ([D/p]A' \to [D/p]A'')\,[[D/p]C]$.

case $e = e_1\ e_2$.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e_1 : A_1 \to A\,[C]$, and $S; \Gamma \vdash_{\Delta, p\#K} e_2 : A_1\,[C]$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta} [D/p]e_1 : ([D/p]A_1 \to [D/p]A)\,[[D/p]C]$, and $S; [D/p]\Gamma \vdash_{\Delta} [D/p]e_2 : ([D/p]A_1)\,[[D/p]C]$.
3. Conclude, $S; [D/p]\Gamma \vdash_{\Delta} ([D/p]e_1)\ ([D/p]e_2) : ([D/p]A)\,[[D/p]C]$.

case $e = \mathbf{fix}\ x{:}A.\ e'$.

1. By typing derivation, $S; \Gamma, x{:}A \vdash_{\Delta, p\#K} e' : A\,[C]$. and $S; \Delta, p\#K \vdash A$ wf.
2. By induction hypothesis, $S; [D/p]\Gamma, x{:}[D/p]A \vdash_{\Delta} [D/p]e' : ([D/p]A)\,[[D/p]C]$,
3. and by Lemma 2.3, $S; \Delta \vdash ([D/p]A)$ wf.
4. Combining the two, conclude $S; [D/p]\Gamma \vdash_{\Delta} (\mathbf{fix}\ x{:}([D/p]A).\ [D/p]e') : ([D/p]A)\,[[D/p]C]$.

case $e = \mathbf{box}\ e'$, where $A = \Box(A'[C'])$.

1. By typing derivation, $S; \Gamma^{\triangledown} \vdash_{\Delta, p\#K} e' : A'\,[C']$.
2. By induction hypothesis, $S; [D/p](\Gamma^{\triangledown}) \vdash_{\Delta} [D/p]e' : ([D/p]A')\,[[D/p]C']$.
3. Since $[D/p](\Gamma^{\triangledown}) = ([D/p]\Gamma)^{\triangledown}$, we get $S; [D/p]\Gamma \vdash_{\Delta} \mathbf{box}\ [D/p]e' : \Box([D/p](A'[C']))\,[[D/p]C]$.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e_1 : \Box(A'[C'])\,[C]$, and,
2. either $S; \Gamma, u{::}A' \vdash_{\Delta, p\#K} e_2 : A\,[C]$, or $S; \Gamma, u{\div}A'[C'] \vdash_{\Delta, p\#K} e_2 : A\,[C]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. By (1) and induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta} [D/p]e_1 : [D/p]\Box(A'[C'])\,[[D/p]C]$
4. By (2) and induction hypothesis, $S; [D/p]\Gamma, u{::}[D/p]A' \vdash_{\Delta} [D/p]e_2 : ([D/p]A)\,[[D/p]C]$, or $S; [D/p]\Gamma, u{\div}([D/p](A'[C'])) \vdash_{\Delta} [D/p]e_2 : ([D/p]A)\,[[D/p]C]$.
5. Reassemble into $S; [D/p]\Gamma \vdash_{\Delta} \mathbf{let\ box}\ u = [D/p]e_1\ \mathbf{in}\ [D/p]e_2 : ([D/p]A)\,[[D/p]C]$.

case $e = a \,.\, e'$, where $A = \underset{a:P}{\text{И}} A'$ and $a{:}P \in S$.

Assume $a \notin D$ to ensure capture avoiding. This can always be achieved by alpha-renaming a into some other fresh name.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e' : A'\,[C]$ and $\Delta, p\#K \vdash \mathbf{fp}(A') \;\#\; a$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta} ([D/p]A')\,[[D/p]C]$.
3. By Lemma 3, $\Delta \vdash \mathbf{fp}([D/p]A') \;\#\; a$.
4. Assemble back into $S; [D/p]\Gamma \vdash_{\Delta} (a \,.\, [D/p]e') : (\underset{a:P}{\text{И}}([D/p]A'))\,[[D/p]C]$.

case $e = e' \,@\, a$, where $a{:}P \in S$.

Assume, as before, that $a \notin D$, to ensure capture avoiding.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e' : (\underset{a:P}{\text{И}} A)\,[C]$. and $\Delta, p\#K \vdash \mathbf{fp}(A) \;\#\; a$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta} [D/p]e' : [D/p](\underset{a:P}{\text{И}} A)\,[[D/p]C]$.
3. By Lemma 3, $\Delta \vdash \mathbf{fp}([D/p]A) \;\#\; a$.
4. Also, $[D/p](\underset{a:P}{\text{И}} A) = \underset{a:P}{\text{И}}([D/p]A)$,
5. and we can assemble back $S; [D/p]\Gamma \vdash_{\Delta} ([D/p]e') \,@\, a : ([D/p]A)\,[[D/p]C]$.

case $e = \mathbf{new}\ a{:}P\ \mathbf{in}\ e'$.

To avoid capture, assume that a is a fresh name, not occurring in any of the variable, name, or parameter contexts.

1. By typing derivation, $S, a{:}P; \Gamma \vdash_{\Delta, p\#K} e' : A\,[C]$, and $\Delta\#a, p\#(K, a) \vdash (A[C]) \;\#\; a$.
2. By induction hypothesis, $S, a{:}P; [D/p]\Gamma \vdash_{\Delta} [D/p]e' : ([D/p]A)\,[[D/p]C]$.
3. From assumption $\Delta \vdash D \;\#\; K$ and the fact that a is fresh, we get $\Delta\#a \vdash D \;\#\; (K, a)$.
4. By Lemma 2.2, $\Delta\#a \vdash ([D/p](A[C])) \;\#\; a$, so follows the required
5. $S; [D/p]\Gamma \vdash_{\Delta} \mathbf{new}\ a{:}P\ \mathbf{in}\ [D/p]e' : ([D/p]A)\,[[D/p]C]$.

case $e = \Lambda q\#K'.\, e'$, where $A = \forall q\#K'.\, A'$.

To avoid capture, assume q is fresh.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K, q\#K'} e' : A'\,[C]$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta, q\#K'} [D/p]e' : ([D/p]A')\,[[D/p]C]$.
3. Thus, $S; [D/p]\Gamma \vdash_{\Delta} \Lambda q\#K'.\, [D/p]e' : (\forall q\#K'.\, [D/p]A')\,[[D/p]C]$.

case $e = e'\ [\![D']\!]$, where $A = [D'/q]A'$ (q will be introduced later).

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e' : (\forall q\#M.\, A')\,[C]$ where $\Delta, p\#K \vdash D' \;\#\; M$ and $D' \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta, p\#K)$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_{\Delta} [D/p]e' : (\forall q\#M.\, [D/p]A')\,[[D/p]C]$.
3. By Lemma 2.1, $\Delta \vdash ([D/p]D') \;\#\; M$.
4. Next, obviously, $([D/p]D') \subseteq \mathbf{dom}(S) \cup \mathbf{dom}(\Delta)$.
5. Thus conclude, $S; [D/p]\Gamma \vdash_{\Delta} ([D/p]e')\ [\![[D/p]D']\!] : ([D/p]A)\,[[D/p]C]$.

case $e = \{a \doteq e_1\}\ e_2$, where $a{:}P \in S$.

1. By typing derivation, $S; \Gamma \vdash_{\Delta, p\#K} e_1 : P\,[C]$, and $S; \Gamma \vdash_{\Delta, p\#K} e_2 : A\,[a, C]$.

2. By induction hypothesis, $S; [D/p]\Gamma \vdash_\Delta [D/p]e_1 : ([D/p]P)\,[[D/p]C]$, and $S; [D/p]\Gamma \vdash_\Delta [D/p]e_2 : ([D/p]A)\,[a, [D/p]C]$.
3. Since $P = [D/p]P$, reassemble into $S; [D/p]\Gamma \vdash_\Delta \{a \doteq [D/p]e_1\}\ ([D/p]e_2) : ([D/p]A)\,[[D/p]C]$.

case subtyping from $A' \leqslant: A$.

1. By typing derivation, $S; \Gamma \vdash_{\Delta,p\#K} e : A'\,[C]$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_\Delta [D/p]e : ([D/p]A')\,[[D/p]C]$.
3. By Lemma 2.5, $[D/p]A' \leqslant: [D/p]A$,
4. so by subtyping, $S; [D/p]\Gamma \vdash_\Delta [D/p]e : ([D/p]A)\,[[D/p]C]$.

case $e = (\textbf{case}\ e_0\ \textbf{of box}\ \pi \Rightarrow e'\ \textbf{else}\ e'')$.

1. By derivation, $S; \Gamma \vdash_{\Delta,p\#K} e_0 : \Box(P[D'])\,[C]$, and $S; \cdot \Vdash_{\Delta,p\#K} \pi : P\,[D'] \Longrightarrow \Gamma_1$ and $S; \Gamma, \Gamma_1 \vdash_{\Delta,p\#K} e' : B\,[C]$, and $S; \Gamma \vdash_{\Delta,p\#K} e'' : B\,[C]$.
2. By induction hypothesis, $S; [D/p]\Gamma \vdash_\Delta [D/p]e_0 : \Box(P[[D/p]D'])\,[[D/p]C]$, and $S; [D/p](\Gamma, \Gamma_1) \vdash_\Delta [D/p]e' : ([D/p]B)\,[[D/p]C]$, and $S; [D/p]\Gamma \vdash_\Delta [D/p]e'' : ([D/p]B)\,[[D/p]C]$.
3. By Lemma 17, $S; \cdot \Vdash_\Delta \pi : P\,[[D/p]D'] \Longrightarrow [D/p]\Gamma_1$.
4. By typing rule for **case**, just assemble back into the required $S; [D/p]\Gamma \vdash_\Delta \textbf{case}\ ([D/p]e_0)\ \textbf{of box}\ \pi \Rightarrow ([D/p]e')\ \textbf{else}\ ([D/p]e'') : ([D/p]B)\,[[D/p]C]$.

■

Lemma 12 (Contraction Termination and Type Preservation)
If $S_1, S_2; \Gamma^\Delta \vdash_\Delta e : A\,[C]$ then there exists unique term w, such that $e \xrightarrow{S_2} w$. Furthermore, w is S_2-contracted and $S_1, S_2; \Gamma^\Delta \vdash_\Delta w : A\,[C]$.
Proof: By induction on the derivation $S_1, S_2; \Gamma^\Delta \vdash e : A\,[C]$.

case $e = *$, $e = x$, $e = u$ or $e = a$.

For each of these cases $e \xrightarrow{S_2} e$, and they are all already contracted. So we have existence, uniqueness, contractedness, and the types are preserved.

case $e = \lambda x{:}A'.\, e'$, and $A = A' \to A''$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta, x{:}A' \vdash_\Delta e' : A''\,[C]$.
2. Since $\Gamma^\Delta, x{:}A' = (\Gamma, x{:}A')^\Delta$, by induction hypothesis there exists unique w' such that $e' \xrightarrow{S_2} w'$, and this w' is contracted and $S_1, S_2; (\Gamma, x{:}A')^\Delta \vdash_\Delta w' : A''\,[C]$.
3. Then the term $w = \lambda x{:}A'.\, w'$ satisfies all the requirements of the lemma.

case $e = e'\ e''$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash_\Delta e' : A' \to A\,[C]$, and $S_1, S_2; \Gamma^\Delta \vdash_\Delta e'' : A'\,[C]$.
2. By induction hypothesis, there are w' and w'' with the requested properties, and the term w is $w = w'\ w''$.

case $e = \textbf{fix}\ x{:}A.\, e'$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta, x{:}A \vdash_\Delta e' : A\,[C]$.
2. Since $\Gamma^\Delta, x{:}A = (\Gamma, x{:}A)^\Delta$, by induction hypothesis there exists unique w' such that $e' \xrightarrow{S_2} w'$, and this w' is contracted and $S_1, S_2; (\Gamma, x{:}A)^\Delta \vdash_\Delta w' : A\,[C]$.

3. Then the term $w = \mathbf{fix}\ x{:}A.\ w'$ satisfies all the requirements of the lemma.

case $e = \mathbf{box}\ e'$, where $A = \Box(A'[C'])$.

This is actually one of the base cases (together with hypotheses and constants) – boxed expressions contract to themselves.

case $e = \mathbf{let\ box}\ u = e'\ \mathbf{in}\ e''$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash e' : \Box(A'[C'])\,[C]$, and,
2. either $S_1, S_2; \Gamma^\Delta, u{::}A' \vdash e' : \Box(A'[C'])\,[C]$, or $S_1, S_2; \Gamma^\Delta, u{\div}A'[C'] \vdash e' : \Box(A'[C'])\,[C]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. Also notice that $(\Gamma^\Delta, u{::}A') = (\Gamma, u{::}A')^\Delta$, and if $C' \neq \emptyset$, then $(\Gamma^\Delta, u{\div}A'[C']) = (\Gamma, u{\div}A'[C'])^\Delta$.
4. Then, by induction hypothesis, we have w' and w'' satisfying the prescribed properties. Combine them into $w = \mathbf{let\ box}\ u = w'\ \mathbf{in}\ w''$.

case $e = a\,.\,e'$, where $A = \underset{a:P}{\text{И}}\,A'$, and $a{:}P \in S_1, S_2$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash_\Delta e' : A'\,[C]$, and $\Delta \vdash \mathbf{fp}(A')\ \#\ a$.
2. By induction hypothesis, there is unique w' such that $e' \xrightarrow{S_2} w'$ and w' is contracted and of the same type A'.
3. Now, pick $w = a\,.\,w'$.

case $e = e' \mathbin{@} a$, where $a{:}P \in S_1, S_2$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash_\Delta e' : (\underset{a:P}{\text{И}}\,A')\,[C]$, and $\Delta \vdash \mathbf{fp}(A)\ \#\ a$.
2. By induction hypothesis, there is unique w' such that $e' \xrightarrow{S_2} w'$ and w' is contracted and of the same type $(\underset{a:P}{\text{И}}\,A')$.
3. Now, pick $w = w' \mathbin{@} a$.

case $e = \mathbf{new}\ a{:}P\ \mathbf{in}\ e'$.

1. By typing derivation, $S_1, (S_2, a{:}P); \Gamma_\Delta \vdash e' : A\,[C]$, and $\Delta \# a \vdash A[C]\ \#\ a$.
2. By induction hypothesis, there is unique w' such that $e' \xrightarrow{S_2, a:P} w'$. This w' is also contracted and of type A.
3. Pick $w = \mathbf{new}\ a{:}P\ \mathbf{in}\ w'$, and it has the required properties.

case $e = \Lambda p \# K.\ e$ and $e = e'\ [\![D]\!]$.

Just as the previous cases, these two also go easily.

case $e = \{a \doteq e'\}\ e''$, where $a{:}P \in S_1, S_2$.

1. By typing derivation, $S_1, S_2; \Gamma^\Delta \vdash_\Delta e' : P\,[C]$, and $S_1, S_2; \Gamma^\Delta \vdash_\Delta e'' : A\,[a, C]$.
2. By induction hypothesis, there are unique w' and w'' such that $e' \xrightarrow{S_2} w'$ and $e'' \xrightarrow{S_2} w''$, plus they are contracted and preserve the types.
3. Now, distinguish two cases: (1) $a \in \mathbf{dom}(S_2)$, and (2) $a \notin \mathbf{dom}(S_2)$.
4. In the first case, pick $w = \{a \doteq w'\}\ w''$. It is contracted and has the correct typing.
5. In the second case, pick $w = \{w'/a\}w''$. By the contraction rules, $e \xrightarrow{S_2} w$. By Lemma 13.1, it is contracted. By substitution principle (Lemma 10.4), it also has the correct typing $S_1, S_2; \Gamma^\Delta \vdash_\Delta \{w'/a\}w'' : A\,[C]$.

case subtyping from $A' \leqslant: A$.

1. By typing derivation, $S_1, S_2; \Gamma^{\Delta} \vdash_{\Delta} e : A' [C]$.
2. By induction hypothesis, there exists w with the required properties. In particular, w' has the same type, i. e. $S_1, S_2; \Gamma^{\Delta} \vdash_{\Delta} w : A' [C]$,
3. so use subtyping to conclude, $S_1, S_2; \Gamma^{\Delta} \vdash_{\Delta} w : A [C]$.

■

Theorem 15 (Progress and Type Preservation)
If $S; \cdot \vdash e : A[\,]$, then either

1. *e is a value, or*
2. *there exists $S' \supseteq S$ such that $\mu_S, e \longmapsto \mu_{S'}, e'$; furthermore e' is unique and $S'; \cdot \vdash e' : A[\,]$.*

Proof: By induction on the typing derivation $S; \cdot \vdash e : A[\,]$.

case $e = x$, $e = u$ or $e = a$.

These are not applicable, because both the hypothesis context and the name annotations in the above typing derivation are empty.

case $e = *$ or $e = \lambda x.\, e'$.

Both terms are already values.

case $e = e_1\ e_2$.

Assume both e_1 and e_2 are values (otherwise trivial).

1. By typing derivation, $S; \cdot \vdash e_1 : A' \to A[\,]$, and $S; \cdot \vdash e_2 : A'[\,]$.
2. Then $e_1 = \lambda x.\, e'$, and $\mu_S, e \longmapsto \mu_S, [e_2/x]e'$.
3. By substitution principle 10.1, the reduct has the same typing.

case $e = \textbf{fix}\ x{:}A.\, e'$.

1. By typing derivation, $S; x{:}A \vdash e' : A[\,]$,
2. and so, by substitution principle 10.1, $S; \vdash [e/x]e' : A[\,]$.
3. Thus, $S' = S$ satisfies the requirements of the theorem.

case $e = \textbf{box}\ e'$, where $A = \Box(A'[C'])$.

Assume that e' is not contracted (otherwise e is a value).

1. By typing derivation, $S; \cdot \vdash e' : A'[C']$.
2. By Lemma 12, there exists unique w' such that $e' \to w'$, which in addition has the typing $S; \cdot \vdash w' : A'[C']$.
3. By reduction rules, **box** e' reduces exactly to **box** w', and so the typing is preserved.

case $e = \textbf{let box}\ u = e_1\ \textbf{in}\ e_2$.

Assume that e_1 is a value (otherwise trivial). In that case $e_1 = \textbf{box}\ w_1$, where w_1 is contracted, and $\mu_S, e \longmapsto \mu_S, [w_1/u]e_2$.

1. By typing derivation, $S; \cdot \vdash w_1 : A'[C']$, and,
2. either $S; u{::}A' \vdash e_2 : A[\,]$, or $S; u{\div}A'[C'] \vdash e_2 : A[\,]$, depending whether $C' = \emptyset$ or $C' \neq \emptyset$.
3. Conclude, using either Lemma 10.2, or Lemma 10.3, that $S; \cdot \vdash [w_1/u]e_2 : A[\,]$.

case $e = a \,.\, e'$, where $A = (\mathop{\text{И}}_{a:P} A')$ and $a{:}P \in S$.

Assume that e' is not a value (otherwise e is a value itself).

1. By typing derivation, $S; \cdot \vdash e' : A' [\,]$, and $\cdot \vdash \mathbf{fp}(A') \;\#\; a$ (because $\mathbf{fp}(A') = \emptyset$).
2. By induction hypothesis, there exists $S' \supseteq S$ such that $\mu_S, e' \longmapsto \mu_{S'}, e''$ and $S'; \cdot \vdash e'' : A' [\,]$.
3. By reduction rules, $\mu_S, a \,.\, e' \longmapsto \mu_{S'}, a \,.\, e''$, and
4. since $S'; \cdot \vdash a \,.\, e'' : \mathop{\text{И}}_{a:P} A' [\,]$, the typing is preserved.

case $e = e' \,@\, a$, where $a{:}P \in S$.

Assume that e' is a value (otherwise, trivial).

1. By typing derivation, $S; \cdot \vdash e' : (\mathop{\text{И}}_{a:P} A) [\,]$, and $\cdot \vdash \mathbf{fp}(A) \;\#\; a$ (because $\mathbf{fp}(A) = \emptyset$).
2. Then $e' = b \,.\, v'$, where $b{:}P \in S$, and $S; \cdot \vdash v' : (a\ b)A [\,]$.
3. By reduction rules, $\mu_S, e \longmapsto \mu_S, (a\ b)v'$.
4. By Lemma 11.2, $(a\ b)S; \cdot \vdash (a\ b)v' : (a\ b)(a\ b)A [\,]$.
5. Now, both $a, b \in \mathbf{dom}(S)$, so $(a\ b)S = S$.
6. By idempotency of swapping, $S; \cdot \vdash (a\ b)v' : A [\,]$, and the typing is preserved.

case $e = \mathbf{new}\ a{:}P\ \mathbf{in}\ e'$, where $a{:}P \notin S$.

1. By typing derivation, $S, a{:}P; \cdot \vdash e' : A [\,]$.
2. By reduction rules, $\mu_S, e \longmapsto (\mu_S, a), e'$, so indeed, $S' = (S, a{:}P)$ satisfies the requirements.

case $e = \Lambda p\#K.\, e'$.

Trivial; e is already a value.

case $e = e' \llbracket D \rrbracket$.

Assume e' is a value (otherwise trivial).

1. By typing derivation, $S; \cdot \vdash e' : \forall p\#K.\, A'$, where $\vdash D \;\#\; K$, and $A = [D/p]A'$.
2. Since e' is a value, it is of the form $e' = \Lambda p\#M.\, e''$, where $M \subseteq K$ and thus $\vdash D \;\#\; M$.
3. By typing rules, $S; \cdot \vdash_{p\#M} e'' : A' [\,]$.
4. By Lemma 11.1, $S; \cdot \vdash [D/p]e'' : ([D/p]A') [\,]$,
5. Since $\mu_S, e \longmapsto \mu_S, [D/p]e''$, we have just shown that the typing is preserved.

case $e = \{a \doteq e_1\}\ e_2$, where $a{:}P \in S$.

Assume both e_1 is a value (otherwise trivial).

1. By typing derivation, $S; \cdot \vdash e_1 : P [\,]$, and $S; \cdot \vdash e_2 : A [a]$.
2. By Lemma 10.4, $S; \cdot \vdash \{e_1/a\}e_2 : A [\,]$.
3. By reduction rules, $\mu_S, e \longmapsto \mu_S, \{e_1/a\}e_2$, so the statement is proved.

case subtyping from $A' \leqslant: A$.

1. By induction hypothesis, there exists $S' \supseteq S$ such that $\mu_S, e \longmapsto \mu_{S'}, e'$ and $S'; \cdot \vdash e' : A' [\,]$.
2. By subsumption, we also have $S'; \cdot \vdash e' : A [\,]$, i. e. the types are preserved.

case $e = (\mathbf{case}\ e_0\ \mathbf{of}\ \mathbf{box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2)$.

If e_0 is not a value, the case is trivial. So assume that e_0 is a value, i.e. is of the form $e_0 = \mathbf{box}\ w$ for some reduced w.

1. By typing derivation, $S; \cdot \Vdash \pi \Longrightarrow \Gamma_1$, and $S; \Gamma_1 \vdash e_1 : A\,[\,]$ and $S; \cdot \vdash e_2 : A\,[\,]$.
2. If there exists μ' and Θ such that $\mu_S, w \rhd \pi \Longrightarrow \mu', \Theta$, then $e' = \Theta(e_1)$.
3. In such a case, by type preservation for pattern-matching (Lemma19), there exists $S' \supseteq S$ such that $\mu' = \mu_{S'}$ and $S' \vdash_\Delta \Theta : \Gamma_1$.
4. From (1) and (3), by substitution principle for value variables (Lemma 10.1), $S'; \cdot \vdash_\Delta \Theta(e_1) : A\,[\,]$. Here the substitution principle is applied once for every variable in the domain of Θ.
5. If no μ' and Θ exist, then $e' = e_2$.
6. From (1) by name-context weakening, $S'; \cdot \vdash_\Delta e_2 : A\,[\,]$.
7. In any case, from (4) and (6), we have $S; \cdot \vdash_\Delta e' : A\,[\,]$.

■