Meta-programming with Names and Necessity

Aleksandar Nanevski

November 2002
CMU-CS-02-123R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report is a significant revision of and supersedes the version published in April 2002 under the number CMU-CS-02-123

Abstract

Meta-programming is a discipline of writing programs in a certain programming language that generate, manipulate or execute programs written in another language. In a typed setting, meta-programming languages usually contain a modal type constructor to distinguish the level of object programs (which are the manipulated data) from the meta programs (which perform the computations). In functional programming, modal types of object programs generally come in two flavors: open and closed, depending on whether the expressions they classify may contain any free variables or not. Closed object programs can be executed at run-time by the meta program, but the computations over them are more rigid, and typically produce less efficient residual code. Open object programs provide better inlining and partial evaluation, but once constructed, expressions of open modal type cannot be evaluated.

Recent work in this area has focused on combining the two notions into a sound type system. We present a novel calculus to achieve this, which we call $\nu^{\Box}$. It is based on adding the notion of *names* inspired by 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, we extend our calculus with primitives for inspection and destruction of object programs at run-time in a type-safe manner.

Keywords: meta-programming, modal logic, nominal logic, higher-order abstract syntax, staged computation, run-time code generation, symbolic computation

Contents

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).

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. This is the idea behind the work on run-time code generation [LL96, WLP98, WLPD98] and the functional programming concept of staged computation [DP01].

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 object 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 $\lambda^{\Box}$-calculus is the proof-term language for the modal logic S4, whose necessity constructor $\Box$ annotates *valid* propositions [DP01, PD01]. The type $\Box A$ has been used in run-time code generation to classify generators of code of type A [WLP98, WLPD98]. The $\lambda^{\bigcirc}$-calculus is the proof-term language for discrete linear temporal logic, and the type $\bigcirc A$ classifies terms associated with the subsequent time moment. The intended application of $\lambda^{\bigcirc}$ is in partial evaluation because the typing annotation of a $\lambda^{\bigcirc}$-program can be seen as its binding-time specification [Dav96]. Both calculi provide a distinction between levels (or stages) 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. For purposes of meta-programming, the type $\Box A$ is also associated with *closed code* – it classifies closed object terms of type A. On the other hand, the type $\bigcirc A$ is the type of *postponed* code, because it classifies object terms of type A which are associated with the subsequent time moment. The important property of $\lambda^{\bigcirc}$ is that its terms at a certain temporal level n may refer to variables which are on the same temporal level n. Because these variables can be predeclared in the context, the postponed code type of $\lambda^{\bigcirc}$ is frequently conflated with the notion of *open code*. The abstract concept of open code (not necessarily that of $\lambda^{\bigcirc}$) is obviously more general than that of closed code, and it is certainly desirable to endow a meta-programming language with it. As already observed in [Dav96] in the context of $\lambda^{\bigcirc}$, working with open code is more flexible and results in better and more optimized residual programs. However, we also want to run the generated object programs when they are closed, and unfortunately, the modal type of $\lambda^{\bigcirc}$ does not provide for this.

There have been several proposed type systems which incorporate the the advantages from both languages, most notable being MetaML [MTBS99, Tah99, CMT00, CMS01]. MetaML defines its notion of open code to be that of the postponed code of $\lambda^{\bigcirc}$ and then introduces closed code as a refinement – as open code which happens to contain no free variables. Our calculus, which we call $\nu^{\Box}$, has the opposite approach. Rather than refining the notion of postponed code of $\lambda^{\bigcirc}$, we relax the notion of closed code of $\lambda^{\Box}$. 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 inspired by the work on Nominal Logic and FreshML by Pitts and Gabbay [GP02, 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 a fragment of our language. Thus, we can also treat our object expressions as higher-order syntactic 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 rest of the document is organized as follows: Section 2 is a brief exposition of the previous work on $\lambda^{\Box}$. The type system of $\nu^{\Box}$ and its properties are described in Section 3, while Section 4 describes parametric polymorphism in sets of names. Intensional analysis of higher-order syntax is introduced in Section 5. Finally, we illustrate the type system with example programs, before discussing the related work in Section 6. This paper supersedes the previously published [Nan02].

2 Modal $\lambda^\Box$-calculus

This section reviews the previous work on the modal $\lambda^\Box$-calculus and its use in meta-programming to separate, through the mechanism of types, the realms of meta-level programs and object-level programs. The $\lambda^\Box$-calculus is the proof-term calculus for the necessitation fragment of modal logic S4 [PD01, DP01]. Chronologically, it came to be considered in functional programming in the context of specialization for purposes of run-time code generation [WLP98, WLPD98]. For example, consider the exponentiation function, presented below in ML-like notation.

```
fun exp1 (n : int) (x : int) : int =
    if n = 0 then 1 else x * exp1 (n-1) x
```

The function `exp1 : int -> int -> int` is written in curried form so that it can be applied when only a part of its input is known. For example, if an actual parameter for n is available, `exp1(`n`)` returns a function for computing the n-th power of its argument. In a practical implementation of this scenario, however, the outcome of the partial instantiation will be a closure waiting to receive an actual parameter for x before it proceeds with evaluation. Thus, one can argue that the following reformulation of `exp1` is preferable.

```
fun exp2 (n : int) : int -> int =
    if n = 0 then λx:int.1
    else
        let val u = exp2 (n - 1)
        in
             λx:int. x * u(x)
        end
```

Indeed, when only n is provided, but not x, the expression `exp2(`n`)` performs computation steps based on the value of n to produce a residual function specialized for computing the n-th power of its argument. In particular, the obtained residual 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 specialization.

A useful intuition for understanding the programming idiom of the above example, is to view `exp2` as a program generator; once supplied with n, it *generates* the specialized function for computing n-th powers. This immediately suggests a distinction in the calculus between two stages (or levels): the meta and the object stage. The object stage of an expression encodes λ-terms that are to be viewed as data – as results of a process of code generation. In the `exp2` function, such terms would be `(λx:int.1)` and `(λx:int. x * u(x))`. The meta stage describes the specific operations to be performed over the expressions from the object stage. This is why the above-illustrated programming style is referred to as *staged computation*.

The idea behind the type system of $\lambda^\Box$ is to make explicit the distinction between meta and object stages. It allows the programmer to specify the intended staging of a term by annotating object-level subterms of the program. Then the type system can check whether the written code conforms to the staging specifications, making staging errors into type errors. The syntax of $\lambda^\Box$ is presented below.

Types	A	$::=$	$b \mid A_1 \to A_2 \mid \Box A$
Terms	e	$::=$	$c \mid x \mid u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$
Value variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Expression variable contexts	Δ	$::=$	$\cdot \mid \Delta, u{:}A$
Values	v	$::=$	$c \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e$

There are several distinctive features of the calculus, arising from the desire to differentiate between the stages. The most important is the new type constructor $\Box$. It is usually referred to as *modal necessity*, as on the logic side it is a necessitation modifier on propositions [PD01]. In our meta-programming application, it is used to classify object-level terms. Its introduction and elimination forms are the term constructors **box** and **let box**, respectively. If e is an object term of type A, then **box** e would be a meta term of type $\Box A$. The **box** term constructor encases the object term e so that it can be accessed and manipulated by the meta part of the program. The elimination form **let box** $u = e_1$ **in** e_2 does the opposite; it takes the object term encased by e_1 and binds it to the variable u to be used in e_2. Notice that the semantics of $\lambda^\Box$ (presented in Figure 1) distinguishes between two kinds of variables, and consequently has two variable contexts: Γ for variables bound to meta terms, and Δ for variables bound to object terms. We can now use the type system of $\lambda^\Box$ to make explicit the staging of `exp2`.

```
fun exp3 (n : int) : □(int->int) =
    if n = 0 then box (λx:int. 1)
    else
        let box u = exp3 (n - 1)
        in
            box (λx:int. x * u(x))
        end
```

Application of `exp3` at argument 2 produces an object-level function for squaring.

```
- sqbox = exp3 2;
val sqbox = box (λx:int. x *
                     (λy:int. y *
                        (λz:int. 1) y) x) : □(int -> int)
```

In the elimination form **let box** $u = e_1$ **in** e_2, the bound variable u belongs to the context Δ of object-level variables, but it can be used in e_2 in both object positions (i.e. under a box) and meta positions. This way the calculus is capable of expressing not only composition of object programs, but also their evaluation. For example we can use the generated function `sqbox` in the following way.

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

This example demonstrates that object expressions of $\lambda^\Box$ can be *reflected*, i.e. coerced from the object-level into the meta-level. The opposite coercion which is referred to as *reification*, however, is not possible. This suggests that $\lambda^\Box$ should be given a more specific model in which reflection naturally exists, but reification does not. *A possible interpretation exhibiting this behavior considers object terms as actual syntactic expressions, or abstract syntax trees of source programs of the calculus, while the meta terms are compiled executables.* Because $\lambda^\Box$ is typed, in this scenario the object terms represent not only syntax, but *higher-order syntax* as well. The operation of reflection corresponds to the natural process of compiling source code into an executable. The opposite operation of reconstructing source code out of its compiled equivalent is not usually feasible, so this interpretation does not support reification, just as required.

3 Modal calculus of names

3.1 Motivation, syntax and overview

The $\lambda^\Box$ staging of `exp3` from the previous section, is somewhat displeasing. For example, the residual programs that `exp3` produces, e.g. `sqbox`, contain unnecessary variable-for-variable redexes, and hence are not as optimal as one would want. This may not be a serious criticism from the perspective of code generation, but it certainly is severe if we adhere to the interpretation of object terms as higher-order syntax. It exhibits the fact that $\lambda^\Box$ is too restrictive to allow for arbitrary composition of higher-order syntax trees. The reason for this deficiency is that $\lambda^\Box$ requires that its boxed object terms must always be *closed*. In that sense, the type $\Box A$ is a type of closed syntactic expressions of type A. As can be observed from the typing rules in Figure 1, the $\Box$-introduction rule erases all the meta variables before typechecking the argument term. It allows for object level variables, but in run-time they are always substituted by other closed object expressions to produce a closed object expression at the end. Worse yet, if we only have a type of closed syntactic expressions at our disposal, we can't even type the *body* of a λ-abstraction in isolation from the

Typechecking

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

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

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

Operational semantics

$$\frac{e_1 \longmapsto e_1'}{e_1\ e_2 \longmapsto e_1'\ e_2} \qquad \frac{e_2 \longmapsto e_2'}{v_1\ e_2 \longmapsto v_1\ e_2'} \qquad \frac{}{(\lambda x{:}A.\ e)\ v \longmapsto [v/x]e}$$

$$\frac{e_1 \longmapsto e_1'}{\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \longmapsto \mathbf{let\ box}\ u = e_1'\ \mathbf{in}\ e_2} \qquad \frac{}{\mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 \longmapsto [e_1/u]e_2}$$

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

λ-binder itself – subterms of a closed term are not necessarily closed themselves. Thus, it would be impossible to ever inspect, destruct or recurse over object-level expressions with binding structure.

The solution should be to extend the notion of object level to include not only closed syntactic expressions, but also expressions with free variables. This need has long been recognized in the meta-programming community, and Section 6 discusses several different meta-programming systems and their solutions to the problem. The technique predominantly used in these solutions goes back to the Davies' $\lambda^{\bigcirc}$-calculus [Dav96]. The type constructor $\bigcirc$ of this calculus corresponds to discrete temporal logic modality for propositions true at the subsequent time moment. In meta-programming setup, the modal type $\bigcirc A$ stands for open object expression of type A, where the free variables of the object expression are modeled by meta-variables from the subsequent time moment, bound somewhere outside of the expression.

Our $\nu^{\Box}$-calculus adopts a different approach. It seems that for purposes of higher-order syntax, one cannot equate bound meta-variables with free variables of object expressions. For, imagine recursing over two syntax trees with binding structure to compare them for syntactic equality modulo α-conversion. Whenever a λ-abstraction is encountered in both expressions, we need to introduce a new entity to stand for the bound variable of that λ-abstraction, and then recursively proceed comparing the bodies of the abstractions. But then, introducing this new entity standing for the λ-bound variable must not change the type of the surrounding term. In other words, free variables of object expressions cannot be introduced into the computation as λ-bound variables, as it is the case in $\lambda^{\bigcirc}$ and other languages based on it.

Thus, we start with the $\lambda^{\Box}$-calculus, and introduce a separate semantic category of names, motivated by the works of Pitts and Gabbay [PG00, GP02], and also of Odersky [Ode94]. Just as before, object and meta stages are separated through the $\Box$-modality, but now object terms can use names to encode abstract syntax trees with free variables. The names appearing in an object term will be apparent from its type. In addition, the type system must be instrumented to keep track of the occurrences of names, so that the names are prevented from slipping through the scope of their introduction form.

Informally, a term *depends* on a certain name if that name appears in the meta-level part of the term. The set of names that a term depends on is called the *support* of the term. The situation is analogous to that in polynomial algebra, where one is given a base structure S and a set of indeterminates (or generators) I and then freely adjoins S with I into a structure of polynomials. In our setup, the indeterminates are the names, and we build "polynomials" over the base structure of $\nu^{\Box}$ expressions. For example, assuming for a moment that X and Y are names of type *int*, and that the usual operations of addition, multiplication and exponentiation of integers are primitive in $\nu^\Box$, the term

$$e_1 = X^3 + 3X^2Y + 3XY^2 + Y^3$$

would have type *int* and support set $\{X, Y\}$. The names X and Y appear in e_1 at the meta level, and indeed, notice that in order to evaluate e_1 to an integer, we first need to provide definitions for X and Y. On the other hand, if we box the term e_1, we obtain

$$e_2 = \textbf{box } (X^3 + 3X^2Y + 3XY^2 + Y^3)$$

which has the type $\Box(int[X, Y])$, but its support is the empty set, as the names X and Y only appear at the object level (i.e. under a box). Thus, the support of a term (in this case e_1) becomes part of the type once the term itself is boxed. This way, the types maintain the information about the support of subterms at all stages. For example, assuming that our language have pairs, the term

$$e_3 = \langle X^2, \textbf{box } Y^2\rangle$$

would have the type $int \times \Box(int[Y])$ with support $\{X\}$.

We are also interested in compiling and evaluating syntactic entities in $\nu^\Box$ when they have empty support (i.e. when they are closed). Thus, we need a mechanism to eliminate a name from a given expression's support, eventually turning unexecutable expressions into executable ones. For that purpose, we use explicit substitutions. An explicit substitution provides definitions for names which appear at a meta-level in a certain expression. Notice the emphasis on the meta-level; explicit substitutions do not substitute under boxes, as names appearing at the object level of a term do not contribute to the term's support. This way, explicit substitutions provide *extensions*, i.e., definitions for names, while still allowing names under boxes to be used for the *intensional* information of their identity (which we utilize in Section 5).

We next present the syntax of the $\nu^\Box$-calculus and discuss each of the constructors.

$$
\begin{array}{llcl}
\textit{Names} & X & \in & \mathcal{N} \\
\textit{Support sets} & C, D & \in & \mathcal{N}^* \\
\textit{Types} & A & ::= & b \mid A_1 \to A_2 \mid A_1 \nrightarrow A_2 \mid \Box_C A \\
\textit{Explicit substitutions} & \Theta & ::= & \cdot \mid X \to e, \Theta \\
\textit{Terms} & e & ::= & c \mid X \mid x \mid \langle\Theta\rangle u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \\
 & & & \textbf{box } e \mid \textbf{let box } u = e_1 \textbf{ in } e_2 \mid \\
 & & & \nu X{:}A.\ e \mid \textbf{choose } e \\
\textit{Value variable contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\textit{Expression variable contexts} & \Delta & ::= & \cdot \mid \Delta, u{:}A[C] \\
\textit{Name contexts} & \Sigma & ::= & \cdot \mid \Sigma, X{:}A
\end{array}
$$

Just as $\lambda^\Box$, our calculus makes a distinction between meta and object levels, which here too are interpreted as the level of compiled code and the level of source code (or abstract syntax expressions), respectively. The two levels are separated by a modal type constructor $\Box$, except that now we have a whole family of modal type constructors – one for each *finite* set of names C. In that sense, values of the type $\Box_C A$ are the abstract syntax trees of the calculus freely generated over the set of names C. We refer to the finite set C as a *support set* of thus generated syntax trees. All the names are drawn from a countably infinite universe of names $\mathcal{N}$.

As before, the distinction in levels forces a split in the variable contexts. We have a context Γ for meta-level variables (we will also call them compiled code variables, or value variables), and a context Δ for object-level variables (which we also call syntactic expression variables, or just expression variables). The context Δ must keep track not only of the typing of a given variable, but also of the support set that syntactic expression bound to that variable is allowed to have.

The set of terms includes the syntax of the $\lambda^\Box$-calculus from Section 2. However, there are two important distinctions in $\nu^\Box$. First, we can now explicitly refer to names on the level of terms. Second, it is required that all the references to expression variables that a certain term makes are always prefixed by some explicit substitution. For example, if u is an expression variable bound by some **let box** $u = e_1$ **in** e_2 term, then u can only appear in e_2 prefixed by an explicit substitution Θ (and different occurrences of u can have different substitutions associated with them). The explicit substitution is supposed to provide definitions for names in the expression bound to u. When the reference to the variable u is prefixed by an empty substitution, instead of $\langle\cdot\rangle u$ we will simply write u. The explicit substitutions used in $\nu^\Box$-calculus are simultaneous substitutions. We assume that the syntactic presentation of a substitution never defines a denotation for the same name twice.

Example 1 Assuming that X and Y are names of type `int`, the code segment below creates a polynomial over X and Y and then evaluates it at the point $(X = 1, Y = 2)$.

```
- let box u = box (X³ + 3X²Y + 3XY² + Y³)
  in
      ⟨X -> 1, Y -> 2⟩ u
  end

val it = 27 :  int
```

■

The terms $\nu x{:}A.\ e$ and **choose** e are the introduction and elimination form for the type constructor $A \nrightarrow B$. The term $\nu X{:}A.\ e$ binds a name X of type A that can subsequently be used in e. The term **choose** picks a fresh name of type A, substitutes it for the name bound in the argument ν-abstraction of type $A \nrightarrow B$, and proceeds to evaluate the body of the abstraction. To prevent the bound name in $\nu X{:}A.\ e$ from escaping the scope of its definition and thus creating an observable effect, the type system will enforce a discipline on the occurrence of X in e; X can appear in e only in the scope of some explicit substitution which provides it with a definition, or in computationally irrelevant (i.e. dead code) positions.

Finally, enlarging an appropriate context by a new variable or a name is subject to Barendregt's Variable Convention: the new variables and names are assumed distinct, or are renamed in order not to clash with already existing ones. Terms which differ only in the syntactic representation of their bound variables and names are considered equal. The binding forms in the language are $\lambda x{:}A.\ e$, **let box** $u = e_1$ **in** e_2 and $\nu X{:}A.\ e$. As usual, capture-avoiding substitution $[e_1/x]e_2$ of expression e_1 for the variable x in the expression e_2 is defined to rename bound variables and names when descending into their scope. Given a term e, we denote by $\mathbf{fv}(e)$ and $\mathbf{fn}(e)$ the set of free variables of e and the set of names appearing in e at the meta-level. In addition, we overload the function **fn** so that given a type A and a support set C, $\mathbf{fn}(A[C])$ is the set of names appearing in A or C.

Example 2 To illustrate our new constructors, we present a version of the staged exponentiation function that we can write in $\nu^\Box$-calculus. In this and in other examples we resort to concrete syntax in ML fashion, and assume the presence of the base type of integers, recursive functions and `let`-definitions. In any case, these additions do not impose any theoretical difficulties.

```
fun exp (n : int) : □(int -> int) =
    choose (νX : int.
            let fun exp' (m : int) : □_X int =
                if m = 0 then box 1
                else
                    let box u = exp' (m - 1)
                    in
                        box (X * u)
                    end
            in
                let box v = exp' (n)
                in
                    box (λx:int. ⟨X -> x⟩ v)
                end
            end)
```

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

The function exp takes an integer n and generates a fresh name X of integer type. Then it calls the helper function exp' to build the expression $v = \underbrace{X * \cdots * X}_{n} * 1$ of type int and support $\{X\}$. Finally, it turns the expression v into a function by explicitly substituting the name X in v with a newly introduced bound variable x. Notice that the generated residual code for sq does not contain any unnecessary redexes, in contrast to the $\lambda^{\Box}$ version of the program from Section 2. ■

3.2 Explicit substitutions

In this section we formally introduce the concept of explicit substitution over names and define related operations. As already outlined before, substitutions will serve to provide definitions for names, thus effectively removing the substituting names from the support of the term in which they appear. Once the term has empty support, it can be compiled and evaluated.

Definition 1 (Explicit substitution, its domain and range)
An explicit substitution is a function from the set of names to the set of terms

$$\Theta : \mathcal{N} \to \text{Terms}$$

Given a substitution Θ, *its domain* $\mathbf{dom}(\Theta)$ *is the set of names that the substitution does not fix. In other words*

$$\mathbf{dom}(\Theta) = \{X \in \mathcal{N} \mid \Theta\ (X) \neq X\}$$

Range of a substitution Θ *is the image of* $\mathbf{dom}(\Theta)$ *under* Θ*:*

$$\mathbf{range}(\Theta) = \{\Theta\ (X) \mid X \in \mathbf{dom}(\Theta)\}$$

For the purposes of this work, we only consider substitutions with *finite* domains. A substitution Θ with a finite domain has a finitary syntactical representation as a set of ordered pairs $X \to e$, relating a name X from $\mathbf{dom}(\Theta)$, with its substituting expression e. The opposite also holds – any *finite and functional* set of ordered pairs of names and expressions determines a unique substitution. We will frequently equate a substitution and its syntactic representation when it does not result in ambiguities. Just as customary, we denote by $\mathbf{fv}(\Theta)$ the set of free variables in the terms from $\mathbf{range}(\Theta)$. The set of names appearing in $\mathbf{range}(\Theta)$ is denoted by $\mathbf{fn}(\Theta)$.

Each substitution can be uniquely extended to a function over arbitrary terms in the following way.

Definition 2 (Substitution application)
Given a substitution Θ *and a term* e, *the operation* $\{\Theta\}e$ *of applying* Θ *to the* meta level *of* e *is defined recursively on the structure of* e *as given below. The substitution application is capture-avoiding.*

$$
\begin{array}{llcll}
\{\Theta\} & X & = & \Theta(X) & \\
\{\Theta\} & x & = & x & \\
\{\Theta\} & (\langle\Theta'\rangle u) & = & \langle\Theta \circ \Theta'\rangle u & \\
\{\Theta\} & (\lambda x{:}A.\ e) & = & \lambda x{:}A.\ \{\Theta\}e & x \notin \mathbf{fv}(\Theta) \\
\{\Theta\} & (e_1\ e_2) & = & \{\Theta\}e_1\ \{\Theta\}e_2 & \\
\{\Theta\} & (\mathbf{box}\ e) & = & \mathbf{box}\ e & \\
\{\Theta\} & (\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let\ box}\ u = \{\Theta\}e_1\ \mathbf{in}\ \{\Theta\}e_2 & u \notin \mathbf{fv}(\Theta) \\
\{\Theta\} & (\nu X{:}A.\ e) & = & \nu X{:}A.\ \{\Theta\}e & X \notin \mathbf{fn}(\Theta) \\
\{\Theta\} & (\mathbf{choose}\ e) & = & \mathbf{choose}\ \{\Theta\}e &
\end{array}
$$

The most important aspect of the above definition is that substitution application does not recursively descend under **box**. This is of utmost importance for the soundness of our language as it preserves the distinction between the meta and the object levels. It is also justified, as explicit substitutions are intended to only remove names which are in the support of a term, and names appearing under **box** do not contribute to the support.

The operation of substitution application depends upon the operation of *substitution composition* $\Theta_1 \circ \Theta_2$, which we define next.

Definition 3 (Composition of substitutions)
Given two substitutions Θ_1 and Θ_2 with finite domains, their composition $\Theta_1 \circ \Theta_2$ is the substitution defined as

$$(\Theta_1 \circ \Theta_2)(X) \quad = \quad \{\Theta_1\}(\Theta_2(X))$$

The composition of two substitutions with finite domains is well-defined, as the resulting mapping from names to terms is finite. Indeed, for every name $X \notin \mathbf{dom}(\Theta_1) \cup \mathbf{dom}(\Theta_2)$, we have that $(\Theta_1 \circ \Theta_2)(X) = X$, and therefore $\mathbf{dom}(\Theta_1 \circ \Theta_2) \subseteq \mathbf{dom}(\Theta_1) \cup \mathbf{dom}(\Theta_2)$. Now, since this $\mathbf{dom}(\Theta_1 \circ \Theta_2)$ is finite, the syntactic representation of the composition can easily be computed as the set

$$\{X \to \{\Theta_1\}(\Theta_2\ (X)) \mid X \in \mathbf{dom}(\Theta_1) \cup \mathbf{dom}(\Theta_2)\}$$

It would occasionally be beneficial to represent this set as a disjoint union of two smaller sets Θ_1' and Θ_2' defined as:

$$\begin{aligned} \Theta_1' &= \{X \to \Theta_1\ (X) \mid X \in \mathbf{dom}(\Theta_1) \setminus \mathbf{dom}(\Theta_2)\} \\ \Theta_2' &= \{X \to \{\Theta_1\}(\Theta_2\ (X)) \mid X \in \mathbf{dom}(\Theta_2)\} \end{aligned}$$

It is important to notice that, though the definitions of substitution application and substitution composition are mutually recursive, both the operations are terminating. Substitution application is defined inductively over the structure of its argument, so the size of terms on which it operates is always decreasing. Composing substitutions with finite domain also terminates. Indeed, as the above equations show, only a finite amount of work is needed to compute the domain of a composition. After the domain is obtained, the syntactic representation of the composition is computed in finite time by substitution application.

3.3 Type system

The type system of our $\nu^{\Box}$-calculus consists of two mutually recursive judgments:

$$\Sigma; \Delta; \Gamma \vdash e : A\,[C]$$

and

$$\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$$

Both of them are hypothetical and work with three contexts: context of names Σ, context of expression variables Δ, and a context of value variables Γ (the syntactic structure of all three contexts is given in Section 3.1). The first judgment is the typing judgment for expressions. Given an expression e it checks whether e has type A, and is generated by the support set C. The second judgment types the explicit substitutions. Given a substitution Θ and two support sets C and D, the substitution has the type $[C] \Rightarrow [D]$ if it maps expressions of support C to expressions of support D. This intuition will be proved in Section 3.4.

The contexts deserve a few more words. Because the types of $\nu^{\Box}$-calculus depend on names, and types of names can depend on other names as well, we must impose some conditions on well-formedness of contexts. Henceforth, variable contexts Δ and Γ will be well-formed relative to Σ if Σ declares all the names that appear in the types of Δ and Γ. A name context Σ is well-formed if every type in Σ uses only names declared to the left of it. Further, we will often abuse the notation and write $\Sigma = \Sigma', X{:}A$ to define the *set* Σ' obtained after removing the name X from the context Σ. Obviously, Σ' does not have to be a well-formed context, as types in it may depend on X, but we will always transform Σ' into a well-formed context before

Explicit substitutions

$$\frac{C \subseteq D}{\Sigma; \Delta; \Gamma \vdash \langle\ \rangle : [C] \Rightarrow [D]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e : A\,[D] \qquad \Sigma; \Delta; \Gamma \vdash \langle \Theta \rangle : [C \setminus \{X\}] \Rightarrow [D] \qquad X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash \langle X \to e, \Theta \rangle : [C] \Rightarrow [D]}$$

Hypothesis

$$\frac{X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash X : A\,[X, C]} \qquad \frac{}{\Sigma; \Delta; (\Gamma, x{:}A) \vdash x : A\,[C]} \qquad \frac{\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta \rangle : [C] \Rightarrow [D]}{\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta \rangle u : A\,[D]}$$

λ-calculus

$$\frac{\Sigma; \Delta; (\Gamma, x{:}A) \vdash e : B\,[C]}{\Sigma; \Delta; \Gamma \vdash \lambda x{:}A.\ e : A \to B\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e_1 : A \to B\,[C] \qquad \Sigma; \Delta; \Gamma \vdash e_2 : A\,[C]}{\Sigma; \Delta; \Gamma \vdash e_1\ e_2 : B\,[C]}$$

Modality

$$\frac{\Sigma; \Delta; \cdot \vdash e : A\,[C]}{\Sigma; \Delta; \Gamma \vdash \mathbf{box}\ e : \Box_C A\,[D]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e_1 : \Box_D A\,[C] \qquad \Sigma; (\Delta, u{:}A[D]); \Gamma \vdash e_2 : B\,[C]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B\,[C]}$$

Names

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash e : B\,[C] \qquad X \notin \mathbf{fn}(B[C])}{\Sigma; \Delta; \Gamma \vdash \nu X{:}A.\ e : A \nrightarrow B\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e : A \nrightarrow B\,[C]}{\Sigma; \Delta; \Gamma \vdash \mathbf{choose}\ e : B\,[C]}$$

Figure 2: Typing rules of the $\nu^{\Box}$-calculus.

plugging it back into our judgments. Thus, we will always take care, and also implicitly assume, that all the contexts in the judgments are well-formed. The same holds for all the types and support sets that we use in the rules.

The typing rules of $\nu^{\Box}$ are presented in Figure 2. A pervasive characteristic of the type system is *support weakening*. Namely, if a term is in the set of expressions of type A freely generated by a support set C, then it certainly is among the expressions freely generated by some support set $D \supseteq C$. We make this property admissible to both the judgments of the type system, and it will be proved as a lemma in Section 3.4.

Explicit substitutions A substitution with empty syntactic representation is the identity substitution. When an identity substitution is applied to a term containing names from C, the resulting term obviously contains names from C. But the support of the resulting term can be extended by support weakening to a superset D, as discussed above, so we bake this property into the side condition $C \subseteq D$ for the identity substitution rule. We implicitly require that both the sets are well-formed, i.e. that they both contain only names already declared in the name context Σ.

The rule for non-empty substitutions is quite expected. It recursively checks each of its component terms for being well typed in the given contexts and support. It is worth noticing however, that a substitution

Θ can be given a type $[C] \Rightarrow [D]$ where the "domain" support set C is completely unrelated to the set $\mathbf{dom}(\Theta)$. In other words, the substitution can provide definitions for more names or for less names than the typing judgment actually cares for. For example, the substitution $\Theta = (X \to 10, Y \to 20)$ has domain $\mathbf{dom}(\Theta) = \{X, Y\}$, but it can be given (among others) the typings: $[\ ] \Rightarrow [\ ]$, $[X] \Rightarrow [\ ]$, as well as $[X, Y, Z] \Rightarrow [Z]$.

Hypothesis rules Since there are three kinds of variable contexts, we have three hypothesis rules. First is the rule for names. A name X can be used provided it has been declared in Σ and is accounted for in the supplied support set. The implicit assumption is that the support set C is well-formed, i.e. that $C \subseteq \mathbf{dom}(\Sigma)$ The rule for value variables is straightforward. The typing $x{:}A$ can be inferred, if $x{:}A$ is declared in Γ. The actual support of such a term can be any support set C as long as it is well-formed, which is implicitly assumed. Expression variables occur in a term always prefixed with an explicit substitution. The rule for expression variables has to check if the expression variable is declared in the context Δ and if its corresponding substitution has the appropriate type.

λ-calculus fragment The rule for λ-abstraction is quite standard. Its implicit assumption is that the argument type A is well-formed in name context Σ before it is introduced into the variable context Γ. Application rule checks both the function and the application argument against the same support set.

Modal fragment Just as in $\lambda^{\Box}$-calculus, the meaning of the rule for $\Box$-introduction is to ensure that the boxed expression e represents an abstract syntax tree. It checks e for having a given type in a context without compile code variables. The support that e has to match is supplied as an index to the $\Box$ constructor. On the other hand, the support for the whole expression **box** e is empty, as the expression obviously does not contain any names at the meta level. Thus, the support can be arbitrarily weakened to any well-formed support set D. The $\Box$-elimination rule is also a straightforward extension of the corresponding $\lambda^{\Box}$ rule. The only difference is that the bound expression variable u from the context Δ now has to be stored with its support annotation.

Names fragment The introduction form for names is $\nu X{:}A.\ e$ with its corresponding type $A \nrightarrow B$. It introduces an "irrelevant" name $X{:}A$ into the computation determined by e. It is assumed that the type A is well-formed relative to the context Σ. The term constructor **choose** is the elimination form for $A \nrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction. In other words, the operational semantics of the β-redex **choose** $(\nu X{:}A.\ e)$ (formalized in Section 3.5) proceeds with the evaluation of e in a run-time context in which a fresh name has been picked for X. It is justified to do so because X is bound by ν and, by convention, can be renamed with a fresh name. The irrelevancy of X in the above example means that X will never be encountered during the evaluation of e in a computationally significant position. Thus, (1) it is not necessary to specify its run-time behavior, and (2) it can never escape the scope of its introducing ν in any observable way. The the side-condition to ν-introduction serves exactly to enforce this irrelevancy. It effectively limits X to appear only in dead-code subterms of e or in subterms from which it will eventually be removed by some explicit substitution. For example, consider the following term

```
νX : int. νY : int.
   box (let box u = box X
            box v = box Y
         in
               ⟨X -> 1⟩ u
         end)
```

It contains a substituted occurrence of X and a dead-code occurrence of Y, and is therefore well-typed (of type $int \nrightarrow int \nrightarrow \Box int$).

One may wonder what is the use of entities like names which are supposed to appear only in computationally insignificant positions in the computation. The fact is, however, that names are not insignificant at all. Their import lies in their identity. For example, in Section 5 on intensional analysis of syntax, we will compare names for equality – something that cannot be done with ordinary variables. For, ordinary variables are just placeholders for some values; we cannot compare the variables for equality, but only the values that the variables stand for. In this sense we can say that λ-abstraction is parametric, while ν-abstraction is deliberately designed not to be.

It is only that names appear irrelevant because we have to force a certain discipline upon their usage. In particular, before leaving the local scope of some name X, as determined by its introducing ν, we have to "close up" the resulting expression if it depends significantly on X. This "closure" can be achieved by turning the expression into a λ-abstraction by means of explicit substitutions. Otherwise, the introduction of the new name will be an observable effect. To paraphrase, when leaving the scope of X, we have to turn the "polynomials" depending on X into functions. An illustration of this technique is the program already presented in Example 2.

The previous version of this work [Nan02] did not use the constructors ν and **choose**, but rather combined them into a single constructor **new**. This is also the case in the work of Pitts and Gabbay [PG00]. The decomposition is given by the equation

$$\textbf{new } X{:}A \textbf{ in } e \quad = \quad \textbf{choose } (\nu X{:}A.\ e)$$

We have decided on this reformulation in order to make the types of the language follow more closely the intended meaning of the terms and thus provide a stronger logical foundation for the calculus.

3.4 Structural properties

This section explores the basic theoretical properties of our type system. The lemmas developed here will be used to justify the operational semantics that we ascribe to $\nu^{\Box}$-calculus in Section 3.5, and will ultimately lead to the proof of the Type preservation and Progress theorems.

Lemma 4 (Structural properties of contexts)

1. **Exchange** *Let Σ', Δ' and Γ' be well-formed contexts obtained by permutation from Σ, Δ and Γ respectively. Then*
   (a) *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, *then* $\Sigma'; \Delta'; \Gamma' \vdash e : A\,[C]$
   (b) *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, *then* $\Sigma'; \Delta'; \Gamma' \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, *then*
2. **Weakening** *Let* $\Sigma \subseteq \Sigma'$, $\Delta \subseteq \Delta'$ *and* $\Gamma \subseteq \Gamma'$. *Then*
   (a) *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, *then* $\Sigma'; \Delta'; \Gamma' \vdash e : A\,[C]$
   (b) *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, *then* $\Sigma'; \Delta'; \Gamma' \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$
3. **Contraction on variables**
   (a) *if* $\Sigma; \Delta; (\Gamma, x{:}A, y{:}A) \vdash e : B\,[C]$, *then* $\Sigma; \Delta; (\Gamma, w{:}A) \vdash [w/x, w/y]e : B\,[C]$
   (b) *if* $\Sigma; \Delta; (\Gamma, x{:}A, y{:}A) \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, *then* $\Sigma; \Delta; (\Gamma, w{:}A) \vdash \langle [w/x, w/y]\Theta\rangle : [C] \Rightarrow [D]$
   (c) *if* $\Sigma; (\Delta, u{:}A[D], v{:}A[D]); \Gamma \vdash e : B\,[C]$, *then* $\Sigma; (\Delta, w{:}A[D]); \Gamma \vdash [w/u, w/v]e : B\,[C]$.
   (d) *if* $\Sigma; (\Delta, u{:}A[D], v{:}A[D]); \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C_2]$, *then* $\Sigma; (\Delta, w{:}A[D]); \Gamma \vdash \langle [w/u, w/v]\Theta\rangle : [C_1] \Rightarrow [C_2]$.

Proof: By straightforward induction on the structure of the typing derivations. ∎

Notice that contraction on names does not hold in $\nu^{\Box}$. Indeed identifying two different names in a term may make the term syntactically ill-formed. Typical examples are explicit substitutions which are in one-one correspondence with their syntactic representations. Identifying two names may make a syntactic representation assign two different images to a same name which would break the correspondence with substitutions.

The next series of lemmas establishes the admissibility of support weakening, as discussed in Section 3.3.

Lemma 5 (Support weakening)
Support weakening is covariant on the right-hand side and contravariant on the left-hand side of the judgments. More formally, let $C \subseteq C' \subseteq \mathbf{dom}(\Sigma)$ *and* $D' \subseteq D \subseteq \mathbf{dom}(\Sigma)$ *be well-formed support sets. Then the following holds:*

1. *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, *then* $\Sigma; \Delta; \Gamma \vdash e : A\,[C']$.

2. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C']$.

3. *if* $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash e : B\,[C]$, *then* $\Sigma; (\Delta, u{:}A[D']); \Gamma \vdash e : B\,[C]$

4. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D'] \Rightarrow [C]$.

Proof: The first two statements are proved by straightforward simultaneous induction on the given derivations. The third and the fourth part are proved by induction on the structure of their respective derivations. ∎

Lemma 6 (Support extension)
Let $D \subseteq \mathbf{dom}(\Sigma)$ be a well-formed support set. Then the following holds:

1. *if* $\Sigma; (\Delta, u{:}A[C_1]); \Gamma \vdash e : B\,[C_2]$ *then* $\Sigma; (\Delta, u{:}A[C_1 \cup D]); \Gamma \vdash e : B\,[C_2 \cup D]$

2. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C_2]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1 \cup D] \Rightarrow [C_2 \cup D]$

Proof: By induction on the structure of the derivations. ∎

Lemma 7 (Substitution merge)
If $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$ and $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_2] \Rightarrow [D]$ where $\mathbf{dom}(\Theta) \cap \mathbf{dom}(\Theta') = \emptyset$, then $\langle\Theta, \Theta'\rangle : [C_1 \cup C_2] \Rightarrow [D]$.

Proof: By induction on the structure of Θ'. ∎

The following lemma shows that the intuition behind the typing judgment for explicit substitution explained in Section 3.3 is indeed valid.

Lemma 8 (Explicit substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Then the following holds:

1. *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$ *then* $\Sigma; \Delta; \Gamma \vdash \{\Theta\}e : A\,[D]$.

2. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta \circ \Theta'\rangle : [C_1] \Rightarrow [D]$.

Proof: By simultaneous induction on the structure of the derivations. We just present the proof of the second statement.

Given the substitutions Θ and Θ', we split the representation of $\Theta \circ \Theta'$ into two disjoint sets:

$$\begin{array}{rcl} \Theta'_1 & = & \{X \to \Theta(X) \mid X \in \mathbf{dom}(\Theta) \setminus \mathbf{dom}(\Theta')\} \\ \Theta'_2 & = & \{X \to \{\Theta\}(\Theta'\,X) \mid X \in \mathbf{dom}(\Theta')\} \end{array}$$

and set out to show that

(a) $\Sigma; \Delta; \Gamma \vdash \langle\Theta'_1\rangle : [C_1 \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$, and

(b) $\Sigma; \Delta; \Gamma \vdash \langle\Theta'_2\rangle : [C_1 \cap \mathbf{dom}(\Theta')] \Rightarrow [D]$.

These two typings imply the result by the substitution merge lemma (Lemma 7). The statement (b) follows from the typing of Θ' by support weakening (Lemma 5.4), and the first part of the lemma. To establish (a), observe that from the typing of Θ it is clear that $\Theta'_1 : [C \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$. But, since $C_1 \setminus \mathbf{dom}(\Theta') \subseteq C \setminus \mathbf{dom}(\Theta')$ readily follows from the typing of Θ', the result is obtained by support weakening. ∎

The following lemma establishes the hypothetical nature of the two typing judgment with respect to the ordinary value variables.

Lemma 9 (Value substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash e_1 : A\,[C]$. The following holds:

1. *if* $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e_2 : B\,[C]$, *then* $\Sigma; \Delta; \Gamma \vdash [e_1/x]e_2 : B\,[C]$
2. *if* $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \langle\Theta\rangle : [C'] \Rightarrow [C]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle [e_1/x]\Theta\rangle : [C'] \Rightarrow [C]$

Proof: Simultaneous induction on the two derivations. ∎

The situation is not that easy with expression variables. A simple substitution of an expression for some expression variable will not result in a syntactically well-formed term. The reason is, as discussed before, that occurrences of expression variables are always prefixed by an explicit substitution to form a kind of closure. But, explicit substitutions in $\nu^{\Box}$-calculus can occur only as part of closures, and cannot be freely applied to arbitrary terms[1]. Hence, if a substitution of expression e for expression variable u is to produce a syntactically valid term, we need to follow it up with applications over e of *explicit name substitutions* that were paired up with u. This also gives us a control over not only the extensional, but also the intensional form of boxed expressions (which is necessary, as the later are supposed to represent abstract syntax trees). The definition below generalizes capture-avoiding substitution of expression variables in order to handle this problem.

Definition 10 (Substitution of expression variables)
The capture-avoiding substitution of e for an expression variable *u is defined recursively as follows*

$$
\begin{array}{lllll}
[\![e/u]\!] & \langle\Theta\rangle u & = & \{[\![e/u]\!]\Theta\}e & \\
[\![e/u]\!] & \langle\Theta\rangle v & = & \langle[\![e/u]\!]\Theta\rangle v & u \neq v \\
[\![e/u]\!] & x & = & x & \\
[\![e/u]\!] & X & = & X & \\
[\![e/u]\!] & \lambda x{:}A.\ e' & = & \lambda x{:}A.\ [\![e/u]\!]e' & x \notin \mathbf{fv}(e) \\
[\![e/u]\!] & e_1\ e_2 & = & [\![e/u]\!]e_1\ [\![e/u]\!]e_2 & \\
[\![e/u]\!] & \mathbf{box}\ e' & = & \mathbf{box}\ [\![e/u]\!]e' & \\
[\![e/u]\!] & \mathbf{let\ box}\ v = e_1\ \mathbf{in}\ e_2 & = & \mathbf{let\ box}\ v = [\![e/u]\!]e_1\ \mathbf{in}\ [\![e/u]\!]e_2 & u \notin \mathbf{fv}(e) \\
[\![e/u]\!] & \nu X{:}A.\ e' & = & \nu X{:}A.\ [\![e/u]\!]e' & X \notin \mathbf{fn}(e) \\
[\![e/u]\!] & \mathbf{choose}\ e' & = & \mathbf{choose}\ ([\![e/u]\!]e') & \\
 & & & & \\
[\![e/u]\!] & (\cdot) & = & (\cdot) & \\
[\![e/u]\!] & (X \to e', \Theta) & = & (X \to [\![e/u]\!]e', [\![e/u]\!]\Theta) &
\end{array}
$$

Notice that in the first clause $\langle\Theta\rangle u$ of the above definition the resulting expression is obtained by carrying out the explicit substitution.

Lemma 11 (Expression substitution principle)
Let e_1 be an expression without free value variables *such that $\Sigma; \Delta; \cdot \vdash e_1 : A\,[C]$. Then the following holds:*

1. *if* $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e_2 : B\,[D]$
2. *if* $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle\Theta\rangle : [D'] \Rightarrow [D]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle[\![e_1/u]\!]\Theta\rangle : [D'] \Rightarrow [D]$

Proof: By simultaneous induction on the two derivations. We just present one case from the proof of the first statement.

case $e_2 = \langle\Theta\rangle u$.

1. by derivation, $A = B$ and $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$
2. by second part of the lemma, $\Sigma; \Delta; \Gamma \vdash \langle[\![e_1/u]\!]\Theta\rangle : [C] \Rightarrow [D]$
3. by principle of explicit substitution (Lemma 8.1), $\Sigma; \Delta; \Gamma \vdash \{[\![e_1/u]\!]\Theta\}e_1 : B\,[D]$
4. but this is exactly equal to $[\![e_1/u]\!]e_2$

∎

[1] Albeit this extension does not seem particularly hard, we omit it for simplicity.

$$\frac{\Sigma, e_1 \longmapsto \Sigma', e_1'}{\Sigma, (e_1\ e_2) \longmapsto \Sigma', (e_1'\ e_2)} \qquad \frac{\Sigma, e_2 \longmapsto \Sigma', e_2'}{\Sigma, (v_1\ e_2) \longmapsto \Sigma', (v_1\ e_2')} \qquad \frac{}{\Sigma, (\lambda x{:}A.\ e)\ v \longmapsto \Sigma, [v/x]e}$$

$$\frac{\Sigma, e_1 \longmapsto \Sigma', e_1'}{\Sigma, (\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) \longmapsto \Sigma', (\mathbf{let\ box}\ u = e_1'\ \mathbf{in}\ e_2)}$$

$$\frac{}{\Sigma, (\mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2) \longmapsto \Sigma, [\![e_1/u]\!]e_2}$$

$$\frac{\Sigma, e \longmapsto \Sigma', e'}{\Sigma, \mathbf{choose}\ e \longmapsto \Sigma', \mathbf{choose}\ e'} \qquad \frac{}{\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) \longmapsto (\Sigma, X{:}A), e}$$

Figure 3: Structured operational semantics of $\nu^{\Box}$-calculus.

3.5 Operational semantics

We define the small-step call-by-value operational semantics of the $\nu^{\Box}$-calculus through the judgment

$$\Sigma, e \longmapsto \Sigma', e'$$

which relates an expression e with its one-step reduct e'. The relation is defined on expressions with no free variables. An expression can contain free names, but it must have *empty support*. In other words, we only consider for evaluation those terms whose names appear either on the object level, or in computationally irrelevant positions, or are removed by some explicit substitution. Because free names are allowed, the operational semantics has to account for them by keeping track of the run-time name contexts. The rules of the judgment are given in Figure 3, and the values of the language are generated by the grammar below.

$$Values\quad v \quad ::= \quad c \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e$$

The rules are standard, and the only important observation is that the β-redex for the type constructor $\not\twoheadrightarrow$ extends the run-time context with a fresh name before proceeding. This is needed for soundness purposes. The freshly introduced name will indeed not appear in computationally significant positions during the subsequent evaluations, but it may appear in "insignificant" positions (i.e. dead-code or under a substitution), so we keep the name and its typing in the run-time context. This will come handy in the formulation of the Type preservation and Progress theorems below. It will also play a role in intensional analysis of higher-order syntax in Section 5 whose operational semantics sometimes proscribes typechecking the expressions against which we pattern-match.

The evaluation relation is sound with respect to typing, and it never gets stuck, as the following theorems establish.

Theorem 12 (Type preservation)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* $\Sigma'; \cdot; \cdot \vdash e' : A\,[\,]$.

Proof: By a straightforward induction on the structure of e using the substitution principles. ■

Theorem 13 (Progress)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$, *then either*

1. *e is a value, or*

2. *there exist a term e' and a context Σ' extending Σ, such that $\Sigma, e \longmapsto \Sigma', e'$.*

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

The progress theorem seems to indicate that the evaluation relation is not deterministic, as each term may have more then one reduct. However, all these reducts are different only in the identity of the newly introduced irrelevant names, as the lemma below establishes.

Lemma 14 (Determinacy)
If $\Sigma; \cdot; \cdot \vdash e : A[\ ]$ and $\Sigma, e \longmapsto \Sigma_1, e_1$ and $\Sigma, e \longmapsto \Sigma_2, e_2$, then there exists a permutation of names $\pi : \mathcal{N} \to \mathcal{N}$, fixing $\mathbf{dom}(\Sigma)$, such that $\Sigma_1 = \pi(\Sigma_2)$ and $e_1 = \pi(e_2)$.

Proof: By induction on the structure of e. The only interesting case is when $e = \mathbf{choose}\ (\nu X{:}A.\ e')$. Then it must be $e_1 = [X_1/X]e'$, $e_2 = [X_2/X]e'$, and $\Sigma_1 = (\Sigma, X_1{:}A)$, $\Sigma_2 = (\Sigma, X_2{:}A)$, where $X_1, X_2 \in \mathcal{N}$ are fresh. Obviously, the involution $\pi = (X_1\ X_2)$ which swaps these two names has the required properties. ■

4 Support polymorphism

It is frequently necessary to write programs which are polymorphic in the support of their syntactic object-level arguments, because they are intended to manipulate abstract syntax trees whose support is not known at compile time. A typical example would be a function which recurses over some syntax tree with binding structure When it encounters a λ-abstraction, it has to place a fresh name instead of the bound variable, and recursively continue scanning the body of the λ-abstraction, which is itself a syntactic expression but depending on this newly introduced name.[2] For such uses, we extend the $\nu^\Box$-calculus with a notion of explicit support polymorphism in the style of Girard and Reynolds [Gir86, Rey83]. It turns out that the constructs explained here will also play a role in intensional analysis of higher-order syntax in Section 5 as a representation mechanism for encoding object-level functions as meta functions over object-level expressions.

The addition of support polymorphism to the simple $\nu^\Box$-calculus starts with syntactic changes that we summarize below.

Support variables	p, q	$\in$	$\mathcal{P}$
Support sets	C, D	$\in$	$(\mathcal{N} \cup \mathcal{P})^*$
Types	A	$::=$	$\ldots \mid \forall p.\ A$
Terms	e	$::=$	$\ldots \mid \Lambda p.\ e \mid e\ [C]$
Name context	Σ	$::=$	$\ldots \mid \Sigma, p$
Values	v	$::=$	$\ldots \mid \Lambda p.\ e$

We introduce a new syntactic category of *support variables*, which are intended to stand for unknown support sets. In addition, the support sets themselves are now allowed to contain these support variables, to express the situation in which only a portion of a support set is unknown. Consequently, the function $\mathbf{fn}(-)$ must be updated to now return the set of *names and support variables* appearing in its argument. The language of types is extended with the type $\forall p.\ A$ expressing universal support quantification. Its introduction form is $\Lambda p.\ e$, which abstracts an unknown support set p in the expression e. This Λ-abstraction will also be a value in the extended operational semantics. The corresponding elimination form is the application $e\ [C]$ whose meaning is to instantiate the unknown support set abstracted in e with the provided support set C. Because now the types can depend on names as well as on support variables, the name contexts must declare both. We assume the same convention on well-formedness of the name context as before.

The typing judgment has to be instrumented with new rules for typing support-polymorphic abstraction and application.

$$\frac{(\Sigma, p); \Delta; \Gamma \vdash e : A\,[C] \qquad p \notin C}{\Sigma; \Delta; \Gamma \vdash \Lambda p.\ e : \forall p.\ A\,[C]} \qquad\qquad \frac{\Sigma; \Delta; \Gamma \vdash e : \forall p.\ A\,[C]}{\Sigma; \Delta; \Gamma \vdash e\ [D] : ([D/p]A)\,[C]}$$

[2] The calculus described here cannot support this scenario yet because it lacks type polymorphism and type-polymorphic recursion, but support polymorphism is a necessary step in that direction.

The $\forall$-introduction rule requires that the bound variable p does not escape the scope of the constructors $\forall$ and Λ which bind it. In particular it must be $p \notin C$. The convention also assumes implicitly that $p \notin \Sigma$, before it can be added. The rule for $\forall$-elimination substitutes the argument support set D into the type A. It assumes that D is well-formed relative to the context Σ, i.e. that $D \subseteq \mathbf{dom}(\Sigma)$. The operational semantics for the new constructs is also not surprising.

$$\frac{\Sigma, e \longmapsto \Sigma', e'}{\Sigma, (e\ [C]) \longmapsto \Sigma', (e'\ [C])} \qquad \frac{}{\Sigma, (\Lambda p.\ e)\ [C] \longmapsto \Sigma, [C/p]e}$$

The extended language satisfies the following substitution principle.

Lemma 15 (Support substitution principle)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathbf{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Then the following holds.

1. *if $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, then $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e' : A'\,[C']$*
2. *if $\Sigma; \Delta; \Gamma \vdash \Theta : C_1 \to C_2$, then $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta' : C_1' \to C_2'$*

Proof: By simultaneous induction on the two derivations. ∎

The structural properties presented in Section 3.4 readily extend to the new language with support polymorphism. We present the extended versions, as well as their proofs, in the Appendix. The same is true of the Type preservation and Progress theorems (Theorem 12 and 13) whose additional cases involving support abstraction and application are handled using the above Lemma 15.

Example 3 In a support-polymorphic $\nu^{\Box}$-calculus we can slightly generalize the program from Example 2 by pulling out the helper function `exp'` and parametrizing it over the exponentiating expression. In the concrete syntax below we symbolize by `[p]` that `p` is a support variable abstracted in the function definition.

```
fun exp' [p] (e : □_p int) (n : int) : □_p int =
    if n = 0 then box 1
    else
       let box u = exp' [p] e (n - 1)
           box w = e
       in
           box (u * w)
       end

fun exp (n : int) : □(int -> int) =
    choose (νX : int.
       let box w = exp' [X] (box X) n
       in
           box (λx:int. ⟨X -> x⟩ w)
       end)

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

∎

In the development of the $\nu^{\Box}$-calculus we have had a particular semantic interpretation in mind of object level expressions as abstract syntax trees representing templates for source programs. This interpretation will be exploited in an essential way in Section 5. Notice, however, that the fragment presented thus far is not necessarily committed to viewing the object expressions as syntax. It is quite possible (and it remains an important future work) that boxed expressions of core $\nu^{\Box}$ with support polymorphism can be stored in run-time in some intermediate or even compiled form, which might be beneficial to the efficiency of the calculus.

5 Intensional analysis of higher-order syntax

5.1 Syntax and typechecking

As explained in Section 3, we interpret the type $\Box_C A$ as a set of syntactic expressions of type A freely generated over the set of "indeterminates" C. The calculus presented so far was capable of constructing values of type $\Box_C A$, but it is obviously desirable to provide capabilities for inspecting and destructing these syntax trees. Here we extend the support-polymorphic $\nu^\Box$-calculus with primitives for pattern-matching against syntactic expressions with binding structure. For reasons of simplicity, we develop the extension in the setup of our calculus, where the languages used in the meta and the object levels are the same. But this is not necessary, as the same mechanism would work for any object level calculus with binding structure. As a matter of fact, it is probably more appropriate to emphasize the distinction between meta and object calculi, because even the pattern-matching presented here can recognize only a *subset* of term constructors of $\nu^\Box$. In particular, we can only test if an expression is a name, or a λ-abstraction or an application. It is not clear whether the expressiveness of pattern-matching can be extended to handle a larger subset of the object-language without significant additions to the meta-language. But this would in turn require extensions to pattern-match against the additions, which would in turn require new extensions to the meta-language, and so on.

The syntactic additions that we consider in this section are summarized in the the table below.

Pattern variables	w	$\in$	$\mathcal{W}$
Higher-order patterns	π	$::=$	$(w\ x_1 \ldots x_n){:}A[C] \mid X \mid x \mid \lambda x{:}A.\ \pi \mid \pi_1\ \pi_2$
Pattern assignments	σ	$::=$	$\cdot \mid w \rightarrow e, \sigma$
Terms	e	$::=$	$\ldots \mid \textbf{case}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2$

We use higher-order patterns [Mil90] to match against syntactic expressions with binding structure. Higher-order patterns introduce two unrelated notions of variables that we must distinguish between. First is the concept of *free variables.* These are introduced by patterns for binding structure $\lambda x{:}A.\ \pi$ and are syntactic entities that can match only themselves. Second is the concept of *pattern variables.* They are placeholders intended to bind syntactic subexpressions in the process of matching and pass them to the subsequent computation. We use x, y and variants to range over bound variables, and w and variants to range over pattern variables.

The basic pattern $(w\ x_1 \ldots x_n){:}A[C]$ declares a pattern variable w which will be allowed to match a syntactic expression of type A and support C subject to the condition that the expression's free variables are among $x_1, \ldots, x_n$. Pattern X matches a name X from the global name context. Pattern $\lambda x{:}A.\ \pi$ matches a λ-abstraction of domain type A. It declares a new *free variable* x which is local to the pattern, and demands that the body of the matched expression conforms to the pattern π. A free variable x matches only the pattern x. Pattern $\pi_1\ \pi_2$ matches a syntactic expression representing application. Notice that the decision to explicitly assign types to every pattern variable forces the pattern for application to be monomorphic. In other words, the application pattern cannot match a pair of expressions representing a function and its argument if the domain type of the function is now known in advance. It is an important future work to extend intensional analysis to allow patterns which are type-polymorphic in this sense. The patterns are assumed to be linear, i.e. no pattern variable occurs twice.

The typing judgment for patterns has the form

$$\Sigma; \Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$$

It is hypothetical in the global context of names Σ, and the context of *locally declared* free variables Γ. It checks whether the pattern π has type A and support C and if the pattern variables from π conform to the typings given in the residual context Γ_1. The typing rules are presented in Figure 4. Most of them are straightforward and we do not explain them, but the rule for pattern variables deserves special attention. As it shows, in order for the pattern expression $(w\ x_1 \ldots x_n){:}A[C]$ to be well-typed, the free variables $x_1{:}A_1, \ldots, x_n{:}A_n$ have to be declared in the local context Γ and the support set $D \subseteq C$. Then w will match only expressions of type A with the given free variables and the names declared in D. *The residual context types w as a meta-level function over types $\Box_p A_i$ with polymorphic support.* This hints at the operational

$$\frac{D \subseteq C \qquad p \notin \mathbf{dom}(\Sigma)}{\Sigma; (\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n) \vdash ((w\ x_1 \ldots x_n){:}A[D]) : A\,[C] \Longrightarrow w{:}\forall p.\ \Box_p A_1 \to \cdots \to \Box_p A_n \to \Box_{p,D} A}$$

$$\frac{X{:}A \in \Sigma}{\Sigma; \Gamma \vdash X : A\,[X, C] \Longrightarrow \cdot} \qquad \frac{}{\Sigma; (\Gamma, x{:}A) \vdash x : A\,[C] \Longrightarrow \cdot}$$

$$\frac{\Sigma; (\Gamma, x{:}A) \vdash \pi : B\,[C] \Longrightarrow \Gamma_1}{\Sigma; \Gamma \vdash \lambda x{:}A.\ \pi : A \to B\,[C] \Longrightarrow \Gamma_1}$$

$$\frac{\Sigma; \Gamma \vdash \pi_1 : A \to B\,[C] \Longrightarrow \Gamma_1 \qquad \Sigma; \Gamma \vdash \pi_2 : A\,[C] \Longrightarrow \Gamma_2 \qquad \mathbf{fn}(A) \subseteq \mathbf{dom}(\Sigma)}{\Sigma; \Gamma \vdash \pi_1\ \pi_2 : B\,[C] \Longrightarrow (\Gamma_1, \Gamma_2)}$$

Figure 4: Typing rules for patterns.

semantics that will be assigned to higher-order patterns. If an expression e with a local free variable $x{:}A$ matches to a pattern variable w, then w will residualize to a meta-level function whose meaning is as follows: it takes a syntactic expression $e'{:}A$ and returns back the syntactic expression $[e'/x]e$.

In order to incorporate pattern matching into $\nu^{\Box}$, the syntax is extended with a new term constructor **case** e **of box** $\pi \Rightarrow e_1$ **else** e_2. The intended operational interpretation of **case** is to evaluate the argument e to obtain a boxed expression **box** e', then match e' 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.

Example 4 Consider the (rather restricted) function `reduce` that takes a syntactic expression of type A, and checks if it is a β-redex $(\lambda x{:}A.\ w_1)\ (w_2)$. If the answer is yes, it applies the "call-by-value" strategy: it reduces w_2, substitutes the reduct for x in w_1 and then continue reducing thus obtained expression. If the answer is no, it simply returns the argument.

```
fun reduce (e : □A) : □A =
    case e of
         box ((λx:A. ((w1 x):A[])) (w2:A[])) =>
                (* w1 : ∀q. □_qA -> □_qA *)
                (* w2 : ∀q. □_qA *)
             let val e2 = reduce (w2 [])
             in
                  reduce (w1 [] e2)
             end
         else e
```

Ideally, one would want to `reduce` an arbitrary expression, not just simple top-level redexes. We cannot currently write such a function mainly because our language lacks type-polymorphic patterns and type-polymorphic recursion. In particular, if the syntactic argument we are dealing with is an application of a general term of type $A \to A$ rather than a λ-abstraction, we cannot recursively reduce that term first unless the language is equipped with type-polymorphic recursion.

Nevertheless, `reduce` is illustrative of the way higher-order patterns work. Patterns transform an expression with a free variable into a function on syntax that substitutes the free variable with a given argument. *That way we can employ meta-level reduction to perform object-level substitution.* This is reminiscent of the idea of normalization-by-evaluation [BS91, BES98] and type-directed partial evaluation [Dan96]. ■

The typing rule for **case** is:

$$\frac{\Sigma;\Delta;\Gamma \vdash e : \Box_D A\,[C] \quad \Sigma;\cdot \vdash \pi : A\,[D] \Longrightarrow \Gamma_1 \quad \Sigma;\Delta;(\Gamma,\Gamma_1) \vdash e_1 : B\,[C] \quad \Sigma;\Delta;\Gamma \vdash e_2 : B\,[C]}{\Sigma;\Delta;\Gamma \vdash \textbf{case}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2 : B\,[C]}$$

Observe that the second premise of the rule requires an empty variable context, so that patterns cannot contain outside value or expression variables. However (and this is important), they can contain names. It is easy to incorporate the new syntax into the language. We first extend explicit substitution over the new **case** construct

$$\{\Theta\} \quad (\textbf{case}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2) \quad = \quad \textbf{case}\ (\{\Theta\}e)\ \textbf{of box}\ \pi \Rightarrow (\{\Theta\}e_1)\ \textbf{else}\ (\{\Theta\}e_2)$$

and similarly for expression substitution, and then all the structural properties derived in Section 3.4 easily hold. The only complication comes in handling names and support substitution because patterns are allowed to depend on names and support variables from the global context Σ. However, the lemmas below establish the required invariants.

Lemma 16 (Structural properties of pattern matching)

1. **Exchange** *Let Σ', Γ' and Γ'_1 be well-formed contexts obtained by permutation from Σ, Γ and Γ_1 respectively and $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $\Sigma';\Gamma' \vdash \pi : A\,[C] \Longrightarrow \Gamma'_1$*

2. **Weakening** *Let $\Sigma \subseteq \Sigma'$ and $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $\Sigma';\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$*

Proof: By straightforward introduction on the structure of the typing derivations. ■

Lemma 17 (Support substitution principle for pattern matching)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathbf{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Assume also that $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $(\Sigma_1, \Sigma'_2);\Gamma' \vdash \pi' : A'\,[C'] \Longrightarrow \Gamma'_1$.

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

5.2 Operational semantics

Operational semantics for pattern matching is given through the new judgment

$$\Sigma;\Gamma \vdash e \rhd \pi \Longrightarrow \sigma$$

which reads: in a global context of names and support variables Σ and a context of locally declared free variables Γ the matching of the expression e to the pattern π generates an assignment of values σ to the pattern variables of π. The rules for this judgment are given in Figure 5. Most of the rules are self-evident, but the rule for pattern variables deserves more attention. Its premise requires a run-time typecheck of the expression e in the given contexts. That is why the operational semantics of $\nu^{\Box}$-calculus (see Section 3.5) must carry around at run-time the list of currently defined names *and their typings*. The following lemma relates the typing judgment for patterns and their operational semantics.

Lemma 18 (Soundness of pattern matching)
Let π be a pattern such that $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$, where $\Gamma_1 = (w_1{:}A_1, \ldots, w_n{:}A_n)$. Furthermore, let e be an expression matching π to produce a pattern assignment σ, i.e. $\Sigma;\Gamma \vdash e \rhd \pi : A \Longrightarrow \sigma$. Then $\sigma = (w_1 \to e_1, \ldots, w_n \to e_n)$ where $\Sigma;\cdot;\cdot \vdash e_i : A_1$, for every $i = 1, \ldots, n$.

Notice that in the lemma we did not require that e be well-typed, or even syntactically well-formed. If it were not well-formed, the matching simply would not succeed.

Proof: By induction on the structure of π. We present the base case below.

$$\frac{\Sigma; \cdot; (x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e : A\,[D]}{\Sigma; (\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e \rhd ((w\ x_1 \ldots x_n){:}A[D]) : A \Longrightarrow [w \to \Lambda p.\ \lambda y_i{:}\Box_p A_i.\ \textbf{let box}\ x_i = y_i\ \textbf{in box}\ e]}$$

$$\frac{}{(\Sigma, X{:}A); \Gamma \vdash X \rhd X : A \Longrightarrow \cdot} \qquad \frac{}{\Sigma; (\Gamma, x{:}A) \vdash x \rhd x : A \Longrightarrow \cdot}$$

$$\frac{\Sigma; (\Gamma, x{:}A) \vdash e \rhd \pi : B \Longrightarrow \sigma}{\Sigma; \Gamma \vdash \lambda x{:}A.\ e \rhd \lambda x{:}A.\ \pi : (A \to B) \Longrightarrow \sigma} \qquad \frac{\Sigma; \Gamma \vdash e_1 \rhd \pi_1 : A \to B \Longrightarrow \sigma_1 \qquad \Sigma; \Gamma \vdash e_2 \rhd \pi_2 : A \Longrightarrow \sigma_2}{\Sigma; \Gamma \vdash e_1\ e_2 \rhd \pi_1\ \pi_2 : B \Longrightarrow (\sigma_1, \sigma_2)}$$

Figure 5: Operational semantics for pattern matching.

case $\pi = (w\ x_1 \ldots x_n){:}A[D]$, where $\Gamma = \Gamma_2, x_i{:}A_i$.

1. let $e' = (\Lambda p.\ \lambda y_i{:}\Box_p A_i.\ \textbf{let box}\ x_i = y_i\ \textbf{in box}\ e)$ and $A' = \forall p.\ \Box_p A_1 \to \cdots \to \Box_p A_n \to \Box_{p,D} A$
2. by typing derivation, $D \subseteq C$ and $x_i{:}A_i \in \Gamma$ and also $\Gamma_1 = (w{:}A')$
3. by matching derivation, $\Sigma; \cdot; (x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e : A\,[D]$, and $\sigma = (w \to e')$
4. by straightforward structural induction, $\Sigma; (x_1{:}A_1, \ldots, x_n{:}A_n); \cdot \vdash e : A\,[D]$
5. by support weakening, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash e : A\,[D, p]$
6. and thus also, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash \textbf{box}\ e : \Box_{D,p} A\,[\,]$
7. and $(\Sigma, p); \cdot; (y_1{:}\Box_p A_1, \ldots, y_n{:}\Box_p A_n) \vdash \textbf{let box}\ x_i = y_i\ \textbf{in box}\ e : \Box_{D,p} A\,[\,]$
8. and finally, $\Sigma; \cdot; \cdot \vdash e' : A'\,[\,]$

■

The last piece to be added is the operational semantics for the **case** statement, and the required rules are given below. Notice that the premise of last rule makes use of the fact that the operational semantics for patterns is deterministic; the rule applies if the expression and e and the pattern π cannot be matched.

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

$$\frac{\Sigma; \cdot \vdash e \rhd \pi : A \Longrightarrow (w_1 \to e_1', \ldots, w_n \to e_n')}{\Sigma, (\textbf{case box}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2) \longmapsto \Sigma, [e_1'/w_1, \ldots, e_n'/w_n]e_1}$$

$$\frac{\Sigma; \cdot \vdash e \rhd \pi \not\Longrightarrow \sigma \qquad \text{for any } \sigma}{\Sigma, (\textbf{case box}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2) \longmapsto \Sigma, e_2}$$

Finally, using the lemmas established in this section, we can augment the proof of the Type preservation and Progress theorems (Theorem 12 and 13) to cover the extended language. The complete proof is presented in the Appendix.

Example 5 The following examples present 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 source code for f, and an integer n, and returns the source code for $\lambda x.\ (fx)^n$. The idea is to have the resulting source code be as optimized as possible, while still computing the extensionally same result. We rely on programs presented in Section 2 and Examples 2 and 3.

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

```
fun fexp1 (f : □(int->int)) (n : int) : □(int->int) =
    let box g = f
        box p = exp3 n
    in
        box (λv:int. (p (g v)))
    end

- fexp1 (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 redexes. As could be expected, the $\nu^{\Box}$-calculus [3] provides a better way to stage the code, simply by using the function `exp` from Example 2 instead `exp3` from Section 2.

```
fun fexp2 (f : □(int->int)) (n : int) : □(int->int) =
    let box g = f
        box p = exp n
    in
        box (λv:int. p (g v))
    end

-fexp2 (box λw:int. w + 1) 2;
val it = box (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: we can eliminate the outer β-redex from the residual code, at the price of duplicating the inner one.

```
fun fexp3 (f : □(int->int)) (n : int) : □(int->int) =
    choose (νX:int.
      let box g = f
          box e = exp' [X] (box (g X)) n
      in
          box (λv:int. ⟨X -> v⟩e)
      end)

- fexp3 (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 redexes. The reason is that we do not utilize the *intensional* information that the passed argument is actually a boxed λ-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 λ-abstraction. This way we can obtain the most simplified, if not the most efficient residual code.

```
fun fexp (f : □(int->int)) (n : int) : □(int->int) =
    case f of
       box (λx:int. (w x:int[])) =>
           (* w : ∀q. □_q int -> □_q int *)
          choose (νX : int.
              let box F = exp' [X] (w [X] (box X)) n
              in
                  box (λv:int. ⟨X->v⟩F)
              end)
          else fexp2 f n
```

[3] And for that matter, $\lambda^{\bigcirc}$ and MetaML, as well.

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

■

Example 6 This example is a (segment of the) meta function for symbolic differentiation with respect to a distinguished indeterminate X.

```
fun diff (e : □_X real) : □_X real =
    case e of
     box X => box 1

    | box ((w1:real[X]) + (w2:real[X])) =>
           let box e1 = diff (w1 [])
               box e2 = diff (w2 [])
           in
               box (e1 + e2)
           end

    | box ((λx:real. ((FX x):real[X])) (GX:real[X])) =>
           (* FX : ∀q. □_q real -> □_{q,X} real *)
           (* GX : ∀q. □_{q,X} real *)
           (* check if FX really depends on X *)
           choose (νY : real.
                case (FX [Y] (box Y)) of
                  box (F:real[Y]) =>
                        (* FX is independent of X; apply the chain rule *)
                        let box f = F []
                            box f' = diff (box ⟨Y->X⟩f)
                            box gx = GX []
                            box gx' = diff (GX [])
                        in
                            box (⟨X->gx⟩f' * gx')
                        end
                  else  diff (FX [X] (GX [])))
     else (box 0) (* the argument is a constant *)
```

The most interesting part of `diff` is its treatment of application. The same limitations encountered in Example 4 apply here too, in the sense that we can pattern match only when the applying function is actually a λ-abstraction. Although it is wrong, we currently let all the other cases pass through the default case. Nevertheless, the example is still illustrative. After splitting the application into the function part f and the argument part g we test if f is independent of X. If that indeed is the case, it means that our application was actually a composition of functions $f\ (g\ X)$, and thus we can apply the chain rule to compute the derivative as $f'\ (g\ X) * (g'\ X)$. Otherwise, if f contains occurrences of X, the chain rule is inapplicable, so we only reduce the β-redex and differentiate the result. ■

6 Related work

The work presented in this paper lies in the intersection of several related areas: meta-programming, modal logic, run-time code generation and higher-order abstract syntax. The direct motivation and foundation for our type system is provided by the $\lambda^\Box$ and $\lambda^\bigcirc$ calculi. The $\lambda^\Box$-calculus evolved as a type theoretic explanation of run-time code-generation [LL96, WLP98] and logical analysis of staged computation [DP01, WLPD98], and we explained it in Section 2.

The calculus $\lambda^{\bigcirc}$, formulated by Davies in [Dav96], is the first attempt at handling object expressions with free variables. It is the proof-term calculus for discrete temporal logic, and it provides a notion of open object expression where the free variables of the object expression are represented by meta variables on a subsequent temporal level. The original motivation of $\lambda^{\bigcirc}$ was to develop a type system for binding-time analysis in the setup of partial evaluation, but it was quickly adopted for meta-programming through the development of MetaML [MTBS99, Tah99, Tah00].

MetaML adopts the "open code" type constructor of $\lambda^{\bigcirc}$ and generalizes the language with several features. The most important one is the addition of a type refinement for "closed code". Values classified by these "closed code" types are those "open code" expressions which happen to not depend on any free meta variables. It might be of interest here to point out a certain relationship between our concept of names and the phenomenon which occurs in the extension of MetaML with references [CMT00, CMS01]. A reference in MetaML must not be assigned an open code expression. Indeed, in such a case an eventual free variable from the expression may escape the scope of the λ-binder that 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 that it is allowed to contain in dead-code positions. These dead-code annotations are not a type constructor in MetaML, and the dead-code variables belong to the same syntactic category as ordinary variables, but they nevertheless very much compare to our names and ν-abstraction. *Thus, it seems that names are important for meta-programming, even if one is not interested in intensional code analysis.*

Another interesting calculus for meta-programming is Nielsen's $\lambda^{[]}$ described in [Nie01]. It is based on the same idea as our $\nu^{\Box}$-calculus – instead of defining the notion of closed code as a refinement of open code of $\lambda^{\bigcirc}$ or MetaML, it relaxes the notion of closed code of $\lambda^{\Box}$. Where we use names to stand for free variables of object expression, $\lambda^{[]}$ uses variables introduced by **box** (which thus becomes a binding construct). Variables bound by **box** have the same treatment as λ-bound variables. The type-constructor $\Box$ is updated to reflect the *types* (but not the names) of variables that its corresponding **box** binds, and thus it becomes questionable whether it can be used for intensional analysis of higher-order syntax. The language also lacks a concept corresponding to our support polymorphism which is one of the important ingredients for intensional analysis.

Nielsen and Taha in [NT03] present another system for combining the notions of closed and open code. It is based on $\lambda^{\Box}$ but it can explicitly name the object stages of computation through the notion of *environment classifiers*. Because the stages are explicitly named, each stage can be revisited multiple times and extended with new bound variables. This provides a functionality of open code. In many respects, the environment classifiers behave like universally quantified bound variables. In fact, it seems that environment classifiers and our support polymorphism are formalizing the same phenomenon in different base calculi.

Coming from the direction of higher-order abstract syntax [PE88], probably the first work pointing to the importance of a "non-parametric" binder like our ν-abstraction is Miller's [Mil90]. The connection of higher-order abstract syntax to modal logic has been recognized by Despeyroux, Pfenning and Schürmann in the system presented in [DPS97], which was later simplified into a two-level system in Schürmann's dissertation [Sch00]. There are also the works of Hofmann [Hof99], which discusses various presheaf models for higher-order abstract syntax, then Fiore, Plotkin and Turi's [FPT99] which explores untyped abstract syntax in a categorical setup, and an extension to arbitrary types by Fiore [Fio02].

However, the work that explicitly motivated our developments is the series of papers on Nominal Logic and FreshML by Pitts and Gabbay [GP02, PG00, Pit01, Gab00]. The names of Nominal Logic are introduced as the urelements of Fraenkel-Mostowsky set theory. FreshML is a language for manipulation of object syntax with binding structure based on this model. Its primitive notion is that of swapping of two names which is then used to define the operations of name abstraction (producing an α-equivalence class with respect to the abstracted name) and name concretion (providing a specific representative of an α-equivalence class). The earlier version of our paper [Nan02] contained these two operations, which were *almost* orthogonal to add. Name abstraction was used to encode abstract syntax trees which depend on a name whose identity is not known. Typically, the need for this would appear during pattern-matching of λ-binders when a new name must be introduced to stand for the bound variable. Since this name is out of the scope of the subsequent computation branch, it has to be abstracted. In the current version, that role is given to λ-abstraction and support polymorphism as illustrated in Example 4.

Unlike our calculus, FreshML does not keep track of a support of a term, but rather its complement. FreshML introduces names in a computation by a construct **new** X **in** e, which can roughly be interpreted in $\nu^{\Box}$-calculus as

$$\textbf{new}\ X\ \textbf{in}\ e \quad = \quad \textbf{choose}\ (\nu X.\ e)$$

Except in dead-code position, the name X can appear in e in a scope of an abstraction which hides X. One of the main differences between FreshML and $\nu^{\Box}$ is that names in FreshML are run-time values – it is possible in FreshML to evaluate a term with a non-empty support. On the other hand, while our names can have arbitrary types, FreshML names must be of a single type `atm` (though this can be generalized to an arbitrary family of types disjoint from the types of the other values of the language). Our calculus allows the general typing for names thanks to the modal distinction of meta and object levels. For example, without the modality, but with names of arbitrary types, a function defined on integers will always have to perform run-time checks to test if its argument is a valid integer (in which case the function is applied), or if its argument is a name (in which case the evaluation is suspended, and the whole expression becomes a syntactic entity). An added bonus is that $\nu^{\Box}$ can support an explicit name substitution as primitive, while substitution must be user-defined in FreshML.

On the logic side, the direct motivation for this paper comes from Pfenning and Davies' [PD01] which presents a natural deduction formulation for propositional S4. But in general, the interaction between modalities, syntax and names has been of interest to logicians for quite some time. For example, logics that can encode their own syntax are the topic of Gödel's Incompleteness theorems, and some references in that direction are Montague's [Mon63] and Smoryński's [Smo85]. Attardi's viewpoints [AS95] and McCarthy's contexts [McC93] are similar to our notion of support, and are used to express relativized truth. Finally, the names from $\nu^{\Box}$ resemble virtual individuals of Scott [Sco70] and also [Sco79], non-rigid designators of Fitting and Mendelsohn [FM99], and names of Kripke [Kri80]. All this classical work seems to indicate that meta-programming and higher-order syntax are just but a concrete instance of a much broader abstract phenomenon. We hope to draw on the cited work for future developments.

7 Conclusions and future work

This paper presents the $\nu^{\Box}$-calculus, which is a typed functional language for meta-programming, employing a novel way to define a modal type of syntactic object programs with free variables. The system combines the $\lambda^{\Box}$-calculus [PD01] with the notion of names inspired by developments in FreshML and Nominal Logic [PG00, GP02, Pit01, Gab00]. The motivation for combining $\lambda^{\Box}$ with names comes from the long-recognized need of meta-programming to handle object programs with free variables [Dav96, Tah99, MTBS99]. In our setup, the $\lambda^{\Box}$-calculus provides a way to encode closed syntactic code expressions, and names serve to stand for the eventual free variables. Taken together, they give us a way to encode open syntactic program expressions, and also compose, evaluate, inspect and destruct them. Names can be operationally thought of as locations which are tracked by the type system, so that names cannot escape the scope of their introduction form. The set of names appearing in the meta level of a term is called *support* of a term. Support of a term is reflected in the typing of a term, and a term can be evaluated only if its support is empty.

We also considered constructs for support polymorphism and for intensional analysis of higher-order syntax. The later is a pattern-matching mechanism to compare, inspect and destruct object expressions at run-time. We hope this feature can find its use in programming code optimizations in a setup of scientific and symbolic computation.

The $\nu^{\Box}$-calculus presented here supersedes the language considered in the previous version of this paper [Nan02]. Some of the changes we have adopted involve simplification of the operational semantics and the constructs for handling names. Furthermore, we decomposed the name introduction form `new` into two new constructors ν and **choose** which are now introduction and elimination form for a new type constructor $A \nrightarrow B$. This gives a stronger logical foundation to the calculus, as now the level of types follows much more closely the behavior of the terms of the language. We hope to further investigate these logical properties. Some immediate future work in this direction would include the embedding of discrete-time temporal logic and monotone discrete temporal logic into the logic of types of $\nu^{\Box}$, and also considering the proof-irrelevancy modality of [Pfe01] and [AB01] to classify terms of unknown support.

8 Acknowledgment

First and foremost, I would like to thank my advisor Frank Pfenning for all his invaluable contributions to this work, his strategic guidance and, above all, his tireless support. This document would have never seen the light without him. I am also indebted to Dana Scott, Bob Harper, Peter Lee and Andrew Pitts for their helpful comments on the earlier version of the paper.

References

[AB01] Steve Awodey and Andrej Bauer. Propositions as [Types]. Technical Report IML-R–34-00/01–SE, Institut Mittag-Leffler, The Royal Swedish Academy of Sciences, 2001.

[AS95] Giuseppe Attardi and Maria Simi. A formalization of viewpoints. *Fundamenta Informaticae*, 23(3):149–173, 1995.

[BES98] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 1998.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *LICS'91: Logic in Computer Science*, pages 203–211, Amsterdam, 1991.

[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.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *POPL'96: Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, 1996.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In *LICS'96: Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, 1996.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 1997.

[Fio02] Marcelo Fiore. Semantic analysis of normalization by evaluation for typed lambda calculus. In *PPDP'02: Principles and Practice of Declarative Programming*, pages 26–37, Pittsburgh, Pennsylvania, 2002.

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

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *LICS'99: Logic in Computer Science*, pages 193–202, Trento, Italy, 1999.

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

[Gir86] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[Gri89] Andreas Griewank. On automatic differentiation. In Masao Iri and Kunio Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer, 1989.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS'99: Logic in Computer Science*, pages 204–213, Trento, Italy, 1999.

[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 *PLDI'96: Programming Language Design and Implementation*, pages 137–148, 1996.

[McC93] John McCarthy. Notes on formalizing context. In *IJCAI'93: International Joint Conference on Artificial Intelligence*, pages 555–560, Chambery, France, 1993.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures. In *Proceedings of the First Esprit BRA Workshop on Logical Frameworks*, pages 323–335, Antibes, France, 1990.

[Mon63] Richard Montague. Syntactical treatment of modalities, with corollaries on reflexion principles and finite axiomatizability. *Acta Philosophica Fennica*, 16:153–167, 1963.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *ESOP'99: European Symposium on Programming*, pages 193–207, Amsterdam, 1999.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP'02: International Conference on Functional Programming*, pages 206–217, Pittsburgh, Pennsylvania, 2002.

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

[NT03] Michael Florentin Nielsen and Walid Taha. Environment classifiers. In *POPL'03: Principles of Programming Languages*, New Orleans, Louisiana, 2003. To appear.

[Ode94] Martin Odersky. A functional theory of local names. In *POPL'94: Principles of Programming Languages*, pages 48–59, Portland, Oregon, 1994.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI'88: Programming Language Design and Implementation*, pages 199–208, Atlanta, Georgia, 1988.

[Pfe01] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS'01: Logic in Computer Science*, pages 221–230, Boston, Massachusetts, 2001.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer, 2000.

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

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier, 1983.

[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 *POPL'02: Principles of Programming Languages*, pages 154–165, Portland, Oregon, 2002.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie Mellon University, 2000.

[Sco70] Dana Scott. Advice on modal logic. In Karel Lambert, editor, *Philosophical Problems in Logic*, pages 143–173. Dordrecht: Reidel, 1970.

[Sco79] Dana Scott. Identity and existence in intuitionistic logic. In Michael Fourman, Chris Mulvey, and Dana Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979.

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

[Smo85] C. Smoryński. *Self-Reference and Modal Logic.* Springer, 1985.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival. In *PEPM'00: Partial Evaluation and Semantics-Based Program Manupulation*, pages 34–43, Boston, Massachusetts, 2000.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *PLDI'98: Programming Language Design and Implementation*, pages 224–235, Montreal, Canada, 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

A.1 Structural properties

Lemma 5 (Support weakening)
Let $C \subseteq C' \subseteq \mathbf{dom}(\Sigma)$ and $D' \subseteq D \subseteq \mathbf{dom}(\Sigma)$ be well-formed support sets. Then the following holds:

1. *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$*, then* $\Sigma; \Delta; \Gamma \vdash e : A\,[C']$.
2. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$*, then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C']$.
3. *if* $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash e : B\,[C]$*, then* $\Sigma; (\Delta, u{:}A[D']); \Gamma \vdash e : B\,[C]$
4. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$*, then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D'] \Rightarrow [C]$.

Proof: The first two statements are proved by straightforward simultaneous induction on the given derivations. We present only selected cases.

1. case $e = \langle\Theta\rangle u$.
   (a) by derivation, $u{:}A[C''] \in \Delta$ and $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C''] \Rightarrow [C]$
   (b) by 2. $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C''] \Rightarrow [C']$
   (c) result follows by typing

2. case $\Theta = (\cdot)$

   By derivation, $D \subseteq C \subseteq C'$ so the result follows by the typing rules.

   case $\Theta = (X \to e', \Theta')$.

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash e' : A\,[C]$ and $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [D \setminus \{X\}] \Rightarrow [C]$
   (b) by 1. $\Sigma; \Delta; \Gamma \vdash e' : A\,[C']$
   (c) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [D \setminus \{X\}] \Rightarrow [C']$
   (d) result follows by typing

3. Induction on the structure of the derivation.

   case $e = \langle \Theta \rangle u$.

   (a) by derivation, $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash \langle \Theta \rangle : [D] \Rightarrow [C]$
   (b) by induction on the structure of Θ, we get $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash \langle \Theta \rangle : [D] \Rightarrow [C]$
   (c) by 4. $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash \langle \Theta \rangle : [D'] \Rightarrow [C]$
   (d) result follows by typing

4. Induction on the structure of the derivation.

   case $\Theta = (\cdot)$. Then $D' \subseteq D \subseteq C$.

   case $\Theta = (X \to e, \Theta')$.

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [D \setminus \{X\}] \Rightarrow [C]$ and $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$
   (b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [D' \setminus \{X\}] \Rightarrow [C]$
   (c) result follows by typing

■

Lemma 6 (Support extension)
Let $D \subseteq \mathbf{dom}(\Sigma)$ be a well-formed support set. Then the following holds:

1. *if* $\Sigma; (\Delta, u{:}A[C_1]); \Gamma \vdash e : B\,[C_2]$ *then* $\Sigma; (\Delta, u{:}A[C_1 \cup D]); \Gamma \vdash e : B\,[C_2 \cup D]$
2. *if* $\Sigma; \Delta; \Gamma \vdash \langle \Theta \rangle : [C_1] \Rightarrow [C_2]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle \Theta \rangle : [C_1 \cup D] \Rightarrow [C_2 \cup D]$

Proof:

1. By induction on the structure of the derivation.

   case $e = x, X$ is trivially true by support weakening.

   case $e = \langle \Theta \rangle u$, where $A = B$.

   (a) by derivation, $\Sigma; (\Delta, u{:}A[C_1]); \Gamma \vdash \langle \Theta \rangle : [C_1] \Rightarrow [C_2]$
   (b) by induction on the structure of Θ, we get $\Sigma; (\Delta, u{:}A[C_1 \cup D]); \Gamma \vdash \langle \Theta \rangle : [C_1] \Rightarrow [C_2 \cup D]$
   (c) by 2. $\Sigma; (\Delta; u{:}A[C_1 \cup D]); \Gamma \vdash \langle \Theta \rangle : [C_1 \cup D] \Rightarrow [C_2 \cup D]$
   (d) result follows by typing

2. By induction on the structure of the derivation.

   case $\Theta = (X \to e, \Theta')$, where $X{:}A \in \Sigma$.

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash e : A\,[C_2]$ and $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [C_1 \setminus \{X\}] \Rightarrow [C_2]$
   (b) by support weakening, $\Sigma; \Delta; \Gamma \vdash e : A\,[C_2 \cup D]$
   (c) by induction hypothesis and support weakening, $\Sigma; \Delta; \Gamma \vdash \langle \Theta' \rangle : [(C_1 \cup D) \setminus \{X\}] \Rightarrow [C_2 \cup D]$
   (d) result follows by typing

■

Lemma 7 (Substitution merge)
If $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$ *and* $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_2] \Rightarrow [D]$ *where* $\mathbf{dom}(\Theta) \cap \mathbf{dom}(\Theta') = \emptyset$*, then* $\langle\Theta, \Theta'\rangle : [C_1 \cup C_2] \Rightarrow [D]$.
Proof: By induction on the structure of Θ'.

case $\Theta' = (\cdot)$. Then $C_2 \subseteq D$ and results follow by support extension (Lemma 6.2).

case $\Theta' = (X \to e, \Theta'_1)$.

1. by derivation, $\Sigma; \Delta; \Gamma \vdash \langle\Theta'_1\rangle : [C_2 \setminus \{X\}] \Rightarrow [D]$
2. by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle\Theta, \Theta'_1\rangle : [C_1 \cup (C_2 \setminus \{X\})] \Rightarrow [D]$
3. by support weakening, $\Sigma; \Delta; \Gamma \vdash \langle\Theta, \Theta'_1\rangle : [(C_1 \cup C_2) \setminus \{X\}] \Rightarrow [D]$
4. result follows by typing

■

A.2 Substitution principles

Lemma 8 (Explicit substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$*. Then the following holds:*

1. *if* $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$ *then* $\Sigma; \Delta; \Gamma \vdash \{\Theta\}e : A\,[D]$.
2. *if* $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$*, then* $\Sigma; \Delta; \Gamma \vdash \langle\Theta \circ \Theta'\rangle : [C_1] \Rightarrow [D]$.

Proof: By simultaneous induction on the structure of the derivations. We just present selected cases.

1. case $e = x$. The judgment is derived by weakening at the hypothesis, so we can also derive $x{:}A[D]$. This is sufficient, as $\{\Theta\}x = x$.

   case $e = \langle\Theta'\rangle u$.

   (a) by derivation, $u : A\,[C_1] \in \Delta$ and $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$.
   (b) by 2. $\Sigma; \Delta; \Gamma \vdash \langle\Theta \circ \Theta'\rangle : [C_1] \Rightarrow [D]$.
   (c) result follows by typing

   case $e = X$, where $X{:}A \in \Sigma$.

   (a) by derivation, $X \in C$.
   (b) if $X \in \mathbf{dom}(\Theta)$ with $e' = \Theta\ (X)$, then by typing of Θ, $\Sigma; \Delta; \Gamma \vdash e' : A\,[D]$
   (c) if $X \notin \mathbf{dom}(\Theta)$, then by derivation of Θ, it must be $X \in D$ since we assumed $X \in C$
   (d) by support weakening, $\Sigma; \Delta; \Gamma \vdash X : A\,[D]$
   (e) result follows by typing

2. Given the substitutions Θ and Θ', we split the representation of $\Theta \circ \Theta'$ into two disjoint sets:

$$\begin{array}{rcl} \Theta'_1 & = & \{X \to \Theta(X) \mid X \in \mathbf{dom}(\Theta) \setminus \mathbf{dom}(\Theta')\} \\ \Theta'_2 & = & \{X \to \{\Theta\}(\Theta'\,X) \mid X \in \mathbf{dom}(\Theta')\} \end{array}$$

   and set out to show that

   (a) $\Sigma; \Delta; \Gamma \vdash \langle\Theta'_1\rangle : [C_1 \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$, and
   (b) $\Sigma; \Delta; \Gamma \vdash \langle\Theta'_2\rangle : [C_1 \cap \mathbf{dom}(\Theta')] \Rightarrow [D]$.

These two typings imply the result by the substitution merge lemma (Lemma 7). The statement (b) follows from the typing of Θ' by support weakening (Lemma 5.4), and the first part of the lemma. To establish (a), observe that from the typing of Θ it is clear that $\Theta'_1 : [C \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$. But, since $C_1 \setminus \mathbf{dom}(\Theta') \subseteq C \setminus \mathbf{dom}(\Theta')$ readily follows from the typing of Θ', the result is obtained by support weakening.

■

Lemma 9 (Value substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash e_1 : A\,[C]$. The following holds:

1. *if $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e_2 : B\,[C]$, then $\Sigma; \Delta; \Gamma \vdash [e_1/x]e_2 : B\,[C]$*
2. *if $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \langle\Theta\rangle : [C'] \Rightarrow [C]$, then $\Sigma; \Delta; \Gamma \vdash \langle[e_1/x]\Theta\rangle : [C'] \Rightarrow [C]$*

Proof: Simultaneous induction on the two derivations.

1. case $e_2 = x, X$ obvious.

   case $e_2 = \langle\Theta\rangle u$.
   (a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \langle\Theta\rangle : [C'] \Rightarrow [C]$ and $u : B\,[C'] \in \Delta$.
   (b) by 2., $\Sigma; \Delta; \Gamma \vdash \langle[e_1/x]\Theta\rangle : [C'] \Rightarrow [C]$
   (c) result follows by typing.

   case $e_2 = \lambda y{:}B'.\, e'$, where $B = B' \to B''$.
   (a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A, y{:}B') \vdash e' : B''\,[C]$
   (b) by induction hypothesis, $\Sigma; \Delta; (\Gamma, y{:}B') \vdash [e_1/x]e' : B''\,[C]$
   (c) result follows by typing

   case $e_2 = e'\ e''$.
   (a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e' : B' \to B\,[C]$ and $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e'' : B'\,[C]$
   (b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [e_1/x]e' : B' \to B\,[C]$, and $\Sigma; \Delta; \Gamma \vdash [e_1/x]e'' : B'\,[C]$
   (c) result follows by typing

   case $e_2 = \mathbf{box}\ e'$.
   Trivial, since $x \notin \mathbf{fv}(e')$.

   case $e_2 = \mathbf{let\ box}\ u = e'\ \mathbf{in}\ e''$.
   (a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e' : \Box_{C'} B'\,[C]$ and $\Sigma; (\Delta, u{:}B'[C']); (\Gamma, x{:}A) \vdash e'' : B\,[C]$
   (b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [e_1/x]e' : \Box_{C'} B'\,[C]$ and $\Sigma; (\Delta, u{:}B'[C']); \Gamma \vdash [e_1/x]e'' : B\,[C]$.
   (c) result follows by typing

   case $e_2 = \nu X{:}B'.\, e'$, where $B = B' \twoheadrightarrow B''$.
   (a) by derivation, $(\Sigma, X{:}B'); \Delta; (\Gamma, x{:}A) \vdash e' : B''\,[C]$ and $X \notin \mathbf{fn}(B''[C])$
   (b) by induction hypothesis, $(\Sigma, X{:}B'); \Delta; \Gamma \vdash [e_1/x]e' : B''\,[C]$
   (c) result follows by typing

   case $e_2 = \mathbf{choose}\ e'$, where $\Sigma = (\Sigma', X{:}B')$.
   (a) by derivation, $\Sigma'; \Delta; (\Gamma, x{:}A) \vdash e' : B' \twoheadrightarrow B\,[C]$
   (b) by induction hypothesis, $\Sigma'; \Delta; \Gamma \vdash [e_1/x]e' : B' \twoheadrightarrow B\,[C]$
   (c) result follows by typing

   case $e_2 = \Lambda p.\, e'$ where $B = \forall p.\, B'$.
   (a) by derivation, $(\Sigma, p); \Delta; (\Gamma, x{:}A) \vdash e' : B'\,[C]$
   (b) by induction hypothesis, $(\Sigma, p); \Delta; \Gamma \vdash [e_1/x]e' : B'\,[C]$

(c) result follows by typing

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

(a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e' : \forall p.\, B'\,[C]$

(b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [e_1/x]e' : \forall p.\, B'\,[C]$

(c) result follows by typing

case $e_2 = \textbf{case } e' \textbf{ of box } \pi \Rightarrow e'_1 \textbf{ else } e'_2$.

(a) by derivation:

i. $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e' : \Box_D A_1\,[C]$

ii. $\Sigma; \cdot \vdash \pi : A_1\,[D] \Longrightarrow \Gamma_1$

iii. $\Sigma; \Delta; (\Gamma, x{:}A, \Gamma_1) \vdash e'_1 : B\,[C]$

iv. $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e'_2 : B\,[C]$

(b) by induction hypothesis:

i. $\Sigma; \Delta; \Gamma \vdash [e_1/x]e' : \Box_D A_1\,[C]$

ii. $\Sigma; \Delta; (\Gamma, \Gamma_1) \vdash [e_1/x]e'_1 : B\,[C]$

iii. $\Sigma; \Delta; \Gamma \vdash [e_1/x]e'_2 : B\,[C]$

(c) result follows by typing

2. case $\Theta = (\cdot)$ is trivial.

case $\Theta = (X \to e', \Theta')$.

(a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A) \vdash e' : B'\,[C]$ and $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \langle \Theta' \rangle : [C' \setminus \{X\}] \Rightarrow [C]$

(b) by 1. $\Sigma; \Delta; \Gamma \vdash [e_1/x]e' : B'\,[C]$

(c) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle [e_1/x]\Theta' \rangle : [C' \setminus \{X\}] \Rightarrow [C]$

(d) result follows by typing

■

Lemma 11 (Expression substitution principle)
Let e_1 be an expression without free value variables *such that $\Sigma; \Delta; \cdot \vdash e_1 : A\,[C]$. Then the following holds:*

1. *if $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e_2 : B\,[D]$, then $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e_2 : B\,[D]$*

2. *if $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta \rangle : [D'] \Rightarrow [D]$, then $\Sigma; \Delta; \Gamma \vdash \langle [\![e_1/u]\!]\Theta \rangle : [D'] \Rightarrow [D]$*

Proof:

1. By simultaneous induction on the two derivations.

case $e_2 = x, X$ obvious.

case $e_2 = \langle \Theta \rangle u$.

(a) by derivation, $A = B$ and $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta \rangle : [C] \Rightarrow [D]$

(b) by 2. $\Sigma; \Delta; \Gamma \vdash \langle [\![e_1/u]\!]\Theta \rangle : [C] \Rightarrow [D]$

(c) by principle of explicit substitution (Lemma 8.1), $\Sigma; \Delta; \Gamma \vdash \{[\![e_1/u]\!]\Theta\}e_1 : B\,[D]$

(d) but this is exactly equal to $[\![e_1/u]\!]e_2$

case $e_2 = \langle \Theta \rangle v$, where $v{:}B[D'] \in \Delta$

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta \rangle : [D'] \Rightarrow [D]$

(b) by 2., $\Sigma; \Delta; \Gamma \vdash \langle [\![e_1/u]\!]\Theta \rangle : [D'] \Rightarrow [D]$

(c) result follows by typing

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

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); (\Gamma, x{:}B') \vdash e' : B''\,[D]$
(b) by induction hypothesis, $\Sigma; \Delta; (\Gamma, x{:}B') \vdash [\![e_1/u]\!]e' : B''\,[D]$
(c) result follows by typing

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

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : B' \to B\,[D]$ and also $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e'' : B'\,[D]$
(b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : B' \to B\,[D]$, and also $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e'' : B'\,[D]$
(c) result follows by typing

case $e_2 = \mathbf{box}\ e'$, where $B = \Box_{D'} B'$

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : B'\,[D']$
(b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : B'\,[D']$
(c) result follows by typing

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

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : \Box_{D'} B'\,[D]$ and also $\Sigma; (\Delta, u{:}A[C], v{:}B'[D']); \Gamma \vdash e'' : B\,[D]$
(b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : \Box_{D'} B'\,[D]$ and $\Sigma; (\Delta, v{:}B'[D']); \Gamma \vdash [\![e_1/u]\!]e'' : B\,[D]$.
(c) result follows by typing

case $e_2 = \nu X{:}B'.\ e'$, where $B = B' \nrightarrow B''$.

(a) by derivation, $(\Sigma, X{:}B'); (\Delta, u{:}A[C]); \Gamma \vdash e' : B''\,[D]$ and also $X \notin \mathbf{fn}(B''[D])$
(b) by induction hypothesis, $(\Sigma, X{:}B'); \Delta; \Delta \vdash [\![e_1/u]\!]e' : B''\,[D]$
(c) result follows by typing

case $e_2 = \mathbf{choose}\ e'$, where $\Sigma = (\Sigma', X{:}B')$.

(a) by derivation, $\Sigma'; (\Delta, u{:}A[C]); \Gamma \vdash e' : B' \nrightarrow B\,[D]$
(b) by induction hypothesis, $\Sigma'; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : B' \nrightarrow B\,[D]$
(c) result follows by typing

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

(a) by derivation, $(\Sigma, p); (\Delta, u{:}A[C]); \Gamma \vdash e' : B'\,[D]$
(b) by induction hypothesis, $(\Sigma, p); \Delta; \Gamma \vdash [\![e_1/u]\!]e' : B'\,[D]$
(c) result follows by typing

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

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : \forall p.\ B'\,[D]$
(b) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : \forall p.\ B'\,[D]$
(c) result follows by typing

case $e_2 = \mathbf{case}\ e'\ \mathbf{of\ box}\ \pi \Rightarrow e'_1\ \mathbf{else}\ e'_2$.

(a) by derivation:
   i. $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : \Box_{D'} A_1\,[D]$
   ii. $\Sigma; \cdot \vdash \pi : A_1\,[D'] \Longrightarrow \Gamma_1$
   iii. $\Sigma; (\Delta, u{:}A[C]); (\Gamma, \Gamma_1) \vdash e'_1 : B\,[D]$
   iv. $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e'_2 : B\,[D]$
(b) by induction hypothesis:
   i. $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e' : \Box_{D'} A_1\,[D]$
   ii. $\Sigma; \Delta; (\Gamma, \Gamma_1) \vdash [\![e_1/u]\!]e'_1 : B\,[D]$
   iii. $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!]e'_2 : B\,[D]$

(c) result follows by typing

2. case $\Theta = (\cdot)$ is trivial.

   case $\Theta = (X \to e', \Theta')$.

   (a) by derivation, $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash e' : B' \, [D]$ and also $\Sigma; (\Delta, u{:}A[C]); \Gamma \vdash \langle \Theta' \rangle : [D' \setminus \{X\}] \Rightarrow [D]$
   (b) by 1., $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!] e' : B' \, [D]$
   (c) by induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle [\![e_1/u]\!] \Theta' \rangle : [D' \setminus \{X\}] \Rightarrow [D]$
   (d) result follows by typing

■

Lemma 15 (Support substitution principle)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathbf{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Then the following holds.

1. *if $\Sigma; \Delta; \Gamma \vdash e : A \, [C]$, then $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e' : A' \, [C']$*
2. *if $\Sigma; \Delta; \Gamma \vdash \Theta : C_1 \to C_2$, then $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta' : C_1' \to C_2'$*

Proof:

1. case $e = x, X$ is trivial.

   case $e = \langle \Theta \rangle u$.

   (a) by derivation $\Sigma; \Delta; \Gamma \vdash \Theta : C_1 \to C$ and $u{:}A[C_1] \in \Delta$
   (b) by definition $u{:}A'[C_1'] \in \Delta'$
   (c) by 2. $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta' : C_1' \to C'$
   (d) result follows by typing

   case $e = \lambda x{:}A_1.\, e_1$ where $A = A_1 \to A_2$.

   (a) by derivation, $\Sigma; \Delta; (\Gamma, x{:}A_1) \vdash e_1 : A_2 \, [C]$
   (b) by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; (\Gamma', x{:}A_1') \vdash e_1' : A_2' \, [C']$
   (c) result follows by typing

   case $e = e_1 \; e_2$.

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : A_1 \to A \, [C]$, and $\Sigma; \Delta; \Gamma \vdash e_2 : A_1 \, [C]$
   (b) by induction hypothesis $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : A_1' \to A' \, [C']$ and $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_2' : A_1' \, [C']$
   (c) result follows by typing

   case $e = \mathbf{box}\; e_1$, where $A = \Box_{C_1} A_1$.

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : A_1 \, [C_1]$
   (b) by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : A_1' \, [C_1']$
   (c) result follows by typing

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

   (a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : \Box_{C_1} A_1 \, [C]$ and $\Sigma; (\Delta, u{:}A_1[C_1]); \Gamma \vdash e_2 : A \, [C]$
   (b) by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : \Box_{C_1'} A_1' \, [C']$ and $(\Sigma_1, \Sigma_2'); (\Delta', u{:}A_1'[C_1']); \Gamma' \vdash e_2' : A' \, [C']$
   (c) result follows by typing

   case $e = \nu X{:}A.\, e_1$ where $A = A_1 \nrightarrow A_2$

   (a) by derivation, $(\Sigma, X{:}A); \Delta; \Gamma \vdash e_1 : A_2 \, [C]$
   (b) by induction hypothesis $(\Sigma_1, \Sigma_2', X{:}A'); \Delta'; \Gamma' \vdash e_1' : A_2' \, [C']$

(c) result follows by typing

case $e = \mathbf{choose}\ e_1$.

(a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : A_1 \twoheadrightarrow A\,[C]$
(b) by induction hypothesis $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : A_1' \twoheadrightarrow A'\,[C']$
(c) result follows by typing

case $e = \Lambda q.\ e_1$ where $A = \forall q.\ A_1$.

(a) by derivation, $(\Sigma, q); \Delta; \Gamma \vdash e_1 : A_1\,[C]$
(b) by induction hypothesis, $(\Sigma_1, \Sigma_2', q); \Delta'; \Gamma' \vdash e_1' : A_1'\,[C']$
(c) result follows by typing

case $e = e_1\,[C_1]$.

(a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : \forall q.\ A_1\,[C]$ and $A = [C_1/q]A_1$
(b) by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : \forall q.\ A_1'\,[C']$
(c) it is easy to see that $[C_1'/q]A_1' = ([C_1/q]A_1)'$
(d) result follows by typing

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

(a) by derivation:
   i. $\Sigma; \Delta; \Gamma \vdash e_0 : \Box_{D_1} A_1\,[C]$
   ii. $\Sigma; \cdot \vdash \pi : A_1\,[D_1] \Longrightarrow \Gamma_1$
   iii. $\Sigma; (\Gamma, \Gamma_1) \vdash e_1 : A\,[C]$
   iv. $\Sigma; \Gamma \vdash e_2 : A\,[C]$
(b) by induction hypothesis:
   i. $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_0' : \Box_{D_1'} A_1'\,[C']$
   ii. $(\Sigma_1, \Sigma_2'); (\Gamma', \Gamma_1') \vdash e_1' : A'\,[C']$
   iii. $(\Sigma_1, \Sigma_2'); \Gamma' \vdash e_2' : A'\,[C']$
(c) by principle of support substitution for pattern (Lemma 17), $(\Sigma_1, \Sigma_2'); \cdot \vdash \pi' : A_1'\,[D_1'] \Longrightarrow \Gamma_1'$
(d) result follows by typing

2. case $\Theta = (\cdot)$ is trivial.

case $\Theta = (X \to e_1, \Theta_1)$, where $X{:}B \in \Sigma$.

(a) by derivation, $\Sigma; \Delta; \Gamma \vdash e_1 : B\,[D]$ and $\Sigma; \Delta; \Gamma \vdash \Theta_1 : C_1 \setminus \{X\} \to C_2$
(b) by 1. $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e_1' : B'\,[C_2']$
(c) by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : C_1' \setminus \{X\} \to C_2'$
(d) result follows by typing

∎

A.3 Operational semantics

Theorem 12 (Type preservation)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$ and $\Sigma, e \longmapsto \Sigma', e'$, then $\Sigma'; \cdot; \cdot \vdash e' : A\,[\,]$.
Theorem 13 (Progress)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*

2. *there exist a term e' and a context Σ' extending Σ, such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof: We prove both theorems at once. The proof is by induction on the common typing derivation.

case $e = x, u, X$ cannot occur.

case $e = \lambda x{:}A.\ e'$ is already a value.

case $e = e_1\ e_2$ where e_1 is not a value.

1. by derivation, $\Sigma; \cdot; \cdot \vdash e_1 : A_1 \to A$ and $\Sigma; \cdot; \cdot \vdash e_2 : A_1$
2. by induction hypothesis, $\Sigma, e_1 \longmapsto \Sigma', e_1'$ where $\Sigma'; \cdot; \cdot \vdash e_1' : A_1 \to A$
3. then $e \longmapsto e_1'\ e_2$ which has the correct typing

case $e = v_1\ e_2$ where e_2 is not a value.

1. by derivation, $\Sigma; \cdot; \cdot \vdash v_1 : A_1 \to A$ and $\Sigma; \cdot; \cdot \vdash e_2 : A_1$
2. by induction hypothesis, $\Sigma, e_2 \longmapsto \Sigma', e_2'$ where $\Sigma'; \cdot; \cdot \vdash e_2' : A_1$
3. then $e \longmapsto v_1\ e_2'$ which has the correct typing

case $e = (\lambda x{:}A_1.\ e')\ v$.

1. by derivation, $\Sigma; \cdot; x{:}A_1 \vdash e' : A$ and $\Sigma; \cdot; \cdot \vdash v : A_1$
2. by value substitution principle (Lemma 9.1), $\Sigma; \cdot; \cdot \vdash [v/x]e' : A$

case $e = \mathbf{box}\ e'$ is already a value.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$ where e_1 is not a value.

1. by induction hypothesis, $\Sigma, e_1 \longmapsto \Sigma', e_1'$ of the same type

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

1. by derivation, $\Sigma; \cdot; \cdot \vdash e_1 : B\,[C]$ and $\Sigma; u{:}B[C]; \cdot \vdash e_2 : A$
2. by expression substitution principle (Lemma 11.1), $\Sigma; \cdot; \cdot \vdash [\![e_1/u]\!]e_2 : A$

case $e = \nu X{:}A_1.\ e$ is a value.

case $e = \mathbf{choose}\ e_1$ where e_1 is not a value.

1. by induction hypothesis, $\Sigma, e_1 \longmapsto \Sigma', e_1'$ with a preserved type

case $e = \mathbf{choose}\ \nu X{:}A_1.\ e$.

1. by derivation, $(\Sigma, X{:}A_1); \cdot; \cdot \vdash e : A$
2. take $\Sigma' = (\Sigma, X{:}A_1)$ for result

case $e = \Lambda p.\ e_1$ is a value.

case $e = e_1\ [C]$ where e_1 is not a value, and $A = [C/p]A_1$

1. by derivation, $\Sigma; \cdot; \cdot \vdash e_1 : \forall p.\ A_1$
2. by induction hypothesis, $\Sigma, e_1 \longmapsto \Sigma', e_1'$ where $\Sigma'; \cdot; \cdot \vdash e_1' : \forall p.\ A_1$
3. the result follows by typing

case $e = (\Lambda p.\ e_1)\ [C]$, where $A = [C/p]A_1$.

1. by derivation, $\Sigma; \cdot; \cdot \vdash_p e_1 : A_1$ and $C \subseteq \mathbf{dom}(\Sigma)$

2. by support substitution principle (Lemma 15.1), $\Sigma;\cdot;\cdot \vdash [C/p]e_1 : A\,[\,]$

case $e = \textbf{case box } e_0 \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2$

1. by derivation:
   (a) $\Sigma;\cdot;\cdot \vdash e_0 : B\,[C]$
   (b) $\Sigma;\cdot \vdash \pi : B\,[C] \Longrightarrow \Gamma_1$
   (c) $\Sigma;\cdot;\Gamma_1 \vdash e_1 : A\,[\,]$
   (d) $\Sigma;\cdot;\cdot \vdash e_2 : A\,[\,]$
2. assume $\Gamma_1 = (w_1{:}A_1, \ldots, w_n{:}A_n)$
3. if $\Sigma;\cdot \vdash e_0 \rhd \pi : B \Longrightarrow \sigma$, then by soundness of pattern matching (Lemma 18), $\sigma = (w_1 \to e'_1, \ldots, w_n \to e'_n)$, where $\Sigma;\cdot;\cdot \vdash e'_i : A_i\,[\,]$
4. furthermore, $\Sigma, e \longmapsto \Sigma, [e'_1/w_1, \ldots, e'_n/w_n]e_1$
5. by repeated use of value substitution principle (Lemma 9.1) over the derivation $\Sigma;\cdot;\Gamma_1 \vdash e_1 : A\,[\,]$, we obtain $\Sigma;\cdot;\cdot \vdash [e'_1/w_1, \ldots, e'_n/w_n]e_1 : A\,[\,]$
6. if the matching fails, then $\Sigma, e \longmapsto \Sigma, e_2$, but then by assumption $\Sigma;\cdot;\cdot \vdash e_2 : A\,[\,]$

■

A.4 Intensional analysis of higher-order syntax

Lemma 17 (Support substitution principle for pattern matching)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathbf{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Assume also that $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $(\Sigma_1, \Sigma'_2);\Gamma' \vdash \pi' : A'\,[C'] \Longrightarrow \Gamma'_1$.
Proof: By induction on the derivation.

case $\pi = (w\ x_1 \ldots x_n){:}A[C_1]$.

1. by derivation, $x_1{:}A_1, \ldots, x_n{:}A_n \in \Gamma$ and $\Gamma_1 = \forall q.\ \Box_q A_1 \to \cdots \to \Box_q A_n \to \Box_{q,C} A$ and also $C_1 \subseteq C$
2. by definition of the operation $(-)'$, $x_1{:}A'_1, \ldots, x_n{:}A'_n \in \Gamma'$ and $\Gamma'_1 = \forall q.\ \Box_q A'_1 \to \cdots \to \Box_q A'_n \to \Box_{q,C'} A'$ and also $C'_1 \subseteq C'$
3. result follows by typing

case $\pi = X$, where $X{:}A \in \Sigma$.

1. by derivation, $X{:}A \in \Sigma$ and $C = (C_1, X)$
2. by definition, $X{:}A' \in \Sigma'$ and $C' = (C'_1, X)$
3. result follows by typing

case $\pi = x$, where $\Gamma = \Gamma_1, x{:}A$.

1. by derivation, $\Gamma = \Gamma_1, x{:}A$
2. by definition, $\Gamma' = \Gamma'_1, x{:}A'$
3. result follows by typing

case $\pi = \lambda x{:}A_1.\ \pi_1$ where $A = A_1 \to A_2$.

1. by derivation, $\Sigma;(\Gamma, x{:}A_1) \vdash \pi_1 : A_2\,[C] \Longrightarrow \Gamma_1$
2. by induction hypothesis $(\Sigma_1, \Sigma'_2);(\Gamma', x{:}A'_1) \vdash \pi'_1 : A'_2\,[C'] \Longrightarrow \Gamma'_1$

3. result follows by typing

case $\pi = \pi_1\ \pi_2$.

1. by derivation, $\Sigma; \Gamma \vdash \pi_1 : A_1 \to A\,[C] \Longrightarrow \Gamma_2$ and $\Sigma; \Gamma \vdash \pi_2 : A_1\,[C] \Longrightarrow \Gamma_3$, where $\Gamma_1 = (\Gamma_2, \Gamma_3)$, and also $\mathbf{fn}(A_1) \subseteq \mathbf{dom}(\Sigma)$
2. by induction hypothesis, $(\Sigma_1, \Sigma_2'); \Gamma' \vdash \pi_1' : A_1' \to A'\,[C'] \Longrightarrow \Gamma_2'$ and $(\Sigma_1, \Sigma_2'); \Gamma' \vdash \pi_2' : A_1'\,[C'] \Longrightarrow \Gamma_3'$
3. $\mathbf{fn}(A') \subseteq \mathbf{dom}(\Sigma_1, \Sigma_2')$
4. result follows by typing

■

Lemma 18 (Soundness of pattern matching)
Let π be a pattern such that $\Sigma; \Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$, where $\Gamma_1 = (w_1{:}A_1, \ldots, w_n{:}A_n)$. Furthermore, let e be an expression matching π to produce a pattern assignment σ, i.e. $\Sigma; \Gamma \vdash e \rhd \pi : A \Longrightarrow \sigma$. Then $\sigma = (w_1 \to e_1, \ldots, w_n \to e_n)$ where $\Sigma; \cdot; \cdot \vdash e_i : A_1$, for every $i = 1, \ldots, n$.
Proof: By induction on the structure of π.

case $\pi = (w\ x_1 \ldots x_n){:}A[D]$, where $\Gamma = \Gamma_2, x_i{:}A_i$.

1. let $e' = (\Lambda p.\ \lambda y_i{:}\Box_p A_i.\ \mathbf{let\ box}\ x_i = y_i\ \mathbf{in\ box}\ e)$ and $A' = \forall p.\ \Box_p A_1 \to \cdots \to \Box_p A_n \to \Box_{p,D} A$
2. by typing derivation, $D \subseteq C$ and $x_i{:}A_i \in \Gamma$ and also $\Gamma_1 = (w{:}A')$
3. by matching derivation, $\Sigma; \cdot; (x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e : A\,[D]$, and $\sigma = (w \to e')$
4. by straightforward structural induction, $\Sigma; (x_1{:}A_1, \ldots, x_n{:}A_n); \cdot \vdash e : A\,[D]$
5. by support weakening, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash e : A\,[D, p]$
6. and thus also, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash \mathbf{box}\ e : \Box_{D,p} A\,[\,]$
7. and $(\Sigma, p); \cdot; (y_1{:}\Box_p A_1, \ldots, y_n{:}\Box_p A_n) \vdash \mathbf{let\ box}\ x_i = y_i\ \mathbf{in\ box}\ e : \Box_{D,p} A\,[\,]$
8. and finally, $\Sigma; \cdot; \cdot \vdash e' : A'\,[\,]$

cases $\pi = X, x$ are trivial.

case $\pi = \lambda x{:}A_1.\ \pi_1$, where $A = A_1 \to A_2$.

1. by typing derivation, $\Sigma; (\Gamma, x{:}A_1) \vdash \pi_1 : A_2\,[C] \Longrightarrow \Gamma_1$
2. by matching derivation, $e = \lambda x{:}A_1.\ e_1$ and $\Sigma; \Gamma, x{:}A_1 \vdash e_1 \rhd \pi_1 \Longrightarrow \sigma$
3. result follows by induction hypothesis on π_1

case $\pi = \pi_2\ \pi_3$.

1. by typing derivation, $\Sigma; \Gamma \vdash \pi_2 : A' \to A\,[C] \Longrightarrow \Gamma_2$ and $\Sigma; \Gamma \vdash \pi_3 : A'\,[C] \Longrightarrow \Gamma_3$, where $\Gamma_1 = (\Gamma_2, \Gamma_3)$
2. (by derivation also $\mathbf{fn}(A') \subseteq \mathbf{dom}(\Sigma)$ but that will not be used in the proof)
3. let $\Gamma_1 = (w_1{:}A_1, \ldots, w_n{:}A_n)$ and $\Gamma_2 = (w_1'{:}A_1', \ldots, w_m'{:}A_m')$
4. by matching derivation, $e = e_2\ e_3$ and $\sigma = (\sigma_2, \sigma_3)$, where $\Sigma; \Gamma \vdash e_2 \rhd \pi_2 : A' \to A \Longrightarrow \sigma_2$ and $\Sigma; \Gamma \vdash e_3 \rhd \pi_3 : A' \Longrightarrow \sigma_3$
5. by induction hypothesis, $\sigma_1 = (w_1 \to e_1, \ldots, w_n \to e_n)$ and $\sigma_2 = (w_1' \to e_1', \ldots, w_m' \to e_m')$ with the appropriate typings, and so the result follows

■