Meta-programming with Names and Necessity

Aleksandar Nanevski
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
aleks@cmu.edu

Abstract

Meta-programming languages provide infrastructure to generate and execute object programs at run-time. In a typed setting, they contain a modal type constructor which classifies object code. These code types generally come in two flavors: closed and open. Closed code expressions can be invoked at run-time, but the computations over them are more rigid, and typically produce less efficient residual object programs. Open code provides better inlining and partial evaluation of object programs, but once constructed, expressions of this type cannot in general be evaluated.

Recent work in this area has focused on combining the two notions into a sound system. We present a novel way to achieve this. It is based on adding the notion of *names* from the work on Nominal Logic and FreshML to the $\lambda^{\Box}$-calculus of proof terms for the *necessity* fragment of modal logic S4. The resulting language provides a more fine-grained control over free variables of object programs when compared to the existing languages for meta-programming. In addition, this approach lends itself well to addition of intensional code analysis, i.e. ability of meta programs to inspect and destruct object programs at run-time in a type-safe manner, which we also undertake.

Categories and Subject Descriptors

D.3.1 [**Software**]: Programming Languages — *Formal Definitions and Theory*

General Terms

Languages

Keywords

modal lambda-calculus, higher-order abstract syntax

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) makes it possible to also invoke the generated programs at run-time. This setup is usually refered to as *homogeneous* meta-programming [20].

Among the advantages of meta-programming and of its homogeneous and typed variant we distinguish the following (and see [20] 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. A particular instance of this idea is the functional programming concept of *staged computation*, and has been considered before in a typed setting [23, 24, 3].

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 the 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.

Languages in which programs can not only be composed and executed but also have their structure inspected add further advantages. Efficiency benefits from various optimizations that can be performed knowing the structure of the code. For example, Griewank reports in [8] 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 ability to execute a certain function as well as recurse over its structure: see [19] for examples in computer graphics and numerical analysis, and [18] for an example in machine learning and probabilistic modeling.

Recent developments in type systems for meta-programming have been centered around two particular modal λ-calculi: $\lambda^{\Box}$ and $\lambda^{\bigcirc}$. The first is a language of proof terms for the modal logic S4, whose necessity constructor $\Box$ annotates *valid* propositions [3, 15]. The second is the proof language for discrete linear temporal logic, whose modal operator $\bigcirc$ annotates the time-level separation between propositions [2]. Both calculi provide a distinction between levels of terms, and this explains their use in meta-programming. The lowest, level 0, is the meta language, which is used to manipulate the terms on level 1 (terms of type $\Box A$ in $\lambda^{\Box}$ and $\bigcirc A$ in $\lambda^{\bigcirc}$). This first level is the meta language for the level 2 containing another stratum of boxed and circled types, etc. Functional programming interpretation of these two constructors assigns type $\Box A$ to *closed code* i.e. to closed terms of type A, while $\bigcirc A$ is the type of *postponed* code, i.e. it classifies terms of type A which are associated with the subsequent time moment. Postponed code in $\lambda^{\bigcirc}$ may refer to outside context variables, as long as they are on the same temporal level, and this has contributed to it frequently being associated with the notion of *open* code. For this exact reason, the concept of code in $\lambda^{\bigcirc}$ is obviously broader, allowing for more expressiveness and generation of better and more optimized residual programs (as already observed in [2]), but, unlike $\lambda^{\Box}$, it has no language support for mixing of the code levels, and in particular, no language support for execution of the generated code.

There have been several proposed systems which incorporate the advantages from both languages, most notable being MetaML [11, 22, 1]. MetaML starts with the postponed/open code type of $\lambda^{\bigcirc}$ and strengthens the notion to introduce closed code as its refinement – as postponed code which happens to contain no variables declared outside of it. The approach of our paper is the opposite. Rather than refining the notion of open code, we relax the notion of closed code. We start with the system of $\lambda^{\Box}$, but provide the additional expressiveness by allowing the code to contain specified object variables as free (and rudiments of this idea have already been considered in [13]). The fact that a given code expression depends on a set of free variables will be reflected in its type. The object variables themselves are represented by a separate semantic category of names (also called symbols or atoms), which admits equality. The treatment of names is adopted, with significant modifications, from the work on Nominal Logic and FreshML by Pitts and Gabbay [7, 17, 16, 6]. This design choice lends itself well to the addition, in an *orthogonal way*, of intensional code analysis, which we also undertake for the simply-typed segment of the language. Thus, we can also treat our simply-typed code expressions as data; they can not only be evaluated, but can also be compared for structural equality and destructed via pattern-matching, much in the same way as one would work with any abstract syntax tree.

2 Background

In this section we review the basic development of $\lambda^{\Box}$-calculus. We describe only the core language, but in the presented examples we assume the presence of certain types and term constructs, like integers, conditionals or recursion. We refer the reader to the accompanying technical report [12] for a more detailed treatment of this and other related work. The example we use throughout for illustration is the exponentiation function, presented below in a MinML-like notation.

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

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

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

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

The type system of $\lambda^{\Box}$ allows the programmer to specify the intended staging of operations, so that computations from the subsequent stages are independent of the computations from the current stage. This is achieved by explicitly annotating the stages of the computation and requiring that each stage is a *closed* term, i.e. that it is free of variables declared in the surrounding code. Then the type system can check whether the written code conforms to the staging specification, making staging errors into type errors.

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

To declare that a subterm e of type A is closed, $\lambda^{\Box}$ provides the type constructor $\Box$ and its introduction term **box**, so that **box** e has type $\Box A$ (consult the typing rules below). It is in this sense that the type constructor $\Box$ is associated with closed code. In the spirit of this “run-time code generation” interpretation, the operational semantics does not proscribe reductions under the box; boxed expressions are values. For the purposes of this paper, we will consider boxed code expressions to be uncompiled, i.e. stored and carried around in the form of their *abstract syntax trees*.

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

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

$$\frac{\Delta;\Gamma \vdash e_1 : A \to B \qquad \Delta;\Gamma \vdash e_2 : A}{\Delta;\Gamma \vdash e_1\ e_2 : B} \qquad \frac{\Delta;\cdot \vdash e : A}{\Delta;\Gamma \vdash \mathbf{box}\ e : \Box A}$$

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

$$\frac{}{c \hookrightarrow c} \qquad \frac{}{\lambda x{:}A.\ e \hookrightarrow \lambda x{:}A.\ e}$$

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

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

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

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

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

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

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

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

This $\lambda^{\Box}$ staging of `powbox` leaves a lot to be desired. In particular, the residual programs that `powbox` produces, e.g. `sqbox`, contain variable-for-variable redices, and hence are not as efficient as one would want. Ideally, we would like to completely inline all the function calls from `sqbox` and obtain `sqbox = box(λx:int. x*x*1)`. The reason the unwanted redices occur is, of course, because boxed code expressions are values; they completely suspend the evaluation of the enclosed term. As witnessed by the example of `sqbox`, it may be advantageous to have a general programming mechanism[1] whereby one could specify that certain reductions in a code expression are to take place. Of course, $\lambda^{\Box}$ already contains mechanisms to encode substitutions of closed code, but there is no way to perform substitutions of open code which is required in the `sqbox` example. The solution should be to extend the notion of code to include not only closed expressions, but also expressions which may contain free variables.

3 Core language

In this section we present the syntax and static semantics of our core language. It extends the $\lambda^{\Box}$ calculus with constructs for a unified treatment of the notions of closed and open code – a problem which initiated the extension of $\lambda^{\bigcirc}$ into MetaML [11, 22]. The motivational distinction between these two systems and our calculus is that we want to provide intensional code analysis as part of the language, while $\lambda^{\bigcirc}$ and MetaML do not do that. To be clear, we believe that extending $\lambda^{\bigcirc}$ or MetaML with code analysis is possible. It will most likely require similar machinery as developed here, except that the considerations would probably be more complicated because these calculi are more involved than $\lambda^{\Box}$. At any rate, the extension of $\lambda^{\Box}$ with the machinery required by code analysis already attains enough expressiveness to encode quite a few, if not all, interesting programs from $\lambda^{\bigcirc}$ and MetaML. Understanding the exact relationship between all these languages, however, remains future work.

Our approach starts with the closed code of $\lambda^{\Box}$, and allows a code expression to contain only those free variables that have been listed as dependencies in its type. Only the expressions with no dependencies will be executable. To begin with, we handle free variables of a code expression not as meta-level bound variables (as it happens to be the case in $\lambda^{\bigcirc}$ and MetaML), but by a separate binding and abstraction mechanism. The main reason is the following: intensional code analysis ought to provide a test whether two free variables in a code expression are different or equal. The result of this test is obviously not preserved under substitution, so it looks questionable to tie the free code variables to outside lambda abstractions. Having them both tied to the same mechanism of variable binding will almost certainly cause problems in the long run (as witnessed by, e.g., interaction of code analysis with cross-stage persistence in MetaML, explained in

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

[21]). The introduction of a separate binding mechanism for free variables of syntactic code expressions has been proposed before in [20], and even earlier in [10].

Furthermore, we do not want the variable introduction form of this second mechanism to be a type introduction form as well. The reason for this is that we want to support recursion over syntax trees of code expressions. A function which scans syntactic code expressions and recurses under a lambda binder, has to provide some symbol to stand for the bound variable, before it can go on and recurse over the body of the abstraction. Introduction of that temporary symbol should not change the type of the recursing function. Thus, we need to resort to *names* (see for example [14]). Additionally, we opt to separate the operation of name creation (renameability), from the name abstraction (hiding of a name), because that provides strictly more expressiveness in manipulation of code and names, than if the two are combined into a single constructor. This is where we employ the mechanisms of Nominal Logic and FreshML, which were designed with exactly that purpose in mind (see [16] and [17]). We introduce a new semantic category of *names* (also called symbols, atoms or indeterminates) which are to stand for free variables in boxed expressions. Thus, boxed expressions, as before in $\lambda^{\Box}$, cannot contain free variables, but we allow them to contain names, under the provision that the occurring free names are listed in the type of the expression. Correspondingly, the boxed types are now of the form $\Box(A[C])$ where C is a finite set of names[2] that the boxed term may depend on.

Informally, a term *depends* on a certain name if that name must be provided with a definition before the term can be evaluated. The set of names that a term depends on is called the *support* of the term. The notion of evaluation that we have in mind in these definitions is the one from $\lambda^{\Box}$ calculus. In particular, since the boxed expressions in $\lambda^{\Box}$ are values, the boxed expressions must have empty support.

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 our language, the term

$$t_1 = X^3 + 3X^2Y + 3XY^2 + Y^3$$

would have type *int* and support set $\{X,Y\}$. Indeed, in order to evaluate t_1 to an integer, we first need to substitute integer values for X and Y, and thus t_1 depends on both X and Y. On the other hand, if we box the term t_1, we obtain

$$t_2 = \textbf{box}\ (X^3 + 3X^2Y + 3XY^2 + Y^3)$$

which has type $\Box(int[X,Y])$, but its support is the empty set, as t_2 is already a value. Notice how support of a term (in this case t_1) becomes part of the type, once the term itself is boxed. This way, the types maintain the information about the support of subterms of all code levels – no matter under how many **box**'s a subterm may appear. For example, the term

$$t_3 = \langle X^2, \textbf{box}\ Y^2 \rangle$$

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

[2]Actually, C will have a bit more complex structure, to be introduced shortly

As only expressions with empty support can actually be evaluated, we needs a construct that would eliminate a name from the expression's support, eventually turning unexecutable expressions into executable ones. The construct for that is $\{X \doteq e_1\}\ e_2$, and it stands for *explicit substitution* of the value of e_1 for the occurrences of the name X on the *current* code level of e_2. Notice the emphasis on the current code level; the explicit name substitution of X only removes the occurrences of X which actually contribute to the support of e_2. It does not (and it would not be sound if it did) remove those occurrences of X which lie under one or more **box** constructors. This way, name substitution provides *extensions*, i.e. definitions for names, while still allowing names under boxes to be used for the *intensional* information of their identity.

Another construct that we need in the language is *name abstraction*. Quite often we need to express that a term depends on *some* name, but it is not really important how that name is called (or the name is not accessible in the local context). For example, such a need arises when recursing over a syntax tree for a λ-expression. Before descending under the λ, a temporary name has to be introduced on the fly to stand for the bound variable, but the identity of that name is not really important. We adopt the treatment of name abstraction from Nominal Logic and FreshML. For example, if X is a name of type P, the construct for abstracting the name X would be $X\,.\,(-)$, and it has a corresponding type constructor $\underset{X:P}{\text{И}}(-)$, which binds the occurrences of the name X in the supplied type. The intended operational semantics of $X\,.\,e$ is to *pair up* the name X and the *value* of e into a *closure*, thus explicitly hiding the identity of X (or, which is equivalent, returning the α-equivalence class of e with respect to X). The quantifier И has already been investigated in [6] and [16], but it has not been used explicitly in the definition of FreshML.

Just as in FreshML, the elimination form for name abstraction is *name concretion*. Its syntax is $e@Y$, where e is a name abstraction and Y is a name not occurring in e. Its operational meaning is to swap Y with the name abstracted in e.

For example, assuming as before that $X,Y{:}int$ are available names, we can create the term

$$t_4 = X\,.\,\textbf{box}\ (X^2 - 1)$$

which depends on *one* name (e.g. a "polynomial" in one indeterminate), ignoring the exact identity of that name. The type of t_4 is $\underset{Z:int}{\text{И}}\Box(int[Z])$, reflecting the fact that the actual indeterminate is not really known. But, if a need arises to manipulate the unknown indeterminate, we can always provide a fresh name for it by concretion, like in the term

$$t_5 = t_4 @ Y$$

which reduces to $\textbf{box}\ (Y^2 - 1)$.

We also need a way to dynamically introduce fresh names into the computation. Just like in FreshML, this duty is given to the term constructor

$$\textbf{new}\ X{:}P\ \textbf{in}\ e$$

which creates a new local name X and proceeds to evaluate e in the extended environment. Just like in FreshML, the type system will make sure that the value of e does not contain unsubstituted or unabstracted occurrences of X. Unlike in FreshML, where types of names belong to a separate universe, our name can have arbitrary type P, as long as P is simple (i.e. $\Box$-free). Notice that this constraint on simple types is fairly arbitrary; we wanted to understand the restricted language first before we extend it and generalize it.

As $\Box$-types can now explicitly store the information about the support of the terms they classify, another feature we need to consider is explicit support polymorphism. A program may want to manipulate code expressions no matter what their support sets are, or code expressions whose supports are unknown at compile time. A typical example would be a function which scans over some boxed term. When it encounters a lambda expression, it has to place a fresh name instead of the bound variable, and recursively continue scanning the body of the lambda, which is itself a boxed expression, but depending on this newly introduced name. For such uses, we extend the notion of support of a term to not only list the names appearing in the term, but to also allow variables standing for unknown support sets. Our language provides a term construct $\Lambda p\#K.\ e$ of type $\forall p\#K.\ A$ which is a polymorphic abstraction of an unknown support set p disjoint from a set of names K. Both the constructs bind the variable p, and two terms/types differing only by α-variation of the bound variable are considered equal. When K is empty, we abbreviate the constructs into $\Lambda p.\ e$ and $\forall p.\ A$. The term $e\ [\![C]\!]$ is the polymorphic instantiation, substituting a support set C for the support variable bound in e.

The syntax of our language is presented in Figure 1. Similarly to $\lambda^{\Box}$, we make a distinction between ordinary (value) variables and expression variables. We further distinguish between expression variables and expressions that have empty support, and those that may depend on some name; the first kind can be compiled and executed, and the second cannot. In analogy with Kripke semantics for Modal Logic, we will call the first kind *reflexive*, and the second kind *nonreflexive*. Thus, a variable context Γ may contain three forms of variable typings: $x{:}A$ for value variables, $u{::}A[C]$ for reflexive and $t \mathrel{\not{::}} A[C]$ for nonreflexive expression variables with support C. Notice that in a seeming contradiction to the definition of reflexive expression variables, we allow the support C to occur in their typing. The reason for it is in the interaction of expression variables with the term constructor **box** for expressions. Expression variables which are non-reflexive (and hence unreachable) outside a boxed term, will be accessible in the inside of it. It is in this sense that non-reflexive expressions cannot be executed, but only substituted into other non-reflexive expressions. This intuition will be formalized later in the typing judgment where the rule for box introduction will change the status of non-reflexive variables into reflexive ones. We call the type A with a support C an *annotated type*.

The support variable context Δ associates support variables with *disjointness annotations*. For example, $p\#K \in \Delta$ would mean that the support set variable p stands for an unknown support set C such that: (1) C contains no names from the set K, and (2) if $q \in C$ is a support variable, then $q\#K \in \Delta$, too. Enlarging an appropriate context by a new variable or a name is subject to Barendregt's Variable Convention: the new variables are assumed distinct, or are renamed in order not to clash with already existing ones. Terms which differ only in names of their bound variables are considered equal. The type constructor $\text{И}_{X:P}\, A$ is also a binder, abstracting a name $X{:}P$ from the type A, but it does not introduce a new name into the name context. As usual, capture avoiding substitution is defined to rename variables and names when descending into their scope. Free support variables of a given type A are denoted by $\mathbf{fp}(A)$, free variables of a term e by $\mathbf{fv}(e)$, and its free names are $\mathbf{fn}(e)$.

Example 1 To illustrate our language constructs and motivate the further development, we present a version of the staged exponentiation function that we could write in our system. In this example we assume that the language is extended with the base type of integers. In general, in the examples throughout the paper, we will assume the additional type and term constructs as we need them, either to illustrate the point or just improve readability. In any of the cases, the addition should not impose tremendous technical difficulties.

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

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

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

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

Notice that the generated residual code for `sqcode` does not contain any unnecessary redices, in contrast to the $\lambda^{\Box}$ version of the program from Section 2.

3.1 Auxiliary judgments

In order to state the typechecking rules, we will need a couple of auxiliary judgments. First is the judgment for *disjointness* (also referred to as *freshness*) of support sets. It has the form $\Delta \vdash C\ \#\ K$, where Δ is a context storing support variables with their freshness annotations, C is a support set, and K is a set of names. The judgment is satisfied if none of the names from K appears in C, and if all the variables from C are declared disjoint from K in the context Δ.

$$\begin{array}{llcl}
\textit{Simple types} & P & ::= & 1 \mid P_1 \to P_2 \\
\textit{Types} & A & ::= & 1 \mid A_1 \to A_2 \mid \Box(A[C]) \mid \mathop{\text{И}}_{X:P} A \mid \forall p\#K.\, A \\
\textit{Terms} & e & ::= & * \mid x \mid X \mid \lambda x{:}A.\, e \mid e_1\, e_2 \mid \mathbf{box}\, e \mid \mathbf{let\ box}\, u = e_1\ \mathbf{in}\ e_2 \mid \\
 & & & X\,.\,e \mid e@X \mid \mathbf{new}\ X{:}P\ \mathbf{in}\ e \mid \Lambda p\#K.\, e \mid e\,[\![C]\!] \mid \{a \doteq e_1\}\ e_2 \mid \mathbf{fix}\, x{:}A.\, e \\
\textit{Variable contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, t∺A[C] \mid \Gamma, u{::}A[C] \\
\textit{Name contexts} & S & ::= & \cdot \mid S, X{:}P \\
\textit{Supportvariable contexts} & \Delta & ::= & \cdot \mid \Delta, p\#K
\end{array}$$

Figure 1. Syntax of the core language (K is a finite set of names, and C is a finite set of names and support variables).

The concept of disjointness for support sets is then extended to disjointness for types, so that we have a judgment $\Delta \vdash A \ \#\ X$, where A is a type and X is a name. It is satisfied if X does not appear free in the dependencies of A on any code level. We will often combine the two judgments into a new judgment for disjointness of *annotated types* $\Delta \vdash A[C] \ \#\ X$.

Also required is a judgment to decide if a given type A is well-formed in the name context S and parameter context Δ, i.e. whether all the *free names* and support variables of A are declared in S or Δ, respectively. We denote it as $S;\Delta \vdash A$ wf.

We next define weakening on types: if a type depends on a certain set of names, we can always pass it as a type with a superset of names instead. Notice that we may need to alpha-rename the bound names when comparing two И-types.

$$\frac{}{1 \leqslant: 1} \qquad \frac{B_1 \leqslant: A_1 \quad A_2 \leqslant: B_2}{A_1 \to A_2 \leqslant: B_1 \to B_2} \qquad \frac{A \leqslant: B}{\mathop{\text{И}}_{X:P} A \leqslant: \mathop{\text{И}}_{X:P} B}$$

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

Finally, we implicitly equate two types A and B if $A \leqslant: B$ and $B \leqslant: A$. This is justified by the fact that two types will be in this relation iff they differ only in the ordering of names and variables in their supports. But support sets are indeed considered sets, so this ordering should not matter.

3.2 The type system

The typing judgment of our language has the form

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

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

Before proceeding further, we define an operation $\Gamma^\triangledown$ on variable contexts. It erases the ordinary variables from Γ and changes nonreflexive expression hypotheses $t∺A$ into reflexive ones $t{::}A$. As already hinted before, it will be used in the box introduction rule to make the non-reflexive variables accessible under the box.

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

The typing rules of our language are presented in Figure 2. We explain the most important ones of them next.

Hypothesis rules Notice first that the hypotheses rules exist only for the ordinary value variables and for the reflexive expression variables. The non-reflexive variables cannot be accessed until they are turned into the reflexive ones by the rule for **box** introduction. Thus, as already commented before, non-reflexive code expressions cannot be evaluated, but can only be used to compose new code expressions. The intention behind this is to prevent evaluation of code which is not closed. In addition, all the hypothesis rules check if the their support sets are well-formed, i.e. if all the names and support variables are declared in the name context S and the dependency variable context Δ.

λ-calculus fragment The rule for λ-abstraction relies on one of the auxiliary judgments to check whether the type A of the bound variable is well-formed, i.e. whether all its names and support variables have been already declared in the name context S and support variable context Δ. This ensures that the contexts used in the judgment are kept well-formed. The synthesized type B does not have to be checked for well-formedness, as the typing rules guarantee it.

Modal fragment Just as in $\lambda^\Box$-calculus, our rule for **box** checks the boxed expression e against a variable context $\Gamma^\triangledown$ from which the value variables have been erased. In addition, $\Gamma^\triangledown$ changes the status of all the nonreflexive expression variables into reflexive ones, so that they can be used in e (e being on a higher code level from **box** e). The support set of **box** e is empty, and thus it can be freely extended in the judgment by a well-formed support set D. We also have two different rules for the **let box** constructor: one classifies its local variable as reflexive, the other classifies it as non-reflexive, depending on the support set of the expression bound to the variable.

Names fragment The construct **new** generates a fresh name, and then checks, using the auxiliary disjointness judgment, if the synthesized type and support set do not contain free occurrences of this new name. The operation $\Delta\#X$ extends with X the freshness annotation of every support variable in Δ. This is justified because X is a new name, and is necessary in order to type possible abstractions with name X in the body of **new**.

In our system, just like in FreshML, the process of name abstraction and concretion is *separated* from name creation which is carried out by `new`. Thus, the typing rules for abstraction, concretion and explicit substitution require that the name they use has already been placed into the name context. The side condition $\Delta \vdash \mathbf{fp}(A) \ \#\ X$ in the typing rules for abstraction and concretion is a bit harder to explain. It ensures that the

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

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

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

$$\frac{S;\Gamma^{\triangledown} \vdash_\Delta e : A\,[C] \quad D \subseteq \mathbf{dom}(S,\Delta)}{S;\Gamma \vdash_\Delta \mathbf{box}\, e : \Box(A\,[C])\,[D]} \qquad \frac{S;\Gamma \vdash_\Delta e_1 : \Box(A[\,])\,[C] \quad S;\Gamma,u{::}A[\,] \vdash_\Delta e_2 : B\,[C] \quad u \notin \mathbf{dom}(\Gamma)}{S;\Gamma \vdash_\Delta \mathbf{let\ box}\, u = e_1 \,\mathbf{in}\, e_2 : B\,[C]}$$

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

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

$$\frac{S,X{:}P;\Gamma \vdash_{\Delta\#X} e : A\,[C] \quad \Delta\#X \vdash A[C]\ \#\ X \quad X \notin \mathbf{dom}(S)}{S;\Gamma \vdash_\Delta \mathbf{new}\, X{:}P \,\mathbf{in}\, e : A\,[C]}$$

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

$$\frac{S,X{:}P;\Gamma \vdash_\Delta e_1 : P\,[C] \quad S,X{:}P;\Gamma \vdash_\Delta e_2 : B\,[X,C]}{S,X{:}P;\Gamma \vdash_\Delta \{X \doteq e_1\}\, e_2 : B\,[C]} \qquad \frac{S;\Gamma \vdash_\Delta e : A\,[C] \quad A \leqslant: B}{S;\Gamma \vdash_\Delta e : B\,[C]}$$

Figure 2. Typing rules of the core language.

support variables occurring in e could not be substituted with a set containing the name X. If that were possible, the new occurrence of X would be abstracted on the level of terms, but there would be no binding in the corresponding type, thus causing unsound behavior. The reason for that is that the quantifier in $\underset{X:p}{\text{И}} A$ is itself a binder, and two name abstraction types which differ only in the identities of their bound names, are considered equal.

Another important observation about the rule for name abstraction is that it does not change the support C of the involved term. In particular, it does not remove the abstracted name from it. This is justified by the intended interpretation of name abstraction $(X \mathbin{.} e)$: it first *evaluates* its body e before creating the closure with X. Thus, the set of names that need to be provided with definitions in order to evaluate $(X \mathbin{.} e)$ is the same set required for the evaluation of e itself. In other words, the two expressions have the same support. Similar considerations motivate the typing rule for concretion as well.

Subtyping As already commented before, the nature of support sets makes it natural to pass a type with a smaller support annotation when a type with a bigger support annotation is required. Thus, we provide a rule that explicitly coerces terms into types with extended support, as defined by one of the auxiliary judgments.

In the rest of this section we present the basic structural properties of our calculus. As a first step, notice that the usual properties of exchange, weakening and contraction for variable context Γ hold in our system, too. In addition, we have exchange and weakening for name contexts, support variable contexts and support sets themselves. Strengthening of variable contexts holds as well: if a term which does not mention a certain variable $x \in \Gamma$ is well typed, then it is well typed in the context obtained by omitting x from Γ.

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

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

$$\begin{array}{rcl}
(\cdot)^{\triangle} & = & \cdot \\
(\Gamma, x{:}A)^{\triangle} & = & \Gamma^{\triangle}, x{:}A \\
(\Gamma, t\mathrel{∺}A[C])^{\triangle} & = & \Gamma^{\triangle}, t\mathrel{∺}A[C] \\
(\Gamma, u{::}A[\emptyset])^{\triangle} & = & \Gamma^{\triangle}, u{::}A[\emptyset] \\
(\Gamma, u{::}A[C])^{\triangle} & = & \Gamma^{\triangle}, u\mathrel{∺}A[C] \;\; \text{if } C \neq \emptyset
\end{array}$$

The next step is to define a capture-avoiding name substitution $\{X/e\}e'$. It substitutes the name X by e, but *only on the current code level* in e'; the occurrences of X on higher code levels (i.e. under boxes), as well as the names in abstracting and concreting positions, or in polymorphic abstractions and instantiations will not be touched. This operation and its corresponding substitution principle (Lemma 1.4) will be used to justify the operational semantics of the term construct for name substitution $\{X \doteq e\}\ e'$.

We further adopt the operation $(X\ Y)(-)$ of *name transposition* (or *name swapping*) from Nominal Logic and FreshML [17]. The operation interchanges all the occurrences of names X and Y in the argument expression/type/context/support set. Name transposition is different from name substitution: the former swaps two names throughout the given term or type, no matter the code level on which any of the names occur, while the later only works on the current code level.

Lemma 1 (Substitution Principles)

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

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

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

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

The premises in the formulation of the substitution principles deserve further elaboration. Principle 1.2 requires that the substituted term e_1 is typable in a context $\Gamma_1^{\ominus}$, i.e. that it does not contain any free value variables. The intuition behind this is that e_1 substitutes an *expression* variable u. Expression variables may occur on multiple code levels, so the substitution will copy e_1 to multiple code levels too. But the ordinary value variables are anchored by the type system to only the current code level, and thus e_1 must contain none of them. Similar considerations guide the formulation of Principle 1.3. The added twist is that the nonreflexive expression variables from Γ_1 can be treated as reflexive in e_1 because e_1 will not occur in executable positions in the residual term.

Most importantly, Principle 1.4 requires that the context in both the second premise and in the conclusion be of special form $\Gamma_2^{\triangle}$, i.e. that its reflexive variables only have empty support. Note that the principle describes a way to reduce the support of a term e_2 by substituting away the name X. But, the way the operation of name substitution is defined, it may not necessarily change the expression e_2 itself. For example, consider the case when $e_2 = u$ in the context $\Gamma = u{::}A[X]$. The substitution $\{X/e_1\}u$ produces a term u itself, but there is no typing $S;\Gamma \vdash_{\Delta} u : A[\,]$. That is why we require that the involved reflexive variables have no support. In retrospect, the need to distinguish between expression variables with and without support, which arises from this principle, was the main reason why we introduced nonreflexive variables into the design of the type system at all, instead of staying with only the reflexive variables of $\lambda^{\Box}$.

Another observation of crucial importance is that the local variables of a boxed expression form a context Γ, which is exactly of the form the Principle 1.4 requires, i.e. $\Gamma = \Gamma^{\triangle}$. This can easily be seen, as all the reflexive variables which will be put into the context have empty support (see the typing rules for **let box** in Figure 2). This would allow us to use the meta operation of name substitution $\{e_1/X\}e_2$ to define the operational semantics of the language construct for name substitution $\{X \doteq e_1\}\ e_2$. The idea is to use this construct to perform substitutions within box-annotated expression, and the Principle 1.4 ensures that these substitutions can be carried out without the postponement of evaluation which is the usual operational semantics associated with boxed expressions in $\lambda^{\Box}$.

The following lemma describes the behavior of typing with respect to substitution and name transposition. It is used in the proof of the Type Preservation and Progress theorem to justify the operational semantics assigned to the term constructors for support-polymorphic instantiation and concretion.

Lemma 2 (Parametricity)

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

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

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

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

4 Operational semantics

In this section we define the structured operational semantics for our core language, and formulate the appropriate Progress and Type Preservation theorem. We start by introducing the notion of *contraction*, which will be instrumental in defining the *values* of our language. The idea is that we do not consider, like in $\lambda^{\Box}$, that all boxed expressions are values. Rather, in order to be values, boxed expressions have to be "contracted", i.e. not reduced completely, but only freed of (some) name substitution they may contain. The name substitutions that are carried out (i.e. contracted) under a box in a given expression satisfy two properties: (1) they occur on the current code level, and (2) the substituted name is created outside of the boxed term, rather than being local to it. This is in accordance with the above observation about the Substitution Principle 1.4 that the variable context Γ of variables encountered when traversing the current code level of a boxed term, *and not descending into further and further boxes*, is always of a form $\Gamma = \Gamma^{\triangle}$.

Thus, the said substitution principle is applicable, and the encountered name substitutions can actually be carried out without postponing.

The judgment for contraction has the form

$$e \xrightarrow{S} w$$

and means: if the name substitutions in the expression e of names *other than those* in S are carried out, we obtain w. The "protected" set S carries the locally defined names of e (see the contraction rule for **new**), and is introduced in order to comply with the requirement (2) from above. The judgment is defined with the rules

$$\frac{e \xrightarrow{S,X:P} w}{\textbf{new}\ X{:}P\ \textbf{in}\ e \xrightarrow{S} \textbf{new}\ X{:}P\ \textbf{in}\ w}$$

$$\frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2 \qquad X \in \textbf{dom}(S)}{\{X \doteq e_1\}\ e_2 \xrightarrow{S} \{X \doteq w_1\}\ w_2}$$

$$\frac{e_1 \xrightarrow{S} w_1 \qquad e_2 \xrightarrow{S} w_2 \qquad X \notin \textbf{dom}(S)}{\{X \doteq e_1\}\ e_2 \xrightarrow{S} \{w_1/X\}w_2}$$

and is structural (i.e. commutes) with the other language constructs. An expression e is *S-contracted* if $e \xrightarrow{S} e$. It is *contracted* if and only if it is $\emptyset$-contracted. We use the letter w to range over S-contracted expressions.

Lemma 3 (Contraction Termination)
If $S_1, S_2; \Gamma^{\triangle} \vdash_{\Delta} e : A\,[C]$ then there exists unique term w, such that $e \xrightarrow{S_2} w$. Furthermore, w is S_2-contracted and $S_1, S_2; \Gamma^{\triangle} \vdash_{\Delta} w : A\,[C]$.

We can now define our syntactic category of values.

$$v \quad ::= \quad * \mid \lambda x.\ e \mid X\ .\ v \mid \Lambda p\#K.\ e \mid \textbf{box}\ w$$

It is not difficult to prove that name substitution preserves S-contracted expressions. In the same way, name transposition preserves S-contracted expressions as well, and as a consequence, it also preserves values.

We are now in position to define a small-step operational semantics (see Figure 3), and formulate the Type Preservation and Progress theorem for the core part of the language. Note that the theorem requires empty variable contexts and support.

Theorem 4 (Progress and Type Preservation)
If $S; \cdot \vdash e : A\,[\,]$, then either

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

5 Intensional code analysis

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

$$\begin{array}{lll} \textit{Patterns} \quad \pi & ::= & * \mid x \mid X \mid [E\ x_1 \cdots x_n] \mid \lambda x{:}P.\ \pi \mid \\ & & (\pi_1)\ (\pi_2{:}P) \end{array}$$

The higher-order pattern $[E\ x_1 \cdots x_n]$ declares a pattern variable E matching a code expression subject to condition that the expression's free variables are among $x_1, \ldots, x_n$. We will denote pattern variables with capital E and its variants. Pattern $\lambda x{:}P.\ \pi$ matches a lambda expression of domain type P. It declares a variable x which is local to the pattern, and demand that the body of the matched expression conforms to the pattern π. Bound variables, like x above, are to be distinguished from pattern variables, like $[E\ x_1 \cdots x_n]$. The later provides a placeholder for the matching process; upon execution of a successful matching, it will be bound to a certain expression. The former is just a syntactic constant, which is introduced by a pattern for lambda expressions, and can match only itself. Pattern a matches a name a from the global name context. Pattern $(\pi_1)(\pi_2{:}P)$ matches an application; in order to avoid polymorphic types in patterns, we require that the this pattern proscribes the exact type of the argument in the application.

The judgment for typechecking patterns has the form

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

and reads: in the context of global names S, global parameters Δ, and a context of locally declared variables Γ, the pattern π has the type P, support set *included in C* and produces a residual context Γ_1 of pattern variables and their typings. This residual context is to be passed to subsequent computations. The rules of this judgment are presented in Figure 4. Note that, because we are limited to only the simply-typed fragment, the local variables that the typing rules deposit in Γ will always be ordinary value variables, and always simply typed. On the other hand, we do allow a bit more generality in the case of pattern variables $[E\ x_1 \cdots x_n]$; they still can match only terms of simple types, but these terms can have subterms of more general typing. However, it will always be the case that $\Gamma_1 = \Gamma_1^{\triangle}$ which is easy to show.

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

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

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

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

Observe that the upper-right premise of **case** requires an empty variable context, so that patterns cannot contain outside value

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

$$\frac{S, e_1 \longmapsto S', e_1'}{S, (\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) \longmapsto S', (\mathbf{let\ box}\ u = e_1'\ \mathbf{in}\ e_2)} \qquad \frac{}{S, (\mathbf{let\ box}\ u = \mathbf{box}\ w\ \mathbf{in}\ e_2) \longmapsto S, [w/u]e_2}$$

$$\frac{}{S, \mathbf{fix}\ x{:}A.\ e \longmapsto S, [\mathbf{fix}\ x{:}A.\ e/x]e} \qquad \frac{e \longrightarrow w \qquad e \text{ not contracted}}{S, \mathbf{box}\ e \longmapsto S, \mathbf{box}\ w} \qquad \frac{}{S, (\mathbf{new}\ X{:}P\ \mathbf{in}\ e) \longmapsto (S, X{:}P), e}$$

$$\frac{S, e \longmapsto S', e'}{S, (X\ .\ e) \longmapsto S', (X\ .\ e')} \qquad \frac{S, e \longmapsto S', e'}{S, (e\,@X) \longmapsto S', (e'\,@X)} \qquad \frac{}{S, (Y\ .\ v)\,@X \longmapsto S, (X\ Y)v} \qquad \frac{S, e \longmapsto S', e'}{S, (e\ [\![C]\!]) \longmapsto S', (e'\ [\![C]\!])}$$

$$\frac{}{S, ((\Lambda p\#K.\ e')\ [\![C]\!]) \longmapsto S, [C/p]e'} \qquad \frac{S, e_1 \longmapsto S', e_1'}{S, (\{X \doteq e_1\}\ e_2) \longmapsto S', (\{X \doteq e_1'\}\ e_2)} \qquad \frac{}{S, (\{X \doteq v\}\ e_2) \longmapsto S, \{v/X\}e_2}$$

Figure 3. Structured operational semantics of the core language.

$$\frac{x_i{:}P_i \in \Gamma \qquad C \subseteq \mathbf{dom}(S, \Delta)}{S; \Gamma \Vdash_\Delta [E\ \vec{x}] : P\,[C] \Longrightarrow E{:}\ \mathop{\text{И}}_{X_1:P_1} \ldots \mathop{\text{И}}_{X_n:P_n} \Box(P\,[C, \vec{X}])} \qquad \frac{C \subseteq \mathbf{dom}(S, \Delta)}{S; \Gamma, x{:}P \Vdash_\Delta x : P\,[C] \Longrightarrow \cdot} \qquad \frac{C \subseteq \mathbf{dom}(S, \Delta)}{S, X{:}P; \Gamma \Vdash_\Delta X : P\,[X, C] \Longrightarrow \cdot}$$

$$\frac{S; \Gamma, x{:}P_1 \Vdash_\Delta \pi : P_2\,[C] \Longrightarrow \Gamma_1}{S; \Gamma \Vdash_\Delta \lambda x{:}P_1.\ \pi : P_1 \to P_2\,[C] \Longrightarrow \Gamma_1} \qquad \frac{S; \Gamma \Vdash_\Delta \pi_1 : P_2 \to P\,[C] \Longrightarrow \Gamma_1 \quad S; \Gamma \Vdash_\Delta \pi_2 : P_2\,[C] \Longrightarrow \Gamma_2}{S; \Gamma \Vdash_\Delta (\pi_1)\ (\pi_2 : P_2) : P\,[C] \Longrightarrow \Gamma_1, \Gamma_2}$$

Figure 4. Selected typing rules for patterns.

or expression variables.

The operational semantics for patterns is given through the new judgment

$$S; \Gamma; w \rhd \pi \Longrightarrow S', \Theta$$

which reads: in a global context of names S, global context of parameters Δ, context of local variables Γ, and the support C, the matching of *contracted* expression w to the pattern π extends the global store to S' and generates a substitution Θ for the pattern-variables of π. We present several interesting rules below.

$$\frac{\mathbf{fv}(w) \subseteq \{x_1, \ldots, x_n\} \qquad X_1, \ldots, X_n \text{ fresh} \qquad x_i{:}P_i \in \Gamma}{S; \Delta; \Gamma; w \rhd [E\ \vec{x}] \Longrightarrow (S, X_i{:}P_i), [E \mapsto (\vec{X}\ .\ \mathbf{box}\ [\vec{X}/\vec{x}]w)]}$$

$$\frac{S; \Delta; \Gamma, x{:}P; w \rhd \pi \Longrightarrow S', \Theta}{S; \Delta; \Gamma; \lambda x{:}P.\ w \rhd \lambda x{:}P.\ \pi \Longrightarrow S', \Theta}$$

$$\frac{\begin{array}{c} S; \Gamma; w_1 \rhd \pi_1 \Longrightarrow S_1, \Theta_1 \\ S; \Gamma \vdash w_2 : P_2\,[S] \\ S_1; \Gamma; w_2 \rhd \pi_2 \Longrightarrow S_2, \Theta_2 \end{array}}{S; \Gamma; (w_1\ w_2) \rhd (\pi_1)\ (\pi_2{:}P_2) \Longrightarrow S_2, (\Theta_1 \circ \Theta_2)}$$

As already mentioned, the pattern variable $[E\ x_1 \cdots x_n]$ should match an expression w provided that w depends only on variables $x_1, \ldots, x_n$. Thus, the rule for pattern variables explicitly provides the required check. The residual substitution binds the pattern variable E to a term obtained from w in which the listed variables x_i are substituted by newly generated names X_i and then abstracted. The soundness of the operational semantics for patterns hinges on the following definition and lemma.

Definition 5 (Types for Substitutions)
The judgment $S \vdash_\Delta \Theta : \Gamma$ denotes that Θ is a substitution for the variables in Γ, and that the substituting terms allow occurrences of only the names in S. In other words $S \vdash_\Delta \Theta : \Gamma$ if for every pattern-variable $E{:}A \in \Gamma$ we have $S; \cdot \vdash_\Delta \Theta(E) : A\,[\,]$.

Lemma 6 (Pattern-matching Type Preservation)
If $S; \Gamma_1^\Delta \Vdash_\Delta \pi : P\,[C] \Longrightarrow \Gamma_2$ and $S; \Gamma_1^\Delta \vdash_\Delta w : P\,[C]$ and $S; \Delta; \Gamma_1^\Delta; w \rhd \pi \Longrightarrow S', \Theta$, then $S' \vdash_\Delta \Theta : \Gamma_2$.

The theory already developed for the core languages readily extends to intensional code analysis. In particular, it is easy to establish the new cases arising in the Substitution Principles (Lemma 1), Parametricity of Typing (Lemma 2), and especially in the Progress and Preservation theorem (Theorem 4). The interested reader is referred to the forthcoming report [12] for the details.

Example 2 We can generalize the exponentiation example further: instead of powering only integers, we can power functions too, i.e. have a functional computing $f \mapsto \lambda x.\ (fx)^n$. The functional is passed the code for f, and an integer n, and returns the code for $\lambda x.\ (fx)^n$. The idea is to have this residual code be as

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

-fpow1 (box λy:int. y + 1) 2;
val it = box (λz:int. (λx.x*(x*1)) ((λy.y+1) z))
        : □(int->int)
```

optimized as possible, while still computing the extensionally same result. One possible implementation of this functional in our core language is given above. As a matter of fact, there is at least one other way to obtain the same: we can eliminate the outer beta redex from the above residual code, at the price of duplicating the inner one.

```
fpow2 =
  λf:□(int->int). λn:int.
    new X:int in
      let box f′ = f
          box e = pow′ ⟦X⟧ (box (f′ X)) n
      in
          box (λx:int. {X = x} e)
      end

- fpow2 (box (λy:int. y + 1)) 2;
val it =
  box (λx:int. ((λy.y+1) x) * ((λy.y+1) x) * 1)
  : □(int->int)
```

Neither of the above implementations is quite satisfactory, since, evidently, the residual code in both cases contains unnecessary redices. The reason is that we do not utilize the *intensional* information that the passed argument is actually a boxed *lambda* abstraction, rather than a more general expression of a functional type. Both the shown programs can be encoded in other meta-programming languages such as $\lambda^{\bigcirc}$ and MetaML. In $\lambda^{\Box}$, one has to be content with a rather weaker program that produces even more unnecessary redices. But, in our language extended with intensional code analysis, we could do a bit better. We could test the argument at run-time and output a more optimized code if it is a lambda expression. This way we obtain the most simplified, if not the most efficient residual code.

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

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

Example 3 This example is a (segment) of a function for symbolic differentiation. The function takes a name abstraction as an argument: the body of the abstraction is a boxed term encoding the expression to be differentiated; the abstracted name represents the variable with respect to which the differentiation takes place. When the boxed expression is a sum of two subexpressions, the function just recurses over them. When the boxed expression is a beta-redex (of a limited form), it first reduces it before recursing. Other names and constants are matched in the default case, which thus returns the derivative 0.

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

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

6 Conclusions and future work

This paper presents a typed functional language for meta-programming, employing a novel way to define a modal type of code. The system combines the $\lambda^{\Box}$-calculus [15] with the notion of names based on the developments in FreshML and Nominal Logic [17, 7, 16, 6]. The motivation for combining the two comes from the long-recognized need for meta-programming to handle code expressions with free variables [2, 22, 11]. $\lambda^{\Box}$ 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 code expressions, and also compose, evaluate, inspect and destruct them.

Another way to view the work presented here is as a higher-order extension of the FreshML concept of names. Indeed, in FreshML, types of names are separated from the types of the rest of the language. In this sense, the syntax trees that FreshML can manipulate are first-order. But, if one wants syntax trees of typed syntax (i.e. "higher-order" syntax), then it seems necessary to make a distinction between the meta-level and the object-level (i.e. syntax level) of the language. In other words, one needs a modal type constructor like our □. Not surprisingly then, yet another way to view our contribution is as a generalization of the system presented in [4] for primitive recursion over higher-order abstract syntax.

We list below some extensions of the language which we hope to explore in the future.

Higher-order types for names With the limitation that names can only be simply-typed, our language can encode only object programs with simply-typed free variables. This makes it a two-level, rather than a multi-level language like $\lambda^{\bigcirc}$ and MetaML. It would be interesting to investigate how further generalization of the typing for names, if possible at all, will influence the rest of the language, in particular the operations of name abstraction and concretion.

Type polymorphism and type-polymorphic recursion In a meta-programming language, the typing of object programs is made part of the typing of the meta programs. Consequently, such a language has a lot of types to care for and thus needs strong notions of type polymorphism. This was already evident from our example program for symbolic differentiation in Section 5.

Models Last, but probably most important, we should build models for our type system and put it on a sound logical footing. Interaction between names and modal logic has been of interest to philosophical investigations for quite some time (see [9] and [5]). We hope to draw on this work for the future developments.

7 References

[1] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2001. to appear.

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

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

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

[5] M. Fitting and R. L. Mendelsohn. *First-Order Modal Logic*. Kluwer Academic Publishers, 1999.

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

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

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

[9] S. A. Kripke. *Naming and Necessity*. Harvard University Press, 1980.

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

[11] E. Moggi, W. Taha, Z.-E.-A. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming*, pages 193–207, 1999.

[12] A. Nanevski. Meta-programming with names and necessity. Technical Report CMU-CS-02-123, School of Computer Science, Carnegie Mellon University, April 2002.

[13] M. F. Nielsen. Combining close and open code. Unpublished, 2001.

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

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

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

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

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

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

[20] T. Sheard. Accomplishments and research challenges in metaprogramming. In W. Taha, editor, *SAIG*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.

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

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

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

[24] P. Wickline, P. Lee, F. Pfenning, and R. Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), 1998.