Meta-Programming with Names and Necessity

ALEKSANDAR NANEVSKI and FRANK PFENNING
Carnegie Mellon University, Pittsburgh, PA 15213, USA
(*e-mail:* {aleks,fp}@cs.cmu.edu)

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.

1 Introduction

Meta-programming can be broadly defined as a discipline of algorithmic manipulation of programs written in a certain *object* language, through a program written in another (or *meta*) language. The operations on object programs that the meta program may describe can be very diverse, and may include, among others: generation, inspection, specialization, and, of course, execution of object programs at run-time.

To illustrate the concept we present the following scenario, and refer to (Sheard, 2001) for a more comprehensive treatment. For example, 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 (Lee & Leone, 1996; Wickline *et al.*, 1998b; Wickline *et al.*, 1998a) and the functional programming concept of staged computation (Ershov, 1977; Glück & Jørgensen, 1995; Davies & Pfenning, 2001).

Languages in which object programs can not only be composed and executed but also have their structure inspected add further advantages. In particular, efficiency may benefit from various optimizations that can be performed knowing the structure of the code. For example, (Griewank, 1989) reports 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. There are other applications as well which seem to call for the capability to execute a certain function and also inspect its structure: see (Rozas, 1993) for examples in computer graphics and numerical analysis, and (Ramsey & Pfeffer, 2002) for an example in machine learning and probabilistic modeling.

In this paper, we are concerned with typed functional languages for meta-programming; even more precisely, we limit the considerations to only *homogeneous* meta-programming, which is the especially simple case when the object and the meta language are the same. Recent developments in this direction 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 (Davies & Pfenning, 2001; Pfenning & Davies, 2001). The type $\Box A$ has been used in run-time code generation to classify generators of code of type A (Wickline *et al.*, 1998b; Wickline *et al.*, 1998a). The $\lambda^{\bigcirc}$-calculus is the proof-term language for discrete linear-time 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 a binding-time specification (Davies, 1996). Both calculi provide a distinction between levels (or stages) of terms, and this explains their use in meta-programming. The lowest level is the meta language, which is used to manipulate the terms at the next level (terms of type $\Box A$ in $\lambda^{\Box}$ and $\bigcirc A$ in $\lambda^{\bigcirc}$), which is the meta language for the subsequent level containing another stratum of boxed or 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 operational semantics of $\lambda^{\bigcirc}$ allows reduction under object-level λ-binders, and that is why the the postponed code of $\lambda^{\bigcirc}$ is frequently conflated with the notion of *open code*.

This dichotomy between closed and open code has inspired most of the recent type systems for meta-programming. The abstract concept of open code (not necessarily that of $\lambda^{\bigcirc}$) is more general than closed code. In a specific programming environment (as already observed by (Davies, 1996)), working with open code is more flexible and results in better and more optimized residual object programs. However, we also want to run the generated object programs when they are closed, and thus we need a type system which integrates modal types for both closed and open code.

There have been several proposed type systems providing this expressiveness, most notable being MetaML (Moggi *et al.*, 1999; Taha, 1999; Calcagno *et al.*, 2000; Calcagno *et al.*, 2001). 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.

The approach in our calculus (which we call $\nu^{\Box}$) is opposite. 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 (Nielsen, 2001)). If a given code expression depends on a set of free variables, it 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 (Gabbay & Pitts, 2002; Pitts & Gabbay, 2000; Pitts, 2001; Gabbay, 2000). This design choice leads to a logically motivated and easily extendable type system. For example, we describe in (Nanevski, 2002) an extension with *intensional code analysis* which allows object expressions to be compared for structural equality and destructed via pattern-matching, much in the same way as one would work with any abstract syntax tree.

This paper is organized as follows: Section 2 is a brief exposition of prior 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. We illustrate the type system with example programs, before discussing the related work in Section 5.

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 (Pfenning & Davies, 2001; Davies & Pfenning, 2001). Chronologically, it came to be considered in functional programming in the context of specialization for purposes of run-time code generation (Wickline *et al.*, 1998b; Wickline *et al.*, 1998a). 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; we use b to stand for a predetermined set of base types, and c for constants of those types.

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

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 (Pfenning & Davies, 2001). 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. As Figure 1 shows, 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 wraps 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 enclosed in e_1 and binds it to the variable u to be used in e_2.

The type system of $\lambda^{\Box}$ 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 implicitly assume that exchange holds for both; that is, that the order of variables in the contexts is immaterial.

Figure 2 presents the small-step operational semantics of $\lambda^{\Box}$. We have decided on a call-by-value strategy which, in addition, prohibits reductions on the object level. Thus, if an expression is boxed, its evaluation will be suspended. Boxed expressions themselves are considered values. This choice is by no means canonical, but is necessary for the applications in this paper.

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 not only capable of composing object programs, but can also explicitly force 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*; that is, 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* (Pfenning & Elliott, 1988) as well. The operation of reflection corresponds to the

$$\frac{}{\Delta;(\Gamma,x{:}A)\vdash x:A} \qquad \frac{}{(\Delta,u{:}A);\Gamma\vdash u:A}$$

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

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

Fig. 1. Typing rules for $\lambda^{\Box}$.

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

$$\frac{e_1\longmapsto e_1'}{\mathsf{let\ box}\ u=e_1\ \mathsf{in}\ e_2\longmapsto \mathsf{let\ box}\ u=e_1'\ \mathsf{in}\ e_2}$$

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

Fig. 2. Operational semantics of $\lambda^{\Box}$.

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

If we adhere to the interpretation of object terms as higher-order syntax, then the $\lambda^{\Box}$ staging of `exp3` is rather unsatisfactory. The problem is that the residual object programs produced by `exp3` (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 run-time code generation; indeed, variable-for-variable redexes can easily be eliminated by a compiler. But if object terms are viewed as higher-order syntax (and, as we argued in the previous section, this is a very natural model for the $\lambda^{\Box}$-calculus), the limitation is severe. It exhibits that $\lambda^{\Box}$ is too restrictive to allow for arbitrary composition of higher-order syntax trees. The reason for the deficiency is in the requirement that 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.

Unfortunately, if we only have a type of closed syntactic expressions at our disposal, we can't ever type the *body* of an object-level λ-abstraction in isolation from the 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 5 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 (Davies, 1996). 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 by a type introduction form, like λ-abstraction, 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 (Pitts & Gabbay, 2000; Gabbay & Pitts, 2002), and also (Odersky, 1994). 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

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

which has the type $\Box_{X,Y} int$, 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 has pairs, the term

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

would have the type $int \times \Box_Y int$ 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 non-executable 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. Note 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 a related development described in (Nanevski, 2002)).

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

Names	X	$\in$	$\mathcal{N}$
Support sets	C, D	$\in$	$\mathcal{P}(\mathcal{N})$
Types	A	$::=$	$b \mid A_1 \to A_2 \mid A_1 \nrightarrow A_2 \mid \Box_C A$
Explicit substitutions	Θ	$::=$	$\cdot \mid X \to e, \Theta$
Terms	e	$::=$	$c \mid X \mid x \mid \langle\Theta\rangle u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid$ $\mathsf{box}\ e \mid \mathsf{let\ box}\ u = e_1\ \mathsf{in}\ e_2 \mid$ $\nu X{:}A.\ e \mid \mathsf{choose}\ e$
Value variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Expression variable contexts	Δ	$::=$	$\cdot \mid \Delta, u{:}A[C]$
Name contexts	Σ	$::=$	$\cdot \mid \Sigma, X{:}A$

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 such 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 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 its support set.

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 $\mathsf{let\ box}\ u = e_1\ \mathsf{in}\ e_2$ 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 (X^3 + 3X^2Y + 3XY^2 + Y^3)
  in
      ⟨X -> 1, Y -> 2⟩ u
  end

val it = 27 : int
```

□

The terms $\nu x{:}A.\ e$ and $\mathsf{choose}\ e$ are the introduction and elimination form for the type constructor $A \twoheadrightarrow B$. 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 $A \twoheadrightarrow B$, 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 must enforce a discipline on the use of X in e. An occurrence of X at a certain position in e will be allowed only if the type system can establish that that occurrence of X will not be encountered during evaluation. Such possibilities arise in two ways: if X is eventually substituted away by an explicit substitution,

or if X appears in a computationally irrelevant (i.e., dead-code) part of the term. Needless to say, deciding these questions in a practical language is impossible. Our type system provides a conservative approximation using a fairly simple analysis based on propagation of names encountered during typechecking.

Finally, enlarging an appropriate context by a new variable or a name is subject to the usual variable conventions: the new variables and names are assumed distinct, or are renamed in order not to clash with already existing ones. Terms that differ only in the syntactic representation of their bound variables and names are considered equal. The binding forms in the language are $\lambda x{:}A.\ e$, $\mathsf{let\ box}\ u = e_1\ \mathsf{in}\ e_2$ 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 $\mathrm{fv}(e)$ and $\mathrm{fn}(e)$ 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, $\mathrm{fn}(A[C])$ 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.

```
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 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

$$\Theta : \mathcal{N} \rightarrow \mathit{Terms}$$

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

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

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

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

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 $X \rightarrow e$, relating a name X from $\mathrm{dom}(\Theta)$, 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 the set that represents it when it does not result in ambiguities. Just as customary, we denote by $\mathrm{fv}(\Theta)$ the set of free variables in the terms from $\mathrm{range}(\Theta)$. The set of names appearing either in $\mathrm{dom}(\Theta)$ or $\mathrm{range}(\Theta)$ is denoted by $\mathrm{fn}(\Theta)$.

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. Substitution

application is capture-avoiding.

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

The most important aspect of the above definition is that substitution application does not recursively descend under box. This property is of utmost importance for the soundness of our calculus 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 $X \notin \mathrm{dom}(\Theta_1) \cup \mathrm{dom}(\Theta_2)$, we have that $(\Theta_1 \circ \Theta_2)(X) = X$, and therefore $\mathrm{dom}(\Theta_1 \circ \Theta_2) \subseteq \mathrm{dom}(\Theta_1) \cup \mathrm{dom}(\Theta_2)$. Now, because $\mathrm{dom}(\Theta_1 \circ \Theta_2)$ is finite, the syntactic representation of the composition can easily be computed as the set

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

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

$$
\begin{array}{lcl}
\Theta_1' & = & \{X \to \Theta_1\ (X) \mid X \in \mathrm{dom}(\Theta_1) \setminus \mathrm{dom}(\Theta_2)\} \\
\Theta_2' & = & \{X \to \{\Theta_1\}(\Theta_2\ (X)) \mid X \in \mathrm{dom}(\Theta_2)\}
\end{array}
$$

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 because $\Theta_1 \circ \Theta_2$ requires only applying Θ_1 to the defining terms in Θ_2.

3.3 Type system

The type system of the $\nu^{\Box}$-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 using it again. 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 3. 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 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; that is, they both contain only names already declared in the name context Σ.

The rule for non-empty substitutions 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 $\mathrm{dom}(\Theta)$. In other words, the sub-

Explicit substitutions

$$\frac{C \subseteq D}{\Sigma; \Delta; \Gamma \vdash \langle\,\rangle : [C] \Rightarrow [D]}$$

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

Hypothesis

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

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

λ-calculus

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

Modality

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

Names

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

Fig. 3. Typing rules of the $\nu^{\Box}$-calculus.

stitution can provide definitions for more names or for fewer names than the typing judgment actually expresses. For example, the substitution $\Theta = (X \rightarrow 10, Y \rightarrow 20)$ has domain $\mathrm{dom}(\Theta) = \{X, Y\}$, but it can be given (among others) the typings: $[\,] \Rightarrow [\,]$, $[X] \Rightarrow [\,]$, as well as $[X, Y, Z] \Rightarrow [Z]$. And indeed, Θ does map a term of support $[\,]$ into another term with support $[\,]$, a term of support $[X]$ into a term with support $[\,]$, and a term with support $[X, Y, Z]$ into a term with support $[Z]$.

Hypothesis rules. Because 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; that is, $C \subseteq \mathrm{dom}\,(\Sigma)$. 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. The 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 value 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 $A \twoheadrightarrow B$. 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 $A \twoheadrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction. In other words, the operational semantics of the 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 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 $int \twoheadrightarrow int \twoheadrightarrow \Box int$).

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 a related development on intensional analysis of syntax (Nanevski, 2002), we 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 (Nanevski, 2002) did not use the constructors ν and choose, but rather combined them into a single constructor new. This is also the case in the (Pitts & Gabbay, 2000). The decomposition is given by the equation

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

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 type preservation (Theorem 12) and progress (Theorem 13).

Lemma 4 (*Structural properties of contexts*)

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

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 $C \subseteq C' \subseteq \mathrm{dom}(\Sigma)$ and $D' \subseteq D \subseteq \mathrm{dom}(\Sigma)$ 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 $D \subseteq \mathrm{dom}(\Sigma)$ 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 $\mathrm{dom}(\Theta) \cap \mathrm{dom}(\Theta') = \emptyset$, 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 substitutions 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:

$$\begin{aligned}\Theta'_1 &= \{X \to \Theta(X) \mid X \in \mathrm{dom}(\Theta) \setminus \mathrm{dom}(\Theta')\} \\ \Theta'_2 &= \{X \to \{\Theta\}(\Theta'(X)) \mid X \in \mathrm{dom}(\Theta')\}\end{aligned}$$

and set out to show that

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

These two typings imply the result by the substitution merge lemma (Lemma 7). To establish (a), observe that from the typing of Θ it is clear that $\Theta'_1 : [C \setminus \mathrm{dom}(\Theta')] \Rightarrow [D]$. By definition of $\mathrm{dom}(\Theta')$, if $X \in C_1 \setminus \mathrm{dom}(\Theta')$, then X is fixed by Θ'. Thus, either X does not appear in the syntactic representation of Θ', or the syntactic representation of Θ' contains a sequence of mappings $X \to X_1$, $X_1 \to X_2$, $\ldots$, $X_n \to X$. In the second case, X is the substituting term for X_n, and thus $X \in C$. In the first case, $X \in C$ by inductively appealing to the typing rules for substitutions until the empty substitution is reached. Either way, $C_1 \setminus \mathrm{dom}(\Theta') \subseteq C$, and furthermore $C_1 \setminus \mathrm{dom}(\Theta') \subseteq C \setminus \mathrm{dom}(\Theta')$. Now the result follows by support weakening (Lemma 5.4).

The establish (b) observe that if $X \in \mathrm{dom}(\Theta')$, and $X{:}A \in \Sigma$, then $\Sigma; \Delta; \Gamma \vdash \Theta'(X) : A\,[C]$. By the first induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \{\Theta\}(\Theta'(X)) : A\,[D]$. The typing (b) is now obtained by inductively applying the typing rules for substitutions for each $X \in (C_1 \cap \mathrm{dom}(\Theta'))$. $\square$

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. $\square$

The situation is not that simple 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

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

it up with applications over e of *explicit name substitutions* that were paired up with u. This operation also gives us a control over not only the extensional, but also the intensional form of boxed expressions. 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

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

Note 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 the second induction hypothesis, $\Sigma; \Delta; \Gamma \vdash \langle[\![e_1/u]\!]\Theta\rangle : [C] \Rightarrow [D]$
3. by explicit substitution (Lemma 8.1), $\Sigma; \Delta; \Gamma \vdash \{[\![e_1/u]\!]\Theta\}e_1 : B\,[D]$
4. but this is exactly equal to $[\![e_1/u]\!]e_2$

□

3.5 Operational semantics

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

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

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

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

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

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

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

Fig. 4. Structured operational semantics of $\nu^{\Box}$-calculus.

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 exclusively at 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 4, and the values of the language are generated by the grammar below.

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

The rules are standard, and the only important observation is that the β-redex for the type constructor $\twoheadrightarrow$ extends the run-time context with a fresh name before proceeding. This extension is needed for soundness purposes. Because the freshly introduced name may appear in computationally insignificant positions in the reduct, we must keep the name and its typing in the run-time context.

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 Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash e' : A\,[\,]$.

Proof
By a straightforward induction on the structure of e using the substitution principles. $\square$

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', such that $\Sigma, e \longmapsto \Sigma', e'$.

Proof

By a straightforward induction on the structure of e. $\square$

The progress theorem does not indicate that the reduct e' and the context Σ' are unique for each given e and Σ. In fact, they are not, as fresh names may be introduced during the course of the computation, and two different evaluations of one and the same term may choose the fresh names differently. The determinacy theorem below shows that the choice of fresh names accounts for all the differences between two reductions of the same term. As customary, we denote by $\longmapsto^n$ the n-step reduction relation.

Theorem 14 (*Determinacy*)
If $\Sigma, e \longmapsto^n \Sigma_1, e_1$, and $\Sigma, e \longmapsto^n \Sigma_2, e_2$, then there exists a permutation of names $\pi : \mathcal{N} \to \mathcal{N}$, fixing $\mathrm{dom}(\Sigma)$, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof
By induction on the length of the reductions, using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$ and π is a permutation on names, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$. The only interesting case is when $n = 1$ and $e = \mathsf{choose}\ (\nu X{:}A.\ e')$. In that case, 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. $\square$

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 (Girard, 1986; Reynolds, 1983).

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

Support variables	p, q	$\in$	$\mathcal{S}$
Support sets	C, D	$\in$	$\mathcal{P}(\mathcal{N} \cup \mathcal{S})$
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	$::=$	$\ldots \mid \Lambda p.\ e$

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

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 $\mathrm{fn}(-)$ 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.

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

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; that is, $D \subseteq \mathrm{dom}(\Sigma)$. 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 $D \subseteq \mathrm{dom}(\Sigma_1)$ 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 $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C_2]$, then $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \langle\Theta'\rangle : [C_1'] \Rightarrow [C_2']$

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

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

1. by derivation, $\Sigma; \Delta; \Gamma \vdash e : A\,[C_2]$ and $\Sigma; \Delta; \Gamma \vdash \Theta_1 : [C_1 \setminus \{X\}] \Rightarrow [C_2]$
2. by first induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e' : A'\,[C_2']$
3. by second induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : [(C_1 \setminus \{X\})'] \Rightarrow [C_2']$
4. because $(C_1' \setminus \{X\}) \subseteq (C_1 \setminus \{X\})'$, by support weakening (Lemma 5.4), $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : [C_1' \setminus \{X\}] \Rightarrow [C_2']$

5. result follows from (2) and (4) by the typing rule for non-empty substitutions

□

The structural properties presented in Section 3.4 readily extend to the new language with support polymorphism. The same is true of the type preservation (Theorem 12) and progress (Theorem 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 following program, we use `[p]` in the function definition as a concrete syntax for Λ-abstraction of a support variable `p`.

```
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)
```

□

Example 4 As an example of a more realistic program we present the regular expression matcher from (Davies & Pfenning, 2001) and (Davies, 1996). The example assumes the declaration of the datatype of regular expressions:

```
datatype regexp =
      Empty
    | Plus of regexp * regexp
    | Times of regexp * regexp
    | Star of regexp
    | Const of char
```

```
(*
 * val acc1 : regexp -> (char list -> bool) ->
 *               char list -> bool
 *)

fun acc1 (Empty) k s = k s

  | acc1 (Plus (e1, e2)) k s =
      (acc1 e1 k s) orelse (acc1 e2 k s)

  | acc1 (Times (e1, e2)) k s =
      (acc1 e1 (acc1 e2 k)) s

  | acc1 (Star e) k s =
      (k s) orelse
        acc1 e (λs' =>
                  if s = s' then false
                  else acc1 (Star e) k s')

  | acc1 (Const c) k s =
     case s
      of nil => false
       | (x::l) =>
           ((x = c) andalso (k s))


(*
 * val accept1 : regexp -> char list -> bool
 *)

fun accept1 e s = acc1 e null s
```

Fig. 5. Unstaged regular expression matcher.

We also assume a primitive predicate `null : char list -> bool` testing if the input string is empty. Figure 5 presents an ordinary ML implementation of the matcher, and $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ versions can be found in (Davies & Pfenning, 2001; Davies, 1996).

We would now like to use the $\nu^{\Box}$-calculus to stage the program from Figure 5 so that it can be *specialized* with respect to a given regular expression. For that purpose, it is useful to view the helper function `acc` (called `acc1` in Figure 5) as a code generator. It takes a regular expression e and emits code for parsing according to e, and at the end, it appends k to the generated code. This is the main idea behind the program in Figure 6. Here, for simplicity, we use the name S for the input string to be parsed by the code that `acc` generates. We also want to allow the continuation code k to contain further names standing for yet unbound variables, and hence the support-polymorphic typing `acc : regexp -> ∀p.(`$\Box_{S,p}$`bool -> `$\Box_{S,p}$`bool)`. The support polymorphism pays off when generating code for alternation `Plus(`e_1`, `e_2`)` and iteration `Star(`e`)`. Indeed, observe in the alternation case that the generated code does not duplicate the continuation k. Rather, k is emitted as a separate function which is a joining point for the computation branches corresponding to e_1 and e_2. Similarly, in the case of iteration, we set up a loop in the output code that would attempt zero or more matchings against e. The support polymorphism of `acc` enables us to produce code in chunks without knowing the exact identity of the above-mentioned joining or looping points. Once all the parts of the output code are generated, we just stitch them together by means of explicit substitutions.

```
(*
 * val accept : regexp ->
 *    □(char list -> bool)
 *)

fun accept (e : regexp) =
  choose νS : char list.

  (*
   * acc : regexp -> ∀p.(□_{S,p}bool
   *                      -> □_{S,p}bool)
   *)

  let fun acc (Empty) [p] k = k

        | acc (Plus (e1, e2)) [p] k =
           choose νJOIN : char list
                                -> bool.
           let box u1 =
                acc e1 [JOIN] box(JOIN S)
               box u2 =
                acc e2 [JOIN] box(JOIN S)
               box kk = k
           in
               box(let fun join t =
                         <S->t>kk
                   in
                       <JOIN->join>u1
                         orelse
                       <JOIN->join>u2
                   end)
           end

        | acc (Times (e1, e2)) [p] k =
           acc e1 (acc e2 k)

        | acc (Star e) [p] k =
           choose νT : char list
           choose νLOOP : char list
                                -> bool.
           let box u =
                acc e [T, LOOP]
                  box(if T = S then false
                      else LOOP S)
               box kk = k
           in
               box(let fun loop t =
                     <S->t>kk
                      orelse
                     <LOOP->loop,
                       T->t,S->t>u
                   in
                      loop S
                   end)
           end

        | acc (Const c) [p] k =
           let box cc = lift c
               box kk = k
           in
               box(case S
                    of (x::xs) =>
                         (x = cc) andalso
                         <S->xs>kk
                     | nil => false)
           end

    box code = acc e [] box (null S)
in
    box (λs:char list. <S->s>code)
end
```

Fig. 6. Regular expression matcher staged in the $\nu^{\Box}$-calculus.

At this point, it may be illustrative to trace the execution of the program on a concrete input. Figure 7 presents the function calls and the intermediate results that occur when the $\nu^{\Box}$-staged matcher is applied to the regular expression `Star(Empty)`. Note that the resulting specialized program does not contain variable-for-variable redexes, but it does perform unnecessary boolean tests. It is possible to improve the matching algorithm to avoid emitting this extraneous code. The improvement involves a further examination and preprocessing of the input regular expression, but the thorough description is beyond the scope of this paper. We refer to (Harper, 1999) for an insightful analysis. □

5 Related work

The work presented in this paper lies in the intersection of several related areas: staged computation and partial evaluation, run-time code generation, meta-programming, modal logic and higher-order abstract syntax.

An early reference to staged computation is (Ershov, 1977) which introduces

```
▷ accept (Star (Empty))

    ▷ acc (Star(Empty)) [] (box (null S))

        ▷ acc Empty [T, LOOP] (box (if T = S then false
                                    else LOOP S))

        ◁ box (if T = S then false else LOOP S)

    ◁ box (let fun loop (t) =
                 null (t) orelse
                  if t = t then false else loop(t)
           in
                 loop S
           end)

◁ box (λs. let fun loop (t) =
                   null (t) orelse
                    if t = t then false else loop(t)
           in
               loop s
           end)
```

Fig. 7. Example execution trace for a regular expression matcher in $\nu^{\Box}$. Function calls are marked by $\triangleright$ and the corresponding return results are marked by an aligned $\triangleleft$.

staged computation under the name of "generating extensions". Generating extensions for purposes of partial evaluation were also foreseen by (Futamura, 1971), and the concept is later explored and eventually expanded into multi-level generating extensions by (Jones *et al.*, 1985; Glück & Jørgensen, 1995; Glück & Jørgensen, 1997). Most of this work is done in an untyped setting.

The typed calculus that provided the direct motivation and foundation for our system is the $\lambda^{\Box}$-calculus. It evolved as a type theoretic explanation of staged computation (Davies & Pfenning, 2001; Wickline *et al.*, 1998a), and run-time code-generation (Lee & Leone, 1996; Wickline *et al.*, 1998b), and we described it in Section 2.

Another important typed calculus for meta-programming is $\lambda^{\bigcirc}$. Formulated by (Davies, 1996), 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 (Moggi *et al.*, 1999; Taha, 1999; Taha, 2000).

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 (Calcagno *et al.*, 2000; Calcagno *et al.*, 2001). 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.

Another interesting calculus for meta-programming is Nielsen's $\lambda^{[]}$ described in (Nielsen, 2001). 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. This property makes it unclear if $\lambda^{[]}$ can be extended with a concept corresponding to our support polymorphism.

Nielsen and Taha present another system for combining closed and open code in (Nielsen & Taha, 2003). 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 variables declared in previous visits can be reused. This feature provides the functionality of open code. The environment classifiers are related to our support variables in several respects: they both are bound by universal quantifiers and they both abstract over sets. Indeed, our support polymorphism explicitly abstracts over sets of names, while environment classifiers are used to name parts of the variable context, and thus implicitly abstract over sets of variables.

Coming from the direction of higher-order abstract syntax, probably the first work pointing to the importance of a non-parametric binder like our ν-abstraction is (Miller, 1990). The connection of higher-order abstract syntax to modal logic has been recognized by Despeyroux, Pfenning and Schürmann in the system presented in (Despeyroux *et al.*, 1997), which was later simplified into a two-level system in Schürmann's dissertation (Schürmann, 2000). There is also (Hofmann, 1999) which discusses various presheaf models for higher-order abstract syntax, then (Fiore *et al.*, 1999) which explores untyped abstract syntax in a categorical setup, and an extension to arbitrary types (Fiore, 2002).

However, the work that explicitly motivated our developments is the series of papers on Nominal Logic and FreshML (Gabbay & Pitts, 2002; Pitts & Gabbay, 2000; Pitts, 2001; Gabbay, 2000). 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 (Nanevski, 2002) 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.

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 $\mathsf{new}\ X\ \mathsf{in}\ e$, which can roughly be interpreted in $\nu^{\Box}$-calculus as

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

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 & Davies, 2001) 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, 1963) and (Smoryński, 1985). Viewpoints of (Attardi & Simi, 1995) and contexts of (McCarthy, 1993) are similar to our notion of support, and are used to express relativized truth. Finally, the names from $\nu^{\Box}$ resemble non-rigid designators of (Fitting & Mendelsohn, 1999), names of (Kripke, 1980), and virtual individuals of (Scott, 1970), and also touch on the issues of existence and identity explored in (Scott, 1979). 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.

6 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 (Pfenning & Davies, 2001) with the notion of names inspired by developments in FreshML and Nominal Logic (Pitts & Gabbay, 2000; Gabbay & Pitts, 2002; Pitts, 2001; Gabbay, 2000). The motivation for combining $\lambda^{\Box}$ with names comes from the long-recognized

need of meta-programming to handle object programs with free variables (Davies, 1996; Taha, 1999; Moggi *et al.*, 1999). 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 provide 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.

The $\nu^{\Box}$-calculus is a reformulation of the calculus presented in (Nanevski, 2002). Some of the adopted changes involve simplification of the operational semantics and the constructs for handling names. Furthermore, we decomposed the name introduction form new into two constructors ν and choose which are now introduction and elimination form for a new type constructor $A \nrightarrow B$. This design choice 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 (Pfenning, 2001) and (Awodey & Bauer, 2001) to classify terms of unknown support.

Another important direction for exploration concerns the implementation of $\nu^{\Box}$. The calculus presented in this paper was developed with a particular semantical interpretation in mind of object level expressions as abstract syntax trees representing templates for source programs. But this need not be the only interpretation. It is quite possible that boxed expressions of $\nu^{\Box}$-calculus with support polymorphism can be stored at run-time in some intermediate or even compiled form, which might benefit the efficiency of programs. It remains an important future work to explore these implementation issues.

7 Acknowledgment

We would like to thank Dana Scott, Bob Harper, Peter Lee and Andrew Pitts for their helpful comments on the earlier versions of the paper and Robert Glück for pointing out some missing references.

References

Attardi, Giuseppe, & Simi, Maria. (1995). A formalization of viewpoints. *Fundamenta informaticae*, **23**(3), 149–173.

Awodey, Steve, & Bauer, Andrej. (2001). *Propositions as [Types].* Tech. rept. IML-R–34-00/01–SE. Institut Mittag-Leffler, The Royal Swedish Academy of Sciences.

Calcagno, Cristiano, Moggi, Eugenio, & Taha, Walid. (2000). Closed types as a simple approach to safe imperative multi-stage programming. *Pages 25–36 of:* Montanari, Ugo,

Rolim, José D. P., & Welzl, Emo (eds), *Automata, languages and programming*. Lecture Notes in Computer Science, vol. 1853. Springer.

Calcagno, Cristiano, Moggi, Eugenio, & Sheard, Tim. (2001). Closed types for a safe imperative MetaML. *Journal of functional programming*. To appear.

Davies, Rowan. (1996). A temporal logic approach to binding-time analysis. *Pages 184–195 of: Symposium on Logic in Computer Science, LICS'96*.

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

Despeyroux, Joëlle, Pfenning, Frank, & Schürmann, Carsten. (1997). Primitive recursion for higher-order abstract syntax. *Pages 147–163 of:* de Groote, Philippe, & Hindley, J. Roger (eds), *Typed lambda calculi and applications*. Lecture Notes in Computer Science, vol. 1210. Springer.

Ershov, A. P. (1977). On the partial computation principle. *Information processing letters*, **6**(2), 38–41.

Fiore, Marcelo. (2002). Semantic analysis of normalization by evaluation for typed lambda calculus. *Pages 26–37 of: International Conference on Principles and Practice of Declarative Programming, PPDP'02*.

Fiore, Marcelo, Plotkin, Gordon, & Turi, Daniele. (1999). Abstract syntax and variable binding. *Pages 193–202 of: Symposium on Logic in Computer Science, LICS'99*.

Fitting, Melvin, & Mendelsohn, Richard L. (1999). *First-order modal logic*. Kluwer.

Futamura, Yoshihiko. (1971). Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, computers, controls*, **2**(5), 45–50.

Gabbay, Murdoch J. 2000 (August). *A theory of inductive definitions with α-equivalence*. Ph.D. thesis, Cambridge University.

Gabbay, Murdoch J., & Pitts, Andrew M. (2002). A new approach to abstract syntax with variable binding. *Formal aspects of computing*, **13**, 341–363.

Girard, Jean-Yves. (1986). The system F of variable types, fifteen years later. *Theoretical computer science*, **45**(2), 159–192.

Glück, Robert, & Jørgensen, Jesper. (1995). Efficient multi-level generating extensions for program specialization. *Pages 259–278 of:* Hermenegildo, Manuel, & Swierstra, S. Doaitse (eds), *Programming languages: Implementations, logics and programs*. Lecture Notes in Computer Science, vol. 982. Springer.

Glück, Robert, & Jørgensen, Jesper. (1997). An automatic program generator for multi-level specialization. *Lisp and symbolic computation*, **10**(2), 113–158.

Griewank, Andreas. (1989). On automatic differentiation. *Pages 83–108 of:* Iri, Masao, & Tanabe, Kunio (eds), *Mathematical programming: Recent developments and applications*. Kluwer.

Harper, Robert. (1999). Proof-directed debugging. *Journal of functional programming*, **9**(4), 463–470.

Hofmann, Martin. (1999). Semantical analysis of higher-order abstract syntax. *Pages 204–213 of: Symposium on Logic in Computer Science, LICS'99*.

Jones, Neil D., Sestoft, Peter, & Søndergaard, Harald. (1985). An experiment in partial evaluation: the generation of a compiler generator. *Pages 124–140 of:* Jouannaud, Jean-Pierre (ed), *Rewriting techniques and applications*. Lecture Notes in Computer Science, vol. 202. Springer.

Kripke, Saul A. (1980). *Naming and necessity*. Harvard University Press.

Lee, Peter, & Leone, Mark. (1996). Optimizing ML with run-time code generation. *Pages 137–148 of: Conference on Programming Language Design and Implementation, PLDI'96*.

McCarthy, John. (1993). Notes on formalizing context. *Pages 555–560 of: International Joint Conference on Artificial Intelligence, IJCAI'93.*

Miller, Dale. (1990). An extension to ML to handle bound variables in data structures. *Pages 323–335 of: Proceedings of the first esprit BRA workshop on logical frameworks.*

Moggi, Eugenio, Taha, Walid, Benaissa, Zine-El-Abidine, & Sheard, Tim. (1999). An idealized MetaML: Simpler, and more expressive. *Pages 193–207 of: European Symposium on Programming, ESOP'99.*

Montague, Richard. (1963). Syntactical treatment of modalities, with corollaries on reflexion principles and finite axiomatizability. *Acta philosophica fennica*, **16**, 153–167.

Nanevski, Aleksandar. (2002). Meta-programming with names and necessity. *Pages 206–217 of: International Conference on Functional Programming, ICFP'02.* A significant revision is available as a technical report CMU-CS-02-123R, Computer Science Department, Carnegie Mellon University.

Nielsen, Michael Florentin. (2001). *Combining closed and open code.* Unpublished.

Nielsen, Michael Florentin, & Taha, Walid. (2003). Environment classifiers. *Pages 26–37 of: Symposium on Principles of Programming Languages, POPL'03.*

Odersky, Martin. (1994). A functional theory of local names. *Pages 48–59 of: Symposium on Principles of Programming Languages, POPL'94.*

Pfenning, Frank. (2001). Intensionality, extensionality, and proof irrelevance in modal type theory. *Pages 221–230 of: Symposium on Logic in Computer Science, LICS'01.*

Pfenning, Frank, & Davies, Rowan. (2001). A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, **11**(4), 511–540.

Pfenning, Frank, & Elliott, Conal. (1988). Higher-order abstract syntax. *Pages 199–208 of: Conference on Programming Language Design and Implementation, PLDI'88.*

Pitts, Andrew M. (2001). Nominal logic: A first order theory of names and binding. *Pages 219–242 of:* Kobayashi, Naoki, & Pierce, Benjamin C. (eds), *Theoretical aspects of computer software.* Lecture Notes in Computer Science, vol. 2215. Springer.

Pitts, Andrew M., & Gabbay, Murdoch J. (2000). A metalanguage for programming with bound names modulo renaming. *Pages 230–255 of:* Backhouse, Roland, & Oliveira, José Nuno (eds), *Mathematics of program construction.* Lecture Notes in Computer Science, vol. 1837. Springer.

Ramsey, Norman, & Pfeffer, Avi. (2002). Stochastic lambda calculus and monads of probability distributions. *Pages 154–165 of: Symposium on Principles of Programming Languages, POPL'02.*

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of:* Mason, R. E. A. (ed), *Information processing '83.* Elsevier.

Rozas, Guillermo J. (1993). *Translucent procedures, abstraction without opacity.* Tech. rept. AITR-1427. Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Schürmann, Carsten. (2000). *Automating the meta-theory of deductive systems.* Ph.D. thesis, Carnegie Mellon University.

Scott, Dana. (1970). Advice on modal logic. *Pages 143–173 of:* Lambert, Karel (ed), *Philosophical problems in logic.* Dordrecht: Reidel.

Scott, Dana. (1979). Identity and existence in intuitionistic logic. *Pages 660–696 of:* Fourman, Michael, Mulvey, Chris, & Scott, Dana (eds), *Applications of sheaves.* Lecture Notes in Mathematics, vol. 753. Springer.

Sheard, Tim. (2001). Accomplishments and research challenges in meta-programming. *Pages 2–44 of:* Taha, Walid (ed), *Semantics, applications, and implementation of program generation.* Lecture Notes in Computer Science, vol. 2196. Springer.

Smoryński, C. (1985). *Self-reference and modal logic*. Springer.

Taha, Walid. (1999). *Multi-stage programming: Its theory and applications*. Ph.D. thesis, Oregon Graduate Institute of Science and Technology.

Taha, Walid. (2000). A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival. *Pages 34–43 of: Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'00*.

Wickline, Philip, Lee, Peter, Pfenning, Frank, & Davies, Rowan. (1998a). Modal types as staging specifications for run-time code generation. *ACM computing surveys*, **30**(3es).

Wickline, Philip, Lee, Peter, & Pfenning, Frank. (1998b). Run-time code generation and Modal-ML. *Pages 224–235 of: Conference on Programming Language Design and Implementation, PLDI'98*.