JFP **15** (6): 893–939, 2005. © 2005 Cambridge University Press
doi:10.1017/S095679680500568X Printed in the United Kingdom

Staged computation with names and necessity

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

Abstract

Staging is a programming technique for dividing the computation in order to exploit the early availability of some arguments. In the early stages the program uses the available arguments to generate, at run time, the code for the late stages. A type system for staging should ensure that only well-typed expressions are generated, and that only expressions with no free variables are permitted for evaluation. In this paper, we present a calculus for staged computation in which code from the late stages is composed by splicing smaller code fragments into a larger context, possibly incurring capture of free variables. The type system ensures safety by tracking the names of free variables for each code fragment. The type system is based on the necessity operator $\Box$ from constructive modal logic, which we index with a set of names C. Our type $\Box_C A$ classifies expressions of type A that belong to the late stage, and whose free names are in the set C.

1 Introduction

Staging is a programming technique for explicitly dividing a computation in order to exploit the early availability of some arguments (Ershov, 1977; Jones *et al.*, 1985; Nielson & Nielson, 1988; Glück & Jørgensen, 1995; Davies & Pfenning, 2001). For example, a typical test used in many convex hull algorithms filters a set of points to see on which side of a line given by two points they lie. This test can be staged by first forming the line and its normal, and then checking the position of each point from the set. This way, a staged test obviates the need to repeat the part of the computation pertinent to the normal whenever a new point is tested, and can potentially save a lot of work.

Because it is often quite cumbersome to design programs that fully exploit the natural stage separation of their arguments, it is very desirable for a programming language to provide support for early detection and reporting of staging errors. Two calculi, $\lambda^{\Box}$ and $\lambda^{\bigcirc}$, based on different modal logics, have emerged in the literature as alternatives suitable for capturing different invariants of staged computations.

The $\lambda^{\Box}$-calculus is the proof term calculus for a constructive version of modal logic S4, whose necessity constructor $\Box$ annotates *valid* propositions (Davies & Pfenning, 2001; Pfenning & Davies, 2001). The $\lambda^{\bigcirc}$-calculus is the proof term calculus for discrete linear-time temporal logic, and the type constructor $\bigcirc$ annotates propositions that are true at the subsequent time moment (Davies, 1996).

From the computational perspective, values of the modal type $\Box A$ are *closed expressions* of type A (Davies & Pfenning, 2001). Closed expressions are independent of the context in which they are built, and can be used in any other context. In our example, the expression for computing the normal of the line could be assigned a modal type, as it is independent of the points that it will be testing. The computational meaning of $\bigcirc A$ is a little bit more subtle. Values of type $\bigcirc A$ are also expressions of type A to be evaluated at the next stage, but they may refer to the variables from their environment; they classify expressions that may be *open*, that is, contain free variables subject to some further requirements (Davies, 1996).

Each of the calculi has its advantages and drawbacks. Because values of type $\Box A$ are closed expressions, they can safely be evaluated to obtain a value of type A. But the ability to evaluate comes with a price: composing closed expressions into larger ones, while maintaining the closeness invariant, is cumbersome and produces unnecessarily complex results. On the other hand, a value of type $\bigcirc A$ is an open expression. It behaves nicely under composition, but the type system of $\lambda^{\bigcirc}$ does not provide guarantees for its safe evaluation. The reason is exactly in the openness: it is not sound to evaluate an open expression before all of its free variables are bound.

The desire to combine the advantages of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ has inspired most of the recent work on type systems for staged computation, most notable being MetaML (Taha & Sheard, 1997; Moggi *et al*., 1999; Taha, 1999; Benaissa *et al*., 1999; Calcagno *et al*., 2003a; Sheard, 2001), and its recent variant MetaOCaml (Calcagno *et al*., 2003b; Taha & Nielsen, 2003). The modal type of MetaML and MetaOCaml is that of open expressions from $\lambda^{\bigcirc}$. In addition, various versions of MetaML contain some additional type constructor to classify as closed those expressions that could be proved to contain no free variables. Also, both MetaML and MetaOCaml feature a term constructor for explicit evaluation of closed expressions.

The approach of our calculus (which we call $\nu^{\Box}$) is opposite. Rather than refining the notion of open expressions of $\lambda^{\bigcirc}$, we relax the notion of closed expressions of $\lambda^{\Box}$. We start with the system of $\lambda^{\Box}$, but allow generated code to contain free variables, as long as this is recorded in the types. The free variables of modal expressions are represented by a separate semantic category of names, the treatment of which is inspired by the work on Nominal Logic and FreshML (Gabbay & Pitts, 2002; Pitts & Gabbay, 2000; Pitts, 2001; Gabbay, 2000).

This approach leads to a logically motivated type system, in which one can encode open expressions and evaluate closed ones. The approach is conceptually simpler than that of MetaML, in the sense that only one type constructor for expressions suffices. In this respect, our system is closer to MetaOCaml, which also features only one type constructor for expressions. However, unlike both MetaML and MetaOCaml, we do not require any additional constructs for explicit evaluation of closed code; this operation can be expressed using the already present constructors which are justified by logical considerations.

The rest of the paper is organized as follows. Section 2 is a brief summary of the previous work on $\lambda^{\Box}$. The type system of $\nu^{\Box}$ and its properties are described in section 3, while section 4 describes parametric polymorphism in sets of names. The equational properties of $\nu^{\Box}$, both with intensional and extensional interpretation of the $\Box$ type, are explored in section 5, before we discuss the related work in section 6.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

2 Modal $\lambda^{\Box}$-calculus

This section reviews the previous work on the modal $\lambda^{\Box}$-calculus, and the way $\lambda^{\Box}$ can be used to divide the computation into stages that specify the relative evaluation order of subcomputations.

The $\lambda^{\Box}$-calculus is the proof-term calculus for the necessitation fragment of the 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 a 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. From the operational standpoint, however, no actual work is done, as the parameter x must be supplied in order to proceed with the 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* a specialized function for computing n-th powers. This suggests a stage distinction between the terms of the calculus. The terms at the late stage are to be viewed as data – as results of a process of code generation. Because the terms at this stage are treated as data, i.e. as objects, we refer to this stage as *object stage*. In the `exp2` function, such terms are `(λx:int.1)` and `(λx:int.x * u(x))`. The early stage (also called the run-time, or meta stage) describes the specific operations to be performed over the data from the object stage. The stages are kept separate; variables from the meta stage (n in the above example) are not allowed to appear in the object stage. This is why the above-illustrated programming style is referred to as *staged computation.*

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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 the object-level subterms. Then the type system can check whether the written code conforms to the staging specifications, turning 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	$::=$	$c \mid x \mid u \mid \lambda x{:}A.\, e \mid e_1\ e_2 \mid$
			box $e \mid$ **let box** $u = e_1$ **in** e_2
Value variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Expression variable contexts	Δ	$::=$	$\cdot \mid \Delta, u{:}A$
Values	v	$::=$	$c \mid \lambda x{:}A.\, e \mid$ **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 the necessitation modifier on propositions (Pfenning & Davies, 2001). In our metaprogramming application, it is used to classify object-level terms. Its introduction and elimination forms are the term constructors box and let box, respectively. As Fig. 1 shows, if e is an object term of type A, then **box** e is 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, the order of variables in the contexts is immaterial. Observe that the typing rule for box removes the variable context Γ. This implements a characteristic restriction of type systems for staged languages that variables from the early stage are not allowed to appear in the later stage.

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 in the generated code until it is run. In the expression **box** e, the evaluation of the object expression e is suspended, but we may still substitute for the object variables of e. In a *value* of modal type, all the object variables are substituted away; such values consist of boxed object expressions that are *closed*, i.e. they *do not contain any free variables.*

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
```

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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 modal 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 how closed object expressions can be *reflected*, i.e. coerced from the object level into the meta level. The opposite coercion, referred to as *reification*, is achieved by the box operator for closed expressions, but cannot be written as a function. This suggests that $\lambda^{\Box}$ could be given even a more specific model in which reflection naturally exists, but reification does not. A possible interpretation exhibiting this behavior considers object-level expressions as generated *source* code, i.e., actual closed *syntactic* expressions, or abstract syntax trees of closed $\lambda^{\Box}$-terms. In contrast, the meta-level expressions are compiled executables. The operation of reflection corresponds to the natural process of compiling a source program 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. Furthermore, the typing of $\lambda^{\Box}$ ensures that only *well-typed* syntactic expressions can be represented in the calculus.

The above intuitive "syntactic" model makes the $\lambda^{\Box}$-calculus very appropriate not only for staged computation, but also for *metaprogramming*. In metaprogramming, expressions are again stratified into stages, but this time the *syntactic structure* of object expressions may be *inspected and analyzed*. In metaprogramming, object expressions represent source code which can be compared for syntactic equality and even pattern-matched against (Nanevski, 2002).

In the remainder of this paper, we will frequently rely on the described syntactic nature of object expressions in order to supply the intuition behind formal developments. However, whether a practical implementation actually needs to represent object expression as syntax will depend on the application. In staged computation, for example, we are usually not interested in inspecting the structure of generated programs, so the generated programs may be represented in some intermediate, or even fully compiled form. At this point, we do not commit to any particular implementation strategy, but instead focus on the theoretical properties of the type system.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

3 Modal calculus of names

3.1 Motivation, syntax and overview

If we adhere to the interpretation of object expressions as generated *source* code, then the $\lambda^{\Box}$ staging of `exp3` is rather unsatisfactory. The problem is that the object expressions generated by `exp3` (e.g. `sqbox`), contain unnecessary variable-for-variable redexes.

From the standpoint of syntax manipulation, $\lambda^{\Box}$ is too restrictive. It cannot express this rather simple way in which well-typed syntactic expressions can be put together to form a more complex syntactic expression.

The reason for the deficiency lies in the requirement that the syntactic object expressions that $\lambda^{\Box}$ can represent and manipulate must always be *closed*. Furthermore, if we only have a type of closed syntactic expressions at our disposal, we can never 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.

What we need in order to avoid the problem of superfluous redexes, but also in order to support code inspection, is the ability to represent *open* expressions and specify *substitution with capture*. This need has long been recognized in the staged computation and metaprogramming community, and Section 6 discusses several different type systems and their solution of the problem. The basic idea of these type systems is most concisely captured by Davies' $\lambda^{\bigcirc}$-calculus (Davies, 1996). The type

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

constructor $\bigcirc$ of this calculus corresponds to the discrete temporal logic modality for propositions true at the subsequent time moment. In a metaprogramming interpretation, the modal type $\bigcirc A$ stands for open object expression of type A, where the free variables of the object expression are modeled by λ-bound variables from the subsequent time moment.

Our $\nu^{\Box}$-calculus adopts a different approach to the problem of spurious redexes. 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). The idea is to employ names to stand for the free variables of object expressions, and correspondingly, to employ explicit name substitutions to facilitate capture of free variables. Intuitively, the expressions of the $\nu^{\Box}$-calculus are obtained by freely adjoining names to the expressions of the $\lambda^{\Box}$-calculus. The situation is somewhat analogous to that in polynomial algebra, where one is given a base algebraic structure A and a set of indeterminates (or generators) $\{X_1, \ldots, X_n\}$, which are then freely adjoined to A into a structure of polynomials $A[X_1, \ldots, X_n]$. In our setup, the indeterminates are the names, and we build "polynomials" over the base structure of $\lambda^{\Box}$ expressions.

When an object expression e contains a name X, we will say that e *depends* on X, or that X is in the *support* of e. 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

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

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

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

would have the type $int \times \Box_Y\, int$ with support $\{X\}$.

As illustrated by the above examples, if an object expression depends on some names, then it is only partially specified. Such partially specified expressions cannot be evaluated before every name in the expression's support is provided a definition. We use explicit substitutions for this purpose. Explicit substitutions remove substituted names from the support, eventually turning non-executable expressions into executable ones.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Example 1 Assuming that X and Y are names of type `int`, the $\nu^{\Box}$ segment below creates a "polynomial" expression 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
```

Notice how the explicit substitution $\langle X \to 1, Y \to 2\rangle$ *captures* the names X and Y in the expression $X^3 + 3X^2Y + 3XY^2 + Y^3$, when this expression is substituted for u. □

We next present the syntax of the $\nu^{\Box}$-calculus and discuss each of the constructors. We use capital letters like X, Y and variants to denote names (of which there should be infinitely many) and C and D for finite sets of names.

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	::=	$c \mid X \mid x \mid \langle\Theta\rangle u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid$ **box** $e \mid$ **let box** $u = e_1$ **in** $e_2 \mid$ $\nu X{:}A.\ e \mid$ **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 may be interpreted as the level of executable code and the level of source code, 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, the values of the type $\Box_C A$ are the closed syntactic expressions containing the names from the set C. We refer to the finite set C as a *support* of such expressions. All the names are drawn from a countably infinite universe of names $\mathcal{N}$. In addition to modal and functional types, $\nu^{\Box}$ features a new type $A \nrightarrow B$ whose meaning we explain below.

As before, the distinction in levels motivates a split in the variable contexts. We have a context Γ for ordinary variables (we will also call them value variables), and a context Δ for modal variables (which we also call expression variables). The context Δ must keep track not only of the typing of a given variable, but also of its support. In $\nu^{\Box}$ we also introduce the context Σ which associates types to names.

The types of $\nu^{\Box}$-calculus are dependent on names, so we impose on our contexts the usual well-formedness conditions from dependently typed calculi. 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

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

type in Σ uses only names declared to the left of it. Two contexts are considered equal if they only differ up to a dependency-preserving reordering of their variables or names.

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 we use in the following sections are well-formed. The same holds for all the types and supports.

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 at the level of terms. Second, it is required that all the references to modal variables that a certain term makes are always prefixed by some explicit substitution. For example, if u is a modal variable bound by some **let box** $u = e_1$ **in** e_2 term, then u can only appear in e_2 prefixed by an explicit substitution Θ, written as $\langle\Theta\rangle u$, where different occurrences of u can have different substitutions associated with them. The explicit substitution provides 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 representation of a substitution never defines the same name twice.

The terms $\nu X{:}A.\ e$ and **choose** e are the introduction and elimination form for the type constructor $A \nrightarrow B$. The term $\nu X{:}A.\ e$ binds a name X of type A that can subsequently be used in e. The term choose picks a fresh name of type A, substitutes it for the name bound in the argument ν-abstraction of type $A \nrightarrow B$, and proceeds to evaluate the body of the abstraction. To ensure the progress and preservation properties of $\nu^{\Box}$ (Theorems 12 and 11), we must prevent the bound name in $\nu X{:}A.\ e$ from escaping the scope of its definition. Indeed, if during evaluation, X is encountered outside its defining ν, the evaluation will get stuck. Thus, 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 such an occurrence will not disable the evaluation of e. Allowed positions for X are characterized in one of the following two ways: either X is eventually substituted away by an explicit substitution, or X appears in a part of the term that is not encountered during evaluation. Technically, this discipline is enforced by requiring that X does not appear in the type or the support of e.

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 the already existing ones. Terms are considered equal if they differ only in the syntactic representation of their bound variables and names, or in the ordering of names in the explicit substitutions. The binding forms in the language are $\lambda x{:}A.\ e$, **let box** $u = e_1$ **in** e_2 and $\nu X{:}A.\ e$. As usual, capture-avoiding substitution $[e_1/x]e_2$ of expression e_1 for the variable x in the expression e_2 is defined to rename bound variables and names when descending

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

into their scope. Given a term e, we denote by $\mathbf{fv}(e)$ and $\mathbf{fn}(e)$ the set of free variables of e and the set of names appearing in e at the meta level. In addition, we overload the function **fn** so that given a type A and a support set C, $\mathbf{fn}(A[C])$ is the set of names appearing in A or C.

Example 2 To illustrate our new constructors, we present a version of the staged exponentiation function that we can write in $\nu^{\Box}$-calculus. In this and in other examples we resort to concrete syntax in ML fashion, and assume the presence of the base type of integers, recursive functions and let-definitions.

```
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, incurring capture. 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 evaluated.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Definition 1 (*Explicit substitution, its domain and range*)
An explicit substitution is a finite set of pairs $X \to e$, where X is a name and e is a term, so that a name X appears paired up with at most one term. Given a substitution Θ, its domain and range are the following sets.

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

and

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

The set $\mathbf{fv}(\Theta)$ of free variables of Θ is defined to be the set of free variables of expressions in $\mathbf{range}(\Theta)$. The set $\mathbf{fn}(\Theta)$ of free names of Θ includes the names from names $\mathbf{dom}(\Theta)$ and the names appearing freely in the terms from $\mathbf{range}(\Theta)$.

Each substitution Θ defines a unique function $[\![\Theta]\!] : \mathit{Names} \to \mathit{Terms}$, defined as follows.

$$[\![\Theta]\!](X) = \begin{cases} e & \text{if } X \to e \in \Theta \\ X & \text{otherwise} \end{cases}$$

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

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

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

An important aspect of the above definition is that substitution application does not recursively descend under box. This property 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, their composition $\Theta_1 \circ \Theta_2$ is the set

$$\Theta_1 \circ \Theta_2 = \{X \to \{\Theta_1\}([\![\Theta_2]\!](X)) \mid X \in \mathbf{dom}(\Theta_1) \cup \mathbf{dom}(\Theta_2)\}$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

It will occasionally be beneficial to represent this set as a disjoint union of two smaller sets Ψ_1 and Ψ_2 defined as:

$$\begin{aligned}\Psi_1 &= \{X \to [\![\Theta_1]\!](X) \mid X \in \mathbf{dom}(\Theta_1) \setminus \mathbf{dom}(\Theta_2)\}\\ \Psi_2 &= \{X \to \{\Theta_1\}([\![\Theta_2]\!](X)) \mid X \in \mathbf{dom}(\Theta_2)\}\end{aligned}$$

It is important to notice that, though the definitions of substitution application and substitution composition are mutually recursive, both operations are well founded. Substitution application is defined inductively over the structure of its argument, so the size of terms on which it operates is always decreasing. Computing $\Theta_1 \circ \Theta_2$ only requires applying Θ_1 to subterms in Θ_2.

Lemma 4
Let $\Theta_1, \Theta_2, \Theta_3$ be explicit substitutions. If e is a $\nu^{\Box}$-term, then:

1. $\{\Theta_1\}(\{\Theta_2\}e) = \{\Theta_1 \circ \Theta_2\}e$
2. $\Theta_1 \circ (\Theta_2 \circ \Theta_3) = (\Theta_1 \circ \Theta_2) \circ \Theta_3$

Proof
By simultaneous induction on the structure of e and Θ_3. We present the characteristic cases.

case $e = \langle\Theta\rangle u$. By definition, $\{\Theta_1\}(\{\Theta_2\}e) = \langle\Theta_1 \circ (\Theta_2 \circ \Theta)\rangle u$. By second induction hypothesis, this is equal to $\langle(\Theta_1 \circ \Theta_2) \circ \Theta\rangle u = \{\Theta_1 \circ \Theta_2\}e$.

case $\Theta_3 = (X \mapsto e, \Theta')$. Let Z be an arbitrary name.
If $Z = X$, then $\{\Theta_1\}([\![\Theta_2 \circ \Theta_3]\!](Z)) = \{\Theta_1\}(\{\Theta_2\}e)$. By first induction hypothesis, this is equal to $\{\Theta_1 \circ \Theta_2\}e = \{\Theta_1 \circ \Theta_2\}([\![\Theta_3]\!](Z))$.
If $Z \neq X$, then $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta_3]\!](Z) = \{\Theta_1\}\,[\![\Theta_2 \circ \Theta']\!](Z)$, and also $\{\Theta_1 \circ \Theta_2\}\,[\![\Theta_3]\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta']\!](Z)$. By second induction hypothesis, $\Theta_1 \circ (\Theta_2 \circ \Theta') = (\Theta_1 \circ \Theta_2) \circ \Theta'$, and therefore $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta']\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta']\!](Z)$. Therefore, $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta_3]\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta_3]\!](Z)$, thus concluding the proof.

□

We will frequently blur the distinction between a substitution Θ, and its corresponding function $[\![\Theta]\!]$, and write $\Theta(X)$ instead of $[\![\Theta]\!](X)$, or $\{\Theta\}(X)$. Representations of substitutions that differ only in the ordering of the assignment pairs are considered to define equal substitutions.

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 work with three contexts: context of names Σ, context of modal variables Δ, and a context of value variables Γ (the syntactic structure of all three

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Explicit substitutions

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

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

Hypothesis

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

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

λ-calculus

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

Modality

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

Names

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

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

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 support C. The second judgment types the explicit substitutions. Given a substitution Θ and two support sets C and D, the substitution has type $[C] \Rightarrow [D]$ if it maps expressions of support C to expressions of support D.

The typing rules of $\nu^{\Box}$ are presented in Figure 3. A pervasive characteristic of the type system is *support weakening*. If an expression has support C, then the names the expression contains are elements of C. If $C \subseteq D$, these names are elements of D as well, and the expression may also be ascribed the support D. Support weakening is admissible in both judgments of the type system, which we prove in Section 3.4.

Explicit substitutions. The empty substitution gives rise to the identity function on terms, as applying the empty substitution over a given term does not change the term itself. Thus, when an empty substitution is applied to a term containing names from C, the resulting term obviously contains the same names, all of which are elements of D as well. The typing rule for empty substitutions formalizes this property. The set $D \supseteq C$ is introduced in order to ensure that the support weakening principle holds for this judgment. We implicitly require that both C and D are well-formed;

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

that is, they both contain only names already declared in the name context Σ. The rule for non-empty substitutions recursively checks if each of component expressions is well-typed.

When an explicit substitution $\Theta : [C] \Rightarrow [D]$ is applied to an expression $e : A\,[C]$, the result $\{\Theta\}e$ will have support D. Consider for example the explicit substitution $\Theta = (X \rightarrow 10, Y \rightarrow 20)$, with the domain $\mathbf{dom}(\Theta) = \{X, Y\}$. This substitution can be given (among others) the typings: $[X, Y] \Rightarrow [\,]$, but also $[\,] \Rightarrow [\,]$, or $[X, Y, Z] \Rightarrow [Z]$. And indeed, if Θ is applied to an expression with support $\{X, Y\}$, the result will be an expression with empty support. Similarly, Θ maps an expression of support $[\,]$ into another expression with support $[\,]$, and an expression with support $[X, Y, Z]$ into one 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, $C \subseteq \mathbf{dom}(\Sigma)$. The rule for value variables is straightforward; it postulates that the typing $x{:}A$ can be inferred, if $x{:}A$ is declared in Γ. The support of such a term can be any well-formed support set C. The rule for modal variables prescribes that modal variables are always prefixed by an explicit substitution of matching support.

λ-calculus fragment. The rule for λ-abstraction is standard. It implicitly assumes that the argument type A is well-formed in the name context Σ before the argument type 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 rule for box checks the boxed expression e against an empty context of value variables. This way, it insures that stages of the language are kept separate, as variables from the early stage cannot be referenced in the late stage.

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 rule for let box is also a straightforward extension of the corresponding $\lambda^{\Box}$ rule. The only difference is that the bound modal variable u from the context Δ now has to be stored with its support annotation.

Names fragment. The introduction form for names is $\nu X{:}A.\ e$ with its corresponding type $A \nrightarrow B$. It introduces the name $X{:}A$ to be used in the expression e. It is assumed that the type A is well-formed relative to the context Σ. The term constructor choose is the elimination form for $A \nrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction. In other words, the operational semantics of the 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 side-condition $X \notin \mathbf{fn}(B[C])$ of the rule for ν-abstraction serves to

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

enforce the typing discipline on the appearances of X in e. It effectively limits X to appear only in subterms of e that can never be evaluated or in subterms from which it will eventually be removed by some explicit substitution. For example, consider the term

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

This term contains a substituted occurrence of X and an unused occurrence of Y, and is therefore well-typed (of type $int \nrightarrow int \nrightarrow \Box int$). Another way to paraphrase this typing discipline is the following: when leaving the scope of a name 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.

In the remainder of this section, we compare the name discipline of $\nu^{\Box}$ to some previous work on name calculi. The main motivation for the $\nu^{\Box}$ treatment of names comes from the work on FreshML (Pitts & Gabbay, 2000). In FreshML, names are introduced into the computation by the construct **new** X **in** e which is roughly equivalent to our **choose** $(\nu X.\ e)$. We have decided on this decomposition in order to make the types of the language follow more closely the intended meaning of the terms, as it is the case in the simply-typed λ-calculus. In simply-typed λ-calculus, the computational content of programs is defined by β-reduction. Generating new names obviously has computational interpretation in $\nu^{\Box}$, and our decomposition gives us an appropriate β-reduction to which we can ascribe this computational content:

$$\Sigma, \textbf{choose}\ (\nu X{:}A.\ e) \longmapsto_{\beta} (\Sigma, X{:}A), e$$

Given $e : A \nrightarrow B$, we also have the η-expansion:

$$\Sigma, e \longmapsto_{\eta} \Sigma, \nu X{:}A.\ \textbf{choose}\ e$$

In FreshML, if X is a name appearing in the term e, then the support of e will contain X, unless X occurs in dead code, or is otherwise abstracted using a specific term constructor for name abstraction. The type system of FreshML insists on a side condition similar to our rule for ν, in order to prevent unabstracted names from escaping the scope of their introducing new.

This side condition provides significant simplifications when compared to some previous work on names. For example, the ν-calculus of Pitts & Starck (1993) is similar to FreshML, but it does not track the appearance of names in the terms. This gives rise to a very powerful language, but also a very complex one. The ν-calculus has a rather involved equational theory; in particular, it does not equate a term with its β-reduct.

The $\lambda\nu$-calculus (Odersky, 1994) introduces a somewhat different idea for treating names, characterized by reductions that push the name declaration inside other term

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

constructors. A typical reduction rule in λv would be paraphrased in the notation of $v^{\Box}$ as

$$\textbf{choose}\ (\nu X.\ (\lambda x.\ e)) \longmapsto \lambda x.\ (\textbf{choose}\ (\nu X.\ e)) \qquad (*)$$

Just like the ν-calculus, λv does not keep track of names either. As a consequence, it does not possess the usual progress and preservation properties, because the evaluation of well-typed expressions in λv may get stuck. The typical example is the expression $\nu X.\ X$, which is well typed, but does not denote any value.

All the cited name calculi are designed around the single goal: that of providing the operation of equality on names. In contrast to this goal, the $v^{\Box}$-calculus uses names primarily as a way of describing supports, i.e. as a way of specifying the dependency of an expression on names. In fact, names in the $v^{\Box}$-calculus are second-class objects – they cannot be passed as arguments to other functions, and may not be tested for equality.

Insisting on second-class names is somewhat restrictive when compared to the ν-calculus of Pitts & Stark (1993), and λv of Odersky (1994). However, it allows that names be tracked by the type system (in this respect the $v^{\Box}$-calculus is similar to FreshML), which is exactly the functionality required by our application to staged computation. Furthermore, it results in a language with rather simple and pleasing properties. For example, in Section 5 we explore the equational theory of $v^{\Box}$, and establish that the notion of logical equivalence that we define validates all the β-reductions and η-expansions of $v^{\Box}$, as well as the equivalence (*) of the λv-calculus.

At this point, it may be interesting to observe that, while names in the $v^{\Box}$-calculus may not be *directly* tested for equality, it is possible to test them for equality *indirectly*. Indeed, as mentioned before, $v^{\Box}$ may be extended with pattern-matching against boxed syntactic expressions (Nanevski, 2002). Since the syntactic expressions may contain names, this will provide an indirect way to test for name equality. This extension, however, is beyond the scope of the current paper.

Example 3 This example presents the function `conv` for computing the convolution of two integer lists. Convolution of lists $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_n]$, is the list $[x_n y_1, \ldots, x_1 y_n]$. We ignore the possibility that the two lists can be of different sizes.

The function `conv`, which we present in Fig. 4, is staged in the first argument, so that given the list x, `conv` outputs a source code specialized for computing the convolution with x. In this example, we assume the existence of a function `lift : int` $\rightarrow \Box$`int`, mapping each integer n into **box** n. This is a reasonable assumption, as the base type of integers is always considered observable; in any realistic situation, it would be possible to coerce an integer value into its own syntactic representation. The helper function `conv'` recurses over the list x to build the output code; it keeps the unfinished part of the output abstracted using the variable `z:`$\Box_{\mathrm{TL}}$`intlist`.

Specializing `conv` to the list `[3,2]` results with the following program.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

```
(*
 * val conv : intlist ->
 *             □(intlist -> intlist)
 *)

fun conv (xs : intlist) =
 choose νTL:intlist.
  (*
   * conv' : intlist -> □TLintlist
   *           -> □(intlist -> intlist)
   *)

  let fun conv' (nil) =
     λz:□TLintlist.
        let box u = z
        in
            box (λy:intlist.
                   <TL -> y>u)
        end
| conv' (x::xs') =
  let val f = conv' (xs')
      box x' = lift x
  in
     λz:□TLintlist.
      let box u = z
      in
        f (box (
            let val (hd::tl) = TL
            in
                x'*hd :: <TL -> tl>u
            end))
      end
  end

in
  conv' xs (box nil)
end
```

Fig. 4. Staged convolution.

```
- conv [3,2];
val it =  box (λy:intlist.
                  let val (hd::tl) = y
                  in
                      2*hd :: let val (hd::tl) = tl
                              in
                                  3*hd :: nil
                              end
                  end) : □(intlist -> intlist)
```

It remains a challenge to write a program that could generate a yet more concise specialized code – like for example the following fragment for convolution with `[3,2]`:

```
box (λy:intlist. let val (y1::y2::tl) = y in [2*y1, 3*y2])
```

□

3.4 Structural properties

This section explores the basic theoretical properties of the $\nu^{\Box}$ type system. The lemmas developed here will be used to justify the operational semantics that we ascribe to the $\nu^{\Box}$-calculus in section 3.5, and will ultimately lead to the proof of type preservation (Theorem 11) and progress (Theorem 12).

Lemma 5 (*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 $\Sigma';\Delta';\Gamma' \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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. □

Contraction on names does not hold in $v^{\Box}$. Indeed, identifying two different names in a term may make the term syntactically ill-formed. Typical examples are explicit substitutions. Identifying two names may make an otherwise well-formed substitution assign two different images to the same name.

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

Lemma 6 (*Support weakening*)
Support weakening is covariant on the right-hand side and contravariant on the left-hand side of the judgments. More formally, let $C \subseteq C' \subseteq \mathbf{dom}(\Sigma)$ and $D' \subseteq D \subseteq \mathbf{dom}(\Sigma)$ be well-formed support sets. Then the following holds:

1. if $\Sigma;\Delta;\Gamma \vdash e : A\ [C]$, then $\Sigma;\Delta;\Gamma \vdash e : A\ [C']$.
2. if $\Sigma;\Delta;\Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$, then $\Sigma;\Delta;\Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C']$.
3. if $\Sigma;\Delta;\Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$, 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 part is proved by induction on the structure the derivation. □

The following lemma shows that the intuition behind the typing judgment for explicit substitutions explained in section 3.3 is indeed valid; if $\Theta : [C] \Rightarrow [D]$ is applied to an expression of support C, then the result is an expression of support D.

Lemma 7 (*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.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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

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

Let $X{:}A$. It suffices to show that

(a) if $X \notin \mathbf{dom}(\Psi)$ and $X \in C_1$, then $X \in D$
(b) if $X \to e \in \Psi$, then $\Sigma;\Delta;\Gamma \vdash e : A\,[D]$

To establish (a), observe that $X \notin \mathbf{dom}(\Psi)$ implies $X \notin \mathbf{dom}(\Theta)$ and $X \notin \mathbf{dom}(\Theta')$, by definition. If $X \notin \mathbf{dom}(\Theta')$ and $X \in C_1$, then $X \in C$ by the typing of Θ'. If $X \notin \mathbf{dom}(\Theta)$ and $X \in C$, then $X \in D$, by the typing of Θ.

To establish (b), we need to consider two cases: (1) $X \to e \in \Psi'_1$ and (2) $X \to e \in \Psi'_2$. In case (1), by the typing of Θ, we immediately have $\Sigma;\Delta;\Gamma \vdash e : A\,[D]$. In case (2), there exists a term e' such that $X \to e' \in \Theta'$ and $e = \{\Theta\}e'$. By the typing of Θ', we have $\Sigma;\Delta;\Gamma \vdash e' : A\,[C]$, and by then by the first induction hypothesis, $\Sigma;\Delta;\Gamma \vdash \{\Theta\}e' : A\,[D]$. This concludes the proof, since $e = \{\Theta\}e'$. □

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

Lemma 8 (*Value substitution principle*)
Let $\Sigma;\Delta;\Gamma \vdash e_1 : A\,[C]$. The following holds:

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

Proof
Simultaneous induction on the two derivations. □

The situation is not that simple with modal variables. A simple substitution of an expression for some modal variable will not result in a syntactically well-formed term. The reason is, as discussed before, that occurrences of modal variables are always prefixed by an explicit substitution. But, explicit substitutions in $\nu^{\Box}$-calculus can occur only immediately before modal variables, and cannot be freely applied to arbitrary terms[1]. Hence, if a substitution of the expression e for a modal variable u is to produce a syntactically valid term, we need to follow it up with applications of *explicit name substitutions* that were paired up with each occurrence of u. The definition below generalizes capture-avoiding substitution of modal variables in order to handle this problem.

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

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Definition 9 (*Modal substitution*)
The capture-avoiding substitution of e for an *expression variable* u is defined recursively as follows

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

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 10 (*Modal substitution principle*)
Let e_1 be an expression 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 7.1), $\Sigma;\Delta;\Gamma \vdash \{[\![e_1/u]\!]\Theta\}e_1 : B\,[D]$
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

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

which relates an expression e with its one-step reduct e'. The expressions e and e' do not contain any free variables, but they may contain free names. However, we require that e and e' must have *empty support*. In other words, we only consider for evaluation those terms whose names appear exclusively in boxed subterms, or are

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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

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

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

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

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

Fig. 5. Structured operational semantics of $\nu^{\Box}$-calculus.

otherwise captured by some explicit substitution. Because free names are allowed under these conditions, the operational semantics has to keep track of them in the run-time name contexts Σ and Σ'. The rules of the judgment are given in Fig. 5, and the values of the language are generated by the grammar below.

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

The rules agree with the β-reductions of the calculus, and are standard except for two important observations. First of all, the β-redex for the type constructor $\rightarrowtail$ extends the run-time context with a fresh name before proceeding. This way, we keep track of names that have been generated in the course of evaluation, so that we can select a fresh name when it is needed.

Even more important is to observe that names in $\nu^{\Box}$ are *not values*. This is a direct consequence of the fact that names in $\nu^{\Box}$ can be ascribed an arbitrary type. If a name $X : A$ were a value, then introducing X into the computation extends the type A with a new value. Such a dynamic type extension effectively renders the already defined functions of domain A incomplete. Suddenly, if a function f has domain A, then it is forced to check at run time if its argument is a name-free value (in which case f can be applied), or if its argument is an expression containing a name X. This is where the modal constructor $\Box$ comes in – it classifies object expressions with names, so that the above checks can be done statically during type checking. Thus, while $X{:}A$ is not a value in $\nu^{\Box}$, the expression $(\textbf{box}\ X) : \Box_X A$ is.

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

Theorem 11 (*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 reduction relation, using inversion on the typing derivation and the substitution principles. $\Box$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Theorem 12 (*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 case analysis of e, employing inversion on the typing derivation. □

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, in fact, 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 13 (*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 $\mathbf{dom}(\Sigma)$, 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 $e = \mathbf{choose}\ (\nu X{:}A.\ e')$. 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. □

4 Support polymorphism

It is frequently necessary to write programs that are polymorphic in the support of their arguments, because they manipulate syntactic expressions of unknown support. A typical example is a function that recurses over an expression with binding structure. When this function 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 & Reynolds (Girard, 1986; Reynolds, 1983).

To add support polymorphism to the simple $\nu^{\Box}$-calculus, we create a new syntactic category of *support variables*, which stand for unknown support sets. Then the rest of

[2] The calculus described in this document 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.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

the syntax of $v^{\Box}$ is extended to take support variables into account. We summarize the changes in the following table.

Support variables	p, q	$\in$	$\mathscr{S}$
Supports	C, D	$::=$	$\ldots \mid C, p$
Types	A	$::=$	$\ldots \mid \forall p.\ A$
Expressions	e	$::=$	$\ldots \mid \Lambda p.\ e \mid e\ [C]$
Name contexts	Σ	$::=$	$\ldots \mid \Sigma, p$
Values	v	$::=$	$\ldots \mid \Lambda p.\ e$

Before a support variable can be used, it has to be declared in the name context Σ. For the new definition of Σ, we retain the same well-formedness conditions as before. In particular, a support variable $p \in \Sigma$ may only be used to the right of its declaration. It is important that supports themselves are allowed to contain support variables, to express the situation in which only a portion of a support set is known. Consequently, the function $\mathbf{fn}(-)$ is updated to return the set of names *and support variables* appearing in its argument. The family of types is extended with the type $\forall p.\ A$ expressing universal support quantification. Its introduction form is $\Lambda p.\ e$, which binds a support variable 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.

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, $D \subseteq \mathbf{dom}(\Sigma)$. The operational semantics for the new constructs is also not surprising.

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

The extended language satisfies the following substitution principle.

Lemma 14 (*Support substitution principle*)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathbf{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p.

Then the following holds.

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

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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 6.3), $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : [C_1' \setminus \{X\}] \Rightarrow [C_2']$
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 type preservation (Theorem 11) and progress (Theorem 12) whose additional cases involving support abstraction and application are handled using the above Lemma 14.

Example 4 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 5 As an example of a more realistic program we present the regular expression matcher from Davies & Pfenning (2001) and Davies (1996). The example

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

```
(*
 * 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. 6. Unstaged regular expression matcher.

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
```

We also assume a primitive predicate `null : char list -> bool` for testing if the input list of characters is empty. Figure 6 presents an ordinary ML implementation of the matcher, and $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ versions can be found in Davies & Pfenning (2001) and Davies (1996). The helper function `acc1` in Fig. 6 takes a regular expression e, a continuation function k, and an input string s (represented as a list of characters). The function attempts to match a *prefix* of s to the regular expression e. If the matching succeeds, then the remainder of s is passed to the continuation k to determine if s is accepted or not.

We now want to use the $v^{\Box}$-calculus to stage the program from Fig. 6 so that it can be *specialized* with respect to a given regular expression. For that purpose, it is useful to view the helper function `acc1` from Fig. 6 as a code generator. Indeed, `acc1` may be seen as follows: it first generates code for matching a string against a regular expression e, and then appends k to that code. This is the main idea behind the function `acc`, and the $v^{\Box}$ program in Fig. 7. In this program,

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

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

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

  (*
   * acc : regexp -> ∀p.(□S,pbool
   *                      -> □S,pbool)
   *)

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

        | acc (Plus (e1, e2)) [p] k =
           choose vJOIN : 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 vT : char list
           choose vLOOP : 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. 7. Regular expression matcher staged in the $\nu^{\Box}$-calculus.

we use the name S for the input string to be matched by the code that `acc` generates. The continuation k is not a function anymore, but code to be attached at the end of the generated result. We want code k to contain further names standing for the 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`)`. For example, observe in the alternation case that the generated code does not duplicate the "continuation" code of 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.

At this point, it may be illustrative to trace the execution of the program on a concrete input. Figure 8 presents the function calls and the intermediate results that occur when the $\nu^{\Box}$ matcher is applied to the regular expression `Star(Empty)`. The resulting specialized program is a function accepting only the empty string. This

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

```
▷ 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. 8. Example execution trace for a regular expression matcher in $\nu^{\Box}$. Function calls are marked by ▷ and the corresponding return results are marked by an aligned ◁.

function does not contain variable-for-variable redexes, thanks to the features and expressiveness of $\nu^{\Box}$, but it unnecessarily tests if `t = t`. Removing these extraneous tests requires some further examination and preprocessing of e, but the thorough description of such a process is beyond our scope. We refer to Harper (1999) for an insightful analysis.

□

5 Logical relations for program equivalence

In this section we develop the notion of equivalence between programs in the core $\nu^{\Box}$-calculus (without recursion and support polymorphism), with which we establish the intensional properties of the modal operator, and justify our intuitive view of $\Box_C A$ as classifying *syntactic* expressions.

To that end, we consider two notions of equivalence. The first is *intensional*, or syntactic, by which two programs are equal if and only if their abstract syntax representations are the same; the programs may only differ in the names of their bound variables, and possibly also in the representation of their explicit substitutions. On the other hand, two programs are *extensionally* equivalent if, in some appropriate sense which we will define shortly, they produce the same results. Of course, if two expression are intensionally equivalent, they should also be extensionally equivalent.

One of the questions that we explore in this section is an interplay between intensional and extensional equivalences of programs. The $\nu^{\Box}$-calculus is particularly appropriate for investigating and combining the two notions, because we can use the

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

modal constructs as explicit boundaries between the different notions of equivalence. In particular, we can treat values of modal types as being *observable*, i.e. amenable to inspection of their structure. Then two general expressions of modal type will be extensionally equivalent if and only if their values are intensionally equivalent. We are also interested in exploring the properties of the calculus when only extensional equivalence is used, as the present formulation of $v^{\Box}$ does not contain any constructs for inspecting the structure of modal values. In both of these cases, we will establish that our formulation of $v^{\Box}$ is purely functional, in the sense that it satisfies the logical equivalences arising from the β-reductions and η-expansions of the language. The development presented here will follow the methodology of logical relations, as used, for example, in other works concerned with names in functional programming (Pitts & Stark, 1993). However, the details of our approach are different because we want to make the identity of locally declared names irrelevant for the purposes of expression comparison.

To motivate our approach, we first present several examples of intensional and extensional equivalences that we would like our programs to satisfy. We use the symbol $\cong$ for extensional equivalence, and $=$ for intensional equivalence. The equivalences will always be considered at a certain type and support.

Example 6 In the examples below, we assume that X is a name of integer type.

1. $(\lambda x{:}\mathbf{int}.\ x+1)\ 2 \cong (\lambda x{:}\mathbf{int}.\ x+2)\ 1 \cong 3 : \mathbf{int}$, because all three terms evaluate to 3; however, neither of them is intensionally equivalent to any other.
2. $(\lambda x{:}\mathbf{int}.\ x+X)\ 2 \cong 2+X \cong X+2 : \mathbf{int}\ [X]$, because whenever X is substituted by e (and x is not free in e), the three terms evaluate to the same value.
3. $(\lambda x{:}\Box_X\mathbf{int}.\ 2)\ (\mathbf{box}\ X) \cong (1+1) : \mathbf{int}$, because both terms evaluate to 2. Notice that X does not appear in the second term, nor in the type and support of comparison.
4. $\mathbf{box}\ (X+1) \cong \mathbf{box}\ (X+1) : \Box_X\mathbf{int}$, because $X+1 = X+1 : \mathbf{int}\ [X]$ intensionally, as syntactic expressions.

□

As illustrated by this example, in our equivalence relations we should distinguish between two different kinds of names: (1) names which may appear in either of the compared terms, as well as their type and support (Example 6, cases 2 and 4), and (2) names which are local to some of the terms (case 3). The later kind of names should not influence the equivalence relations – these names could freely be renamed.

The described requirement leads to the following formulation of our relations. The judgment for intensional equivalence compares two expressions for syntactic equality modulo α-equivalence

$$e_1 = e_2,$$

and the judgment for extensional equivalence has the form

$$\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\ [C].$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

In this judgment, we assume that Σ is a well-formed name context and that Δ, Γ, Σ_1, Σ_2, A and C are all well-formed with respect to Σ. Intuitively, the context Σ declares the names that matter when comparing two terms; hence the requirement that Δ, Γ, A and C contain only the names from Σ. On the other hand, the contexts Σ_1 and Σ_2 declare the names that may appear in e_1 and e_2, but these names are, in some sense, irrelevant. They will be subject to renaming, as they do not appear in Δ, Γ, A or C.

For the purposes of this section, we further restrict our considerations of intensional equivalence to only modal terms which are themselves part of the simply typed fragment of $v^{\Box}$. In other words, we introduce new categories of *simple types* and *simple terms* as follows:

1. a type A is *simple* iff $A = b$, or $A = A_1 \to A_2$ or $A = A_1 \nrightarrow A_2$ where A_1, A_2 are simple types
2. a term e is *simple* if it does not contain the modal constructs box and let box.

Then we only allow modal types $\Box_C A$ if A is simple, and modal terms **box** e if e is simple. We justify this restriction by a desire to avoid impredicativity arising in a language that can intensionally analyze the whole set of its expressions. In fact, it seems rather improbable that a language with such strong intensional capabilities can be designed at all. Indeed, we added names and modal constructs in order to represent syntax with free variables. But, the modal constructs can also bind variables, so a new category of names and modalities seems to be required in order to analyze these new bindings, and then a new category of names and modalities is required for the bindings by the previous class of modalities, etc. Thus, here we limit the intensional equivalence to the simply-typed fragment, and leave the possible extensions to larger fragments for future work.

The next step in the development is to formally define the notion of extensional equivalence. As already mentioned before, the idea is that two expressions are considered extensionally equivalent, if and only if they evaluate to the same value. The values that we will consider for comparison are the values at base type b of natural numbers, and values at modal types $\Box_C A$ which are closed simple terms of type A and support C, which we compare for intensional equivalence.

The standard approach in the development of logical relations starts with a bit different premise. Rather than evaluate two expressions and check if their values are the same, we need to check if the values are *extensionally equivalent*. The later notion is much more permissive, which is particularly important when comparing values of functional types. Indeed, two functions ought to be equivalent not only if they are the same, but also if they map related arguments to related results.

Thus, we need to define two mutually recursive judgments: one for the extensional equivalence of (closed) expressions, and another for extensional equivalence of values. Our judgment for extensional equivalence of expressions has the form

$$\Sigma \vdash \Sigma_1.\, e_1 \cong \Sigma_2.\, e_2 : A\, [C]$$

and the judgment for extensional equivalence of values has the form

$$\Sigma \vdash \Sigma_1.\, v_1 \sim \Sigma_2.\, v_2 : A$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

The first is defined by induction on the structure of A and C, by appealing to the second judgment when the support C is empty. The second is defined by induction on the structure of the type A.

$$
\begin{array}{lll}
\Sigma \vdash \Sigma_1.\, e_1 \cong \Sigma_2.\, e_2 : A\,[\,] & \text{iff} & (\Sigma, \Sigma_1), e_1 \longmapsto^* (\Sigma, \Sigma_1'), v_1, \text{ and} \\
 & & (\Sigma, \Sigma_2), e_2 \longmapsto^* (\Sigma, \Sigma_2'), v_2, \text{ and} \\
 & & \Sigma \vdash \Sigma_1'.\, v_1 \sim \Sigma_2'.\, v_2 : A \\
\Sigma \vdash \Sigma_1.\, e_1 \cong \Sigma_2.\, e_2 : A\,[C] & \text{iff} & \Sigma \vdash \Sigma_1'.\, \{\sigma_1\}e_1 \cong \Sigma_2'.\, \{\sigma_2\}e_2 : A\,[\,] \text{ for} \\
 & & \text{any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma \vdash \Sigma_1'.\, \sigma_1 \cong \\
 & & \Sigma_2'.\, \sigma_2\,[C] \\
\Sigma \vdash \Sigma_1.\, v_1 \sim \Sigma_2.\, v_2 : b & \text{iff} & v_1 = v_2 \in \mathbb{N} \\
\Sigma \vdash \Sigma_1.\, v_1 \sim \Sigma_2.\, v_2 : A \to B & \text{iff} & v_i = \lambda x{:}A.\, e_i \text{ and } \Sigma \vdash \Sigma_1'.\, [v_1'/x]e_1 \cong \\
 & & \Sigma_2'.\, [v_2/x]e_2 : B, \text{ for any } \Sigma_i' \supseteq \Sigma_i, \text{ such} \\
 & & \text{that } \Sigma \vdash \Sigma_1'.\, v_1' \sim \Sigma_2'.\, v_2' : A \\
\Sigma \vdash \Sigma_1.\, v_1 \sim \Sigma_2.\, v_2 : \Box_C A & \text{iff} & v_i = \mathbf{box}\ e_i \text{ and } e_1 = e_2 \text{ and } \Sigma \vdash \\
 & & \Sigma_1.\, e_1 \cong \Sigma_2.\, e_2 : A\,[C] \\
\Sigma \vdash \Sigma_1.\, v_1 \sim \Sigma_2.\, v_2 : A \nrightarrow B & \text{iff} & v_i = \nu X{:}A.\, e_i \text{ and } \Sigma \vdash (\Sigma_1, X{:}A).\, e_1 \cong \\
 & & (\Sigma_2, X{:}A).\, e_2 : B\,[\,], \text{ where } X \text{ is a fresh} \\
 & & \text{name.}
\end{array}
$$

Here we abbreviated:

$$
\begin{array}{lll}
\Sigma \vdash \Sigma_1.\, \sigma_1 \cong \Sigma_2.\, \sigma_2\,[C] & \text{iff} & \sigma_1, \sigma_2 \text{ are explicit substitutions for the names} \\
 & & \text{in } C, \text{ such that } \Sigma \vdash \Sigma_1.\, \sigma_1(X) \cong \Sigma_2.\, \sigma_2(X) : \\
 & & B\,[\,] \text{ for any name } X \in C \text{ such that } X{:}B \in \Sigma.
\end{array}
$$

The most important parts of the above definition are the cases defining the relation for values at functional, modal types and $\nrightarrow$ types. The definition for values at functional types formalizes the intuition that we outlined before: two functions are related if they map related arguments to related results. The definition for values at modal types contrasts the notions of intensional vs. extensional. We consider two values **box** e_1 and **box** e_2 *extensionally* related iff the expressions e_1 and e_2 are *intensionally* related. Observe, however, that in the definition we actually insist on the additional requirement that e_1 and e_2 be extensionally related as well. This extra clause is added because, at this stage of development, it is not obvious that intensional equivalence of expressions implies their extensional equivalence. For that matter, it is not obvious at this point that that the two new relations are indeed equivalences at all. We will prove both of these properties in due time, but we need to start the development with a sufficiently strong definition. The definition for values $\nu X.\, e_1$ and $\nu X.\, e_2$ at the $A \nrightarrow B$ type generates a fresh name X, and then tests e_1 and e_2 for equivalence in the local contexts extended with X.

Notice that the above definitions are well-founded. In order to establish this fact, let us define $\mathrm{ord}_\Sigma(X)$ to be the number of names in Σ on which X depends, and which thus must appear to the left of X in Σ. This includes the names that appear in the type of X, the names that appear in the types of these names, etc. The definition of $\mathrm{ord}_\Sigma(X)$ is then invariant of any dependence preserving reordering of Σ.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

In a similar manner, we define $\mathrm{ord}_\Sigma(A[C])$ to be the number of names in Σ on which the type A and support C depend. These are the names that themselves appear in the type A or support C, or in the types of these names, etc. Because a type of a name cannot depend on that name itself, it is clear that if $X{:}A \in \Sigma$, then $\mathrm{ord}_\Sigma(A) = \mathrm{ord}_\Sigma(X)$. Also, if the name X appears in the type A or support C, then $\mathrm{ord}_\Sigma(X) \leqslant \mathrm{ord}_\Sigma(A[C]) - 1$. This holds because $A[C]$ depends on X and all the names on which X itself depends.

We can now order the pairs of type A and support C as follows. The pair $A[C]$ is smaller than $B[D]$ iff

- $\mathrm{ord}_\Sigma(A[C]) < \mathrm{ord}_\Sigma(B[D])$, or
- $\mathrm{ord}_\Sigma(A[C]) = \mathrm{ord}_\Sigma(B[D])$, but the number of type constructors of A is smaller than the number of type constructors of B.

It is now easy to observe that each inductive step in the definitions of the relations strictly decreases this ordering. Indeed, the relation on values preserves the number of names in the type and support, but makes inductive references using types of strictly smaller structure. The relation on expressions with non-empty support C relies on explicit substitutions over the names in C. But for each name $X \in C$ with $X{:}B \in \Sigma$, it is clear that $\mathrm{ord}_\Sigma(B) = \mathrm{ord}_\Sigma(X) \leqslant \mathrm{ord}_\Sigma(A[C]) - 1$.

We next extend our relations to handle expressions with free variables. We start with expressions of empty support.

$$\Sigma;\cdot;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[\,] \quad \text{iff} \quad \Sigma \vdash \Sigma_1'.\ [\rho_1/\Gamma]e_1 \cong \Sigma_2'.\ [\rho_2/\Gamma]e_2 : A\,[\,] \text{ for any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma \vdash \Sigma_1'.\ \rho_1 \sim \Sigma_2'.\ \rho_2 : \Gamma$$

In this definition, ρ_1, ρ_2 are arbitrary substitutions of *values* for variables in Γ, and we write:

$$\Sigma \vdash \Sigma_1.\ \rho_1 \sim \Sigma_2.\ \rho_2 : \Gamma \quad \text{iff} \quad \Sigma \vdash \Sigma_1.\ \rho_1(x) \sim \Sigma_2.\ \rho_2(x) : A \text{ whenever } x{:}A \in \Gamma$$

In the next step, we consider expressions of arbitrary support.

$$\Sigma;\cdot;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C] \quad \text{iff} \quad \Sigma;\cdot;\Gamma \vdash \Sigma_1'.\ \{\sigma_1\}e_1 \cong \Sigma_2'.\ \{\sigma_2\}e_2 : A\,[\,] \text{ for any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma;\Gamma \vdash \Sigma_1'.\ \sigma_1 \cong \Sigma_2'.\ \sigma_2\,[C]$$

where σ_1, σ_2 are explicit substitutions, and

$$\Sigma;\Gamma \vdash \Sigma_1.\ \sigma_1 \cong \Sigma_2.\ \sigma_2\,[C] \quad \text{iff} \quad \Sigma;\cdot;\Gamma \vdash \Sigma_1.\ \sigma_1(X) \cong \Sigma_2.\ \sigma_2(X) : B\,[\,] \text{ for any name } X \in C \text{ such that } X{:}B \in \Sigma$$

Finally, the relation is extended with the context Δ as follows.

$$\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C] \quad \text{iff} \quad \Sigma;\cdot;\Gamma \vdash \Sigma_1'.\ [\![\delta_1/\Delta]\!]e_1 \cong \Sigma_2'.\ [\![\delta_2/\Delta]\!]e_2 : A\,[C] \text{ for any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma \vdash \Sigma_1'.\ \delta_1 = \Sigma_2'.\ \delta_2 : \Delta$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

where δ_1, δ_2 are arbitrary substitutions of expressions for modal variables in Δ, and

$$\Sigma \vdash \Sigma_1.\ \delta_1 = \Sigma_2.\ \delta_2 : \Delta \quad \text{iff} \quad \delta_1(u) = \delta_2(u) \text{ and } \Sigma \vdash \Sigma_1.\ \delta_1(u) \cong \Sigma_2.\ \delta_2(u) : A\,[C] \text{ whenever } u{:}A[C] \in \Delta$$

The above definitions are well-founded, as each one refers only to already introduced definitions. For the sake of completeness, we also parametrize the intensional relation $=$ with the context Δ, as this will be needed in the statement of Lemma 20.

$$\Sigma;\Delta \vdash \Sigma_1.\ e_1 = \Sigma_2.\ e_2 : A\,[C] \quad \text{iff} \quad [\![\delta_1/\Delta]\!]e_1 = [\![\delta_2/\Delta]\!]e_2 \text{ for any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma \vdash \Sigma_1'.\ \delta_1 = \Sigma_2'.\ \delta_2 : \Delta$$

Example 7 Let $\Sigma = X{:}\mathbf{int}$. Then the following are valid instances of intensional equivalence.

1. $\Sigma;\cdot \vdash X + 1 = X + 1 : \mathbf{int}\,[X]$
2. $\Sigma;u{:}\mathbf{int}[X] \vdash (Y{:}\mathbf{int}).\ \langle X \to 1, Y \to 2\rangle u = \langle X \to 1\rangle u : \mathbf{int}\,[\,]$

□

Example 8 Consider the simple expression e such that $\Sigma;\Delta;\Gamma \vdash \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) : \Box\mathbf{int}$. In such a case, it is easy to see that $\Sigma;\Delta;\Gamma \vdash \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) \cong \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) : \Box\mathbf{int}$.

First notice that we can assume Γ to be empty as, by typing, e cannot contain variables from Γ. We can assume that Δ is empty as well; this will not result in any loss of generality because the relation of *intensional* equivalence is closed with respect to modal substitutions δ.

The above relation holds if and only if the two instances of the expression $\mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e)$ evaluate to related values. But, indeed they do, as the particular choice of X in the evaluation of the expressions does not influence e. In fact, because e is a simple expression, the only names that may appear in $\mathbf{box}\ e$ are the ones appearing in its type. In this case, the type in question is $\Box\mathbf{int}$, and it does not contain any names.

Because of reflexivity of α-equivalence, $e = e$. By determinacy of evaluation, it is also the case that $\Sigma \vdash e \cong e : \mathbf{int}$. Thus, we can conclude that $\Sigma \vdash \mathbf{box}\ e \cong \mathbf{box}\ e : \Box\mathbf{int}$. □

Lemma 15 (*Name permutation*)
Let $R_1 : \Sigma_1 \to \Sigma_1'$ and $R_2 : \Sigma_2 \to \Sigma_2'$ be bijections where Σ_1' and Σ_2' are well-formed in Σ. Then:

1. if $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma \vdash \Sigma_1'.\ R_1\ e_1 \cong \Sigma_2'.\ R_2\ e_2 : A\,[C]$
2. if $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, then $\Sigma \vdash \Sigma_1'.\ R_1\ v_1 \sim \Sigma_2'.\ R_2\ v_2 : A$

Proof
By induction on the structure of the definition of the two judgments.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

For the first induction hypothesis, we start by considering the base case when C is empty. In this case, if $(\Sigma, \Sigma_i), e_i \longmapsto^* (\Sigma, \Sigma_i, \Psi_i), v_i$, then by parametricity of the evaluation judgment, we also have $(\Sigma, \Sigma'_i), e_i \longmapsto^* (\Sigma, \Sigma'_i, \Psi_i), R_i\ v_i$. Then we appeal to the second induction hypothesis, to derive that $\Sigma \vdash (\Sigma'_1, \Psi_1).\ R_1\ v_1 \sim (\Sigma'_2, \Psi_2).\ R_2\ v_2 : A$. The result is easily extended to the case when C is not empty.

For the second induction hypothesis, the only interesting case is when $A = \Box_D B$, which is proved by appealing to the first induction hypothesis, and the fact that name permutation does not change the $=$ relation on simple terms. □

Lemma 16 (*Name localization*)
If C is a well-formed support in Σ, then the following holds:

1. $(\Sigma, \Sigma') \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\ [C]$ if and only if $\Sigma \vdash (\Sigma', \Sigma_1).\ e_1 \cong (\Sigma', \Sigma_2).\ e_2 : A\ [C]$
2. $(\Sigma, \Sigma') \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$ if and only if $\Sigma \vdash (\Sigma', \Sigma_1).\ v_1 \sim (\Sigma', \Sigma_2).\ v_2 : A$

Proof
By induction on the structure of the definition of the two judgments.

For the first induction hypothesis, we start by considering the case when C is empty. Let $(\Sigma, \Sigma', \Sigma_i), e_i \longmapsto^* (\Sigma, \Sigma', \Psi_i), v_i$, and $(\Sigma, \Sigma') \vdash \Psi_1.\ v_1 \sim \Psi_2.\ v_2 : A$. By second induction hypothesis, $\Sigma \vdash (\Sigma', \Psi_1).\ v_1 \sim (\Sigma', \Psi_2).\ v_2 : A$, and thus also $\Sigma \vdash (\Sigma', \Psi_1).\ e_1 \cong (\Sigma', \Psi_2).\ e_2 : A$. The opposite direction is symmetric. The result is easily extended to the case of non-empty C.

For the second induction hypothesis, we present the case when $A = A_1 \to A_2$, and $v_i = \lambda x{:}A_1.\ e_i$. In this case, consider $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma \vdash (\Sigma', \Sigma'_1).\ v'_1 \sim (\Sigma', \Sigma'_2).\ v'_2 : A_1$. We need to show $\Sigma \vdash (\Sigma', \Sigma'_1).\ [v'_1/x]e_1 \cong (\Sigma', \Sigma'_2).\ [v'_2/x]e_2 : A_2$. By induction hypothesis at type A_1, we have that $(\Sigma, \Sigma') \vdash \Sigma'_1.\ v'_1 \sim \Sigma'_2.\ v'_2 : A_1$, and therefore $(\Sigma, \Sigma') \vdash \Sigma'_1.\ [v'_1/x]e_1 \cong \Sigma'_2.\ [v'_2/x]e_2 : A_2$. By induction hypothesis at type A_2, we can push Σ' back inside to get $\Sigma \vdash (\Sigma', \Sigma'_1).\ [v'_1/x]e_1 \cong (\Sigma', \Sigma'_2).\ [v'_2/x]e_2 : A_2$. The opposite direction is symmetric. □

Lemma 17 (*Weakening*)
Let $\Sigma' \supseteq \Sigma$, $\Sigma'_1 \supseteq \Sigma_1$ and $\Sigma'_2 \supseteq \Sigma_2$, so that Σ'_1 and Σ'_2 are well-formed with respect to Σ'. Then the following holds:

1. if $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\ [C]$, then $\Sigma' \vdash \Sigma'_1.\ e_1 \cong \Sigma'_2.\ e_2 : A\ [C]$
2. if $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, then $\Sigma' \vdash \Sigma'_1.\ v_1 \sim \Sigma'_2.\ v_2 : A$

Proof
By name localization (Lemma 16), it suffices to consider $\Sigma' = \Sigma$. The proof is by simultaneous induction on the definition of the two judgments.

For the first statement, we only consider the case when C is empty, as the result is easily generalized to non-empty C. In this case, let $(\Sigma, \Sigma_i), e_i \longmapsto^* (\Sigma, \Sigma_i, \Psi_i), v_i$, such that $\Sigma \vdash (\Sigma_1, \Psi_1).\ v_1 \sim (\Sigma_2, \Psi_2).\ v_2 : A$. By name permutation, we could assume that Ψ_1, Ψ_2 are disjoint from Σ'_1, Σ'_2, so that also $(\Sigma, \Sigma'_i), e_i \longmapsto^* (\Sigma, \Sigma'_i, \Psi_i), v_i$. Then by second induction hypothesis, $\Sigma \vdash (\Sigma'_1, \Psi_1).\ v_1 \sim (\Sigma'_2, \Psi_2).\ v_2 : A$, and therefore $\Sigma \vdash \Sigma'_1.\ e_1 \cong \Sigma'_2.\ e_2 : A$.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

For the second induction hypothesis, the only interesting case is when $A = A' \to A''$, and $v_i = \lambda x{:}A'.\ e_i$. In this case, consider $\Sigma_i'' \supseteq \Sigma_i'$, such that $\Sigma \vdash \Sigma_1''.\ v_1'' \sim \Sigma_2''.\ v_2'' : A'$. By definition, $\Sigma \vdash \Sigma_1''.\ [v_1''/x]e_1 \cong \Sigma_2''.\ [v_2''/x]e_2 : A''$, simply because $\Sigma_i'' \supseteq \Sigma_i' \supseteq \Sigma_i$. □

Lemma 18 (*Symmetry and transitivity*)

1. If $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma \vdash \Sigma_2.\ e_2 \cong \Sigma_1.\ e_1 : A\,[C]$.
2. If $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, then $\Sigma \vdash \Sigma_2.\ v_2 \sim \Sigma_1.\ v_1 : A$.
3. If $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, and $\Sigma \vdash \Sigma_2.\ e_2 \cong \Sigma_3.\ e_3 : A\,[C]$, then $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_3.\ e_3 : A\,[C]$
4. If $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, and $\Sigma \vdash \Sigma_2.\ v_2 \sim \Sigma_3.\ v_3 : A$, then $\Sigma \vdash \Sigma_1.\ e_1 \sim \Sigma_3.\ v_3 : A$

Proof

Symmetry is obvious, so we present the proofs for transitivity. The proofs are by induction on the definition of the judgments. For transitivity of the relation on expressions, we only consider the case when the supports C_i are empty, as it is easy to generalize to the case of non-empty supports.

By assumptions, $(\Sigma, \Sigma_1), e_1 \longmapsto (\Sigma, \Psi_1), v_1$, and $(\Sigma, \Sigma_2), e_2 \longmapsto (\Sigma, \Psi_2), v_2$, such that $\Sigma \vdash \Psi_1.\ v_1 \sim \Psi_2.\ v_2 : A$. Also, $(\Sigma, \Sigma_2), e_2 \longmapsto (\Sigma, \Psi_2'), v_2'$, and, $(\Sigma, \Sigma_3), e_3 \longmapsto (\Sigma, \Psi_3), v_3$, such that $\Sigma \vdash \Psi_2'.\ v_2' \sim \Psi_3.\ v_3 : A$.

By determinacy of evaluation, we know that there is a permutation of names π such that $\Psi_2 = \pi(\Psi_2')$ and $v_2 = \pi(v_2')$, and thus by Lemma 15, $\Sigma \vdash \Psi_2.\ v_2 \sim \Psi_3.\ v_3 : A$. Then, by the last induction hypothesis, $\Sigma \vdash \Psi_1.\ v_1 \sim \Psi_3.\ v_3 : A$, and therefore, $\Sigma \vdash \Sigma_1.\ e_1 \sim \Sigma_3.\ e_3 : A$.

For the relation on values, we only present the case $A = A_1 \to A_2$ and $v_i = \lambda x{:}A_1.\ e_i$. In this case, let $\Sigma_1' \supseteq \Sigma_1$ and $\Sigma_3' \supseteq \Sigma_3$, such that $\Sigma \vdash \Sigma_1'.\ v_1' \sim \Sigma_3'.\ v_3' : A_1$. By name permutation, we can assume that Σ_3' and Σ_2 are disjoint; otherwise, we can just rename the conflicting names in Σ_2. By symmetry and transitivity at type A_1, we obtain $\Sigma \vdash \Sigma_3'.\ v_3' \sim \Sigma_3'.\ v_3' : A_1$. By weakening, $\Sigma \vdash \Sigma_1'.\ v_1' \sim \Sigma_2, \Sigma_3'.\ v_3'$ and $\Sigma \vdash \Sigma_2, \Sigma_3'.\ v_3' \sim \Sigma_3'.\ v_3'$; therefore $\Sigma \vdash \Sigma_1'.\ [v_1'/x]e_1 \cong (\Sigma_2, \Sigma_3').\ [v_3'/x]e_2 : A_2$ and $\Sigma \vdash (\Sigma_2, \Sigma_3').\ [v_3'/x]e_2 \cong \Sigma_3'.\ [v_3'/x]e_3 : A_2$. Finally, by first induction hypothesis at type A_2, we get $\Sigma \vdash \Sigma_1'.\ [v_1'/x]e_1 \cong \Sigma_3'.\ [v_3'/x]e_3 : A_2$. □

It is simple now to extend the above results to logical relations over expressions with free variables. The following lemma restates the relevant properties.

Lemma 19

1. (*Name permutation*) Let $R_1 : \Sigma_1 \to \Sigma_1'$ and $R_2 : \Sigma_2 \to \Sigma_2'$ be bijections where Σ_1' and Σ_2' are well-formed in Σ. If $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash \Sigma_1'.\ R_1\ e_1 \cong \Sigma_2'.\ R_2\ e_2 : A\,[C]$.
2. *(Name localization)* Let Δ, Γ, A, C are well-formed in Σ. Then $(\Sigma, \Sigma'); \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$ if and only if $\Sigma; \Delta; \Gamma \vdash (\Sigma', \Sigma_1).\ e_1 \cong (\Sigma', \Sigma_2).\ e_2 : A\,[C]$.
3. *(Weakening)* Let $\Sigma' \supseteq \Sigma$, and $\Sigma_1' \supseteq \Sigma_1$, $\Sigma_2' \supseteq \Sigma_2$, $\Delta' \supseteq \Delta$, $\Gamma' \supseteq \Gamma$ and $C' \supseteq C$, so that $\Sigma_1', \Sigma_2', \Delta'$, Γ' and C' are well-formed with respect to Σ'. If $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma'; \Delta'; \Gamma' \vdash \Sigma_1'.\ e_1 \cong \Sigma_2'.\ e_2 : A\,[C']$.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

4. *(Symmetry)* If $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_2.\ e_2 \cong \Sigma_1.\ e_1 : A\,[C]$.
5. *(Transitivity)* If $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, and $\Sigma;\Delta;\Gamma \vdash \Sigma_2.\ e_2 \cong \Sigma_3.\ e_3 : A\,[C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_3.\ e_3 : A\,[C]$

Proof
The proofs proceed in a straightforward manner, following the definition of the judgment on open expressions. First we consider the case when Γ is non-empty, but both C and Δ are empty. Then we generalize to the case of non-empty C, before finally a non-empty context Δ is considered. Just as in the definition of the logical relations, it is easy to check that in each step of the proof we only rely on the previously established results. □

To complete the logical relations argument, we need to define the notion of extensional relation on the remaining syntactic category of $v^{\Box}$ – the category of explicit substitutions. This definition will be utilized in the statement and the proof of Lemma 20 to establish that term constructors of $v^{\Box}$ (in particular, the constructs for explicit substitutions and modal variables) preserve extensional equivalence.

The judgment for logical relation of extensional equivalence between two explicit substitutions Θ_1 and Θ_2 has the form

$$\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [C] \Rightarrow [D]$$

and is defined by the following clauses:

$\Sigma;\cdot;\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [C] \Rightarrow [D]$ iff $\Sigma;\cdot;\Gamma \vdash \Sigma_1'.\ \{\Theta_1\}e_1 \cong \Sigma_2'.\ \{\Theta_2\}e_2 : A\,[D]$, for any $\Sigma_i' \supseteq \Sigma_i$, such that $\Sigma;\cdot;\Gamma \vdash \Sigma_1'.\ e_1 \cong \Sigma_2'.\ e_2 : A\,[C]$

$\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [C] \Rightarrow [D]$ iff $\Sigma;\cdot;\Gamma \vdash \Sigma_1'.\ \langle[\delta_1/\Delta]\Theta_1\rangle \cong \Sigma_2'.\ \langle[\delta_2/\Delta]\Theta_2\rangle : [C] \Rightarrow [D]$ for any $\Sigma_i' \supseteq \Sigma_i$, such that $\Sigma \vdash \Sigma_1'.\ \delta_1 = \Sigma_2'.\ \delta_2 : \Delta$

As in the case of previous judgments, the relation $\cong$ on explicit substitutions satisfies the properties of name permutation, name localization, weakening, symmetry and transitivity.

Lemma 20
Logical relation is preserved by all the expression constructors of $v^{\Box}$, except box. The constructor box has a stronger requirement that the expressions are related intensionally. More precisely:

1. $(\Sigma, X{:}A);\Delta;\Gamma \vdash \Sigma_1.\ X \cong \Sigma_2.\ X : A\,[X, C]$
2. $\Sigma;\Delta;(\Gamma, x{:}A) \vdash \Sigma_1.\ x \cong \Sigma_2.\ x : A\,[C]$
3. if $\Sigma;(\Delta, u{:}A[D]);\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [D] \Rightarrow [C]$, then $\Sigma;(\Delta, u{:}A[D]);\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle u \cong \Sigma_2.\ \langle\Theta_2\rangle u : A\,[C]$
4. if $\Sigma;\Delta;(\Gamma, x{:}A) \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : B\,[C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \lambda x{:}A.\ e_1 \cong \Sigma_2.\ \lambda x{:}A.\ e_2 : A \to B\,[C]$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

5. if $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A \to B\ [C]$ and $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1' \cong \Sigma_2.\ e_2' : A\ [C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1\ e_1' \cong \Sigma_2.\ e_2\ e_2' : B\ [C]$
6. If $\Sigma;\Delta \vdash \Sigma_1.\ e_1 = \Sigma_2.\ e_2 : A\ [C]$, and $\Sigma;\Delta;\cdot \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\ [C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{box}\ e_1 \cong \Sigma_2.\ \mathbf{box}\ e_2 : \Box_C A\ [D]$
7. if $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : \Box_D A\ [C]$ and $\Sigma;(\Delta, u{:}A[D]);\Gamma \vdash \Sigma_1.\ e_1' \cong \Sigma_2.\ e_2' : B\ [C]$, then
   $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_1' \cong \Sigma_2.\ \mathbf{let\ box}\ u = e_2\ \mathbf{in}\ e_2' : B\ [C]$
8. if $\Sigma;\Delta;\Gamma \vdash (\Sigma_1, X{:}A).\ e_1 \cong (\Sigma_2, X{:}A).\ e_2 : B\ [C]$, then
   $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ vX{:}A.\ e_1 \cong \Sigma_2.\ vX{:}A.\ e_2 : A \nrightarrow B\ [C]$
9. if $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A \nrightarrow B\ [C]$ then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{choose}\ e_1 \cong \Sigma_2.\ \mathbf{choose}\ e_2 : B\ [C]$
10. $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\,\rangle \cong \Sigma_2.\ \langle\,\rangle : [C] \Rightarrow [D]$ if $C \subseteq D$
11. if $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\ [D]$, and $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [C \setminus X] \Rightarrow [D]$, and $X{:}A \in \Sigma$, then
    $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle X \to e_1, \Theta_1\rangle \cong \Sigma_2.\ \langle X \to e_2, \Theta_2\rangle : [C] \Rightarrow [D]$

Proof

To reduce clutter, we just present the selected cases as if the contexts Δ, Γ and the support C were empty. The general results are recovered by considering the interaction between value substitutions ρ, explicit substitutions σ and modal substitutions δ, which is well-behaved in all the cases of the lemma.

In case of (3), consider $\Sigma_i' \supseteq \Sigma_i$ such that $e_1 = e_2$, and $\Sigma \vdash \Sigma_1'.\ e_1 \cong \Sigma_2'.\ e_2 : A\ [D]$. We need to show that $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \{[\![e_1/u]\!]\Theta_1\}e_1 \cong \Sigma_2'.\ \{[\![e_2/u]\!]\Theta_2\}e_2 : A\ [\,]$. From the assumption, we have $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \langle[\![e_1/u]\!]\Theta_1\rangle \cong \Sigma_2'.\ \langle[\![e_2/u]\!]\Theta_2\rangle : [D] \Rightarrow [\,]$, and then the required equality follows by definition of extensional equivalence for explicit substitutions

In case of (7), by equivalence of e_1 and e_2, there exist name sets Ψ_1, Ψ_2, such that $(\Sigma, \Sigma_1), e_1 \longmapsto^* (\Sigma, \Psi_1), \mathbf{box}\ t_1$ and $(\Sigma, \Sigma_2), e_2 \longmapsto^* (\Sigma, \Psi_2), \mathbf{box}\ t_2$, where $t_1 = t_2 : A\ [D]$, and $\Sigma \vdash \Psi_1.\ t_1 \cong \Psi_2.\ t_2 : A\ [D]$. Then it suffices to show that $\Sigma;\cdot;\cdot \vdash \Psi_1.\ [\![t_1/u]\!]e_1' \cong \Psi_2.\ [\![t_2/u]\!]e_2' : B\ [\,]$. But this follows from the second assumption, by definition of extensional equivalence.

In case of (11), again consider $\Sigma_i' \supseteq \Sigma_i$, such that $\Sigma';\cdot;\cdot \vdash \Sigma_1'.\ e_1' \cong \Sigma_2'.\ e_2' : B\ [C]$. To be consistent with the notation, in this case we assume that D, rather than C, is empty. To reduce clutter, denote by σ_1, σ_2 the explicit substitutions $\sigma_1 = \langle X \to e_1, \Theta_1\rangle$ and and $\sigma_2 = \langle X \to e_2, \Theta_2\rangle$. Then we need to show that $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \{\sigma_1\}e_1' \cong \Sigma_2'.\ \{\sigma_2\}e_2' : B\ [\,]$. To establish this, it suffices to prove that $\Sigma;\cdot \vdash \Sigma_1'.\ \sigma_1 \cong \Sigma_2'.\ \sigma_2\ [C]$, i.e., that $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \sigma_1(Z) \cong \Sigma_2'.\ \sigma_2(Z) : A'\ [\,]$ for any name $Z \in C$ such that $Z{:}A' \in \Sigma$. Then the result would follow from the extensional equivalence of e_1' and e_2'. We consider two cases: $Z = X$, and $Z \in C \setminus X$. If $Z = X$, then $A' = A$ and $\sigma_i(Z) = e_i$ and by first assumption, $\Sigma;\cdot;\cdot \vdash \Sigma_1.\ \sigma_1(Z) \cong \Sigma_2.\ \sigma_2(Z) : A$. By weakening, this implies $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \sigma_1(Z) \cong \Sigma_2'.\ \sigma_2(Z) : A$. If $Z \in C \setminus X$, then $\sigma_i(Z) = \{\Theta_i\}Z$, and also obviously $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ Z \cong \Sigma_2'.\ Z : A'\ [C \setminus X]$. Then by the second assumption, $\Sigma;\cdot;\cdot \vdash \Sigma_1'.\ \sigma_1(Z) \cong \Sigma_2'.\ \sigma_2(Z) : A'\ [\,]$. The two cases combined demonstrate $\Sigma;\cdot \vdash \Sigma_1'.\ \sigma_1 \cong \Sigma_2'.\ \sigma_2\ [C]$, and this completes the proof. □

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Lemma 21 (*Reflexivity*)

1. If $\Sigma;\Delta;\Gamma \vdash e : A\,[C]$, then $\Sigma;\Delta;\Gamma \vdash e \cong e : A\,[C]$
2. If $\Sigma;\Delta;\Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, then $\Sigma;\Delta;\Gamma \vdash \langle\Theta\rangle \cong \langle\Theta\rangle : [C] \Rightarrow [D]$

Proof
By induction on the structure of e and Θ, using Lemma 20. □

The lemma has several more interesting consequences. As a first observation, it shws that the $v^{\Box}$-calculus, as considered in this section (i.e., with no recursion), is *terminating*. Indeed, our definition of logical relations on expressions required that related expressions evaluate to related values. Thus, if a well-typed expressions of the calculus is related to itself, than it must have a value.

The second consequence of the lemma is that intensionally related expressions are at the same time extensionally related as well. In other words, if $\Sigma;\Delta \vdash \Sigma_1.\ e_1 = \Sigma_2.\ e_2 : A\,[C]$, where e is a simple term, then $\Sigma;\Delta;\cdot \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$. This property trivially follows from the reflexivity, simply because the intensional equivalence, as defined on closed simple terms equates two terms if and only if they are the same (up to α-renaming) and – more importantly – well-typed. Then the reflexivity lemma can be applied As a result, extensional equivalence of modal expressions **box** e_1 and **box** e_2 need not compare e_1 and e_2 for extensional equivalence (as it is required by the definition), but can only rely on their intensional equivalence. This is important, as intensional equivalence, contrary to the extensional one, is defined inductively, and can be carried out as an algorithm.

Lemma 22 (*Fundamental property of logical relations*)
If $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then

1. if $\Sigma;\Delta;(\Gamma, x{:}A) \vdash e : B\,[C]$, then $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ [e_1/x]e \cong \Sigma_2.\ [e_2/x]e : B\,[C]$
2. if $\Sigma;\Delta;(\Gamma, x{:}A) \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C]$, then
   $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle[e_1/x]\Theta\rangle \cong \Sigma_2.\ \langle[e_2/x]\Theta\rangle : [C_1] \Rightarrow [C]$

Proof
By straightforward simultaneous induction on the structure of the two typing derivations, using the fact that the term constructors of the language preserve the logical relation. □

After developing the theory of the two relations, we will use it to prove some interesting equivalences in the calculus. But before we do that in the next lemma, let us remark on an important property of the our presentation. If we dropped the requirement of intensional equivalence when comparing values of modal types that would correspond to treating modal values extensionally, rather than intensionally. In fact, that may be a more relevant approach for this paper, as in the current presentation of $v^{\Box}$ we do not consider any constructs for structural analysis of modal expressions. In this case, we do not have to limit the modal expressions to only simple expressions.

Finally, the next lemma lists some equivalences which hold in $v^{\Box}$ (irrespective of the treatment of modal values as intensional or extensional entities). Observe that the list includes all the β-reductions and η-expansions of $v^{\Box}$. In this sense, we can claim that the calculus presented in this paper is purely functional.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

Lemma 23
In the logical equivalences below we assume that all the judgments are well-formed and that the terms are well-typed in appropriate contexts.

1. $\Sigma;\Delta;\Gamma \vdash (\lambda x.\ e_1)\ e_2 \cong [e_2/x]e_1 : A\ [C]$
2. $\Sigma;\Delta;\Gamma \vdash e \cong \lambda x.\ (e\ x) : A \to B\ [C]$
3. $\Sigma;\Delta;\Gamma \vdash \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 \cong [\![e_1/u]\!]e_2 : B\ [C]$
4. $\Sigma;\Delta;\Gamma \vdash e \cong \mathbf{let\ box}\ u = e\ \mathbf{in\ box}\ u : \Box_D B\ [C]$
5. $\Sigma;\Delta;\Gamma \vdash \mathbf{choose}\ (\nu X{:}A.\ e) \cong (X{:}A).\ e : B\ [C]$
6. $\Sigma;\Delta;\Gamma \vdash (X{:}A).\ e \cong \nu X{:}A.\ \mathbf{choose}\ e : A \nrightarrow B\ [C]$
7. $\Sigma;\Delta;\Gamma \vdash \lambda z{:}A.\ \mathbf{choose}\ (\nu X{:}A_1.\ e) \cong \mathbf{choose}\ (\nu X{:}A_1.\ \lambda z{:}A.\ e) : A \to B\ [C]$
8. $\Sigma;\Delta;\Gamma \vdash \nu X.\ \nu Y.\ e \cong \nu Y.\ \nu X.\ e : A \nrightarrow A \nrightarrow B\ [C]$
9. $\Sigma;\Delta;\Gamma \vdash e_1\ (\mathbf{choose}\ (\nu X{:}A.\ e_2)) \cong \mathbf{choose}\ (\nu X{:}A.\ (e_1\ e_2)) : B\ [C]$
10. $\Sigma;\Delta;\Gamma \vdash (\mathbf{choose}\ (\nu X{:}A.\ e_1))\ e_2 \cong \mathbf{choose}\ (\nu X{:}A.\ (e_1\ e_2)) : B\ [C]$

Proof
Again, in order to reduce clutter, we present the proofs of these statements in the case when Δ, Γ, C are empty. In the general cases, we need to consider interactions between value substitutions ρ, explicit substitutions σ and modal substitutions δ, but these pose no problems.

In the case Δ, Γ and C are empty, the statements (3) and (4) are trivial, as the two expressions evaluate to the same value. In (5), the expressions evaluate to the same value, modulo the choice of a local name Y to stand for X in $\mathbf{choose}\ (\nu X{:}A.\ e)$. But this choice is irrelevant, by the name permutation property. The statement (10) is completely symmetric to (9).

To establish (1), let $\Sigma;\cdot;x{:}B \vdash e_1 : A$, and $\Sigma;\cdot;\cdot \vdash e_2 : B$. As the calculus is terminating, there exist Ψ and v_2 such that $\Sigma, e_2 \longmapsto^* (\Sigma,\Psi), v_2$, and therefore also $\Sigma \vdash e_2 \cong \Psi.\ v_2 : B$. By the fundamental property of logical relations (Lemma 22), $\Sigma \vdash [e_2/x]e_1 \cong \Psi.\ [v_2/x]e_1 : A$. But it is also the case that $\Sigma \vdash (\lambda x.\ e_1)\ e_2 \cong \Psi.\ [v_2/x]e_1 : A$, simply because the two expressions evaluate to the same value. Then by transitivity, we get $\Sigma \vdash (\lambda x.\ e_1)\ e_2 \cong [e_2/x]e_1 : A$.

To establish (2), let $\Sigma, e \longmapsto^* (\Sigma,\Psi), (\lambda x.\ e')$, so that $\Sigma;\cdot;\cdot \vdash e \cong \Psi.\ (\lambda x.\ e') : A \to B$. By transitivity, this holds if $\Sigma \vdash \Psi.\ \lambda x.\ e' \sim \lambda x.\ (e\ x) : A \to B$. To prove this, consider Σ_1', Σ_2' such that $\Sigma \vdash \Psi, \Sigma_1'.\ v_1 \sim \Sigma_2'.\ v_2 : A$. It suffices to show $\Sigma \vdash (\Psi, \Sigma_1').\ [v_1/x]e' \cong \Sigma_2'.\ (e\ v_2) : B$. By the name permutation property (Lemma 15), we can assume that Ψ and Σ_2 are disjoint. By the properties of evaluation, $(\Sigma', \Sigma_2'), (e\ v_2) \longmapsto^* (\Sigma', \Sigma_2', \Psi), [v_2/x]e'$, and thus

$$\Sigma \vdash \Sigma_2'.\ (e\ v_2) \cong (\Psi, \Sigma_2').\ [v_2/x]e' \qquad (*)$$

By type preservation, $(\Sigma,\Psi);\cdot;x{:}A \vdash e' : B\ [\ ]$, and thus by reflexivity $\Sigma;\cdot;x{:}A \vdash \Psi.\ e' \cong \Psi.\ e' : B\ [\ ]$. Then by definition,

$$\Sigma \vdash (\Psi, \Sigma_1').\ [v_1/x]e' \cong (\Psi, \Sigma_2').\ [v_2/x]e' : B \qquad (**)$$

Finally, from (*) and (**), by transitivity, we obtain the required

$$\Sigma \vdash (\Psi, \Sigma_1').\ [v_1/x]e' \cong \Sigma_2'.\ (e\ v_2) : B.$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

To establish (6), let $(\Sigma, X{:}A), e \longmapsto (\Sigma, X{:}A, \Psi), (\nu Y{:}A.\ e')$. Then, by definition, we have $\Sigma \vdash (X{:}A).\ e \cong (X{:}A, \Psi).\ (\nu Y{:}A.\ e') : A \nrightarrow B$. By transitivity, it suffices to show that $\Sigma \vdash (X{:}A, \Psi).\ \nu Y{:}A.\ e' \sim \nu X{:}A.\ \mathbf{choose}\ e : A \nrightarrow B$

By definition of the logical relation for values at the type $A \nrightarrow B$, this holds if and only if $\Sigma \vdash (X{:}A, \Psi, Y{:}A).\ e' \cong X{:}A.\ \mathbf{choose}\ e : B$. Indeed, we could chose $X{:}A$ in the local context of the second argument by the name permutation property. But the last equation is obviously true, as $(\Sigma, X{:}A), \mathbf{choose}\ e \longmapsto^* (\Sigma, X{:}A, \Psi), \mathbf{choose}\ (\nu Y{:}A.\ e') \longmapsto (\Sigma, X{:}A, \Psi, Y{:}A), e'$.

For (7), the considered equivalence holds iff $\Sigma \vdash \lambda z{:}A.\ \mathbf{choose}\ (\nu X{:}A_1.\ e) \cong (X{:}A_1).\ \lambda z{:}A.\ e : A \to B$, iff $\Sigma; \cdot; z{:}A \vdash \mathbf{choose}\ (\nu X{:}A_1.\ e) \cong (X{:}A_1).\ e : B$. But this is true by (6).

To establish (8), notice that by definition, the required equivalence holds if and only if $\Sigma \vdash (X{:}A, Y{:}A).\ e \cong (Y{:}A, X{:}A).\ e : B$. In this equation, we are justified in choosing the same names X and Y in both sides, by the name permutation property (Lemma 15). But the contexts $(X{:}A, Y{:}A)$ and $(Y{:}A, X{:}A)$ are same, because the type A does not depend on neither X nor Y. Thus, the result follows by reflexivity of $\cong$.

To establish (9), it suffices to show that $\Sigma \vdash e_1 \cong (X{:}A).\ e_1 : B' \to B$ and that $\Sigma \vdash \mathbf{choose}\ (\nu X{:}A.\ e_2) \cong (X{:}A).\ e_2 : B'$. Then the result would be implied by the fact that term constructors preserve the equivalence. The first of the above equivalences follows by reflexivity and weakening. The second has already been established as the β-reduction for the type $A \nrightarrow B'$. □

The developed logical relations analyze the equivalence of terms from the outside, rather than by considering their observable operational behavior. A more general notion of equivalence is the *contextual equivalence*, by which two terms e_1 and e_2 are related if and only if any observable behavior produced by a use of e_1 in a complete program is also produced by a use of e_2, and vice versa.

Logical relations, however, are related to contextual equivalence in the following sense: whenever two terms are logically equated, their behavior in any program context is indiscernible. In other words, logical equivalence is sound with respect to the contextual equivalence. We establish this result in the remainder of the section. The opposite direction of this implication, that is, the completeness of the logical relations with respect to contextual equivalence remains future work.

We start by formalizing what it means to use an expression in a program. For that reason, we define two notions of program contexts: a notion of expression contexts, and a notion of substitution context. An *expression context* (resp. substitution context) is an expression $\mathscr{E}$ (substitution $\mathscr{F}$) with a hole, where the whole can be filled with some expression. We write $\mathscr{E}[e]$ ($\mathscr{F}[e]$) for the expression (substitution) obtained when the hole of $\mathscr{E}$ is filled with e. Furthermore, we consider only contexts that are *extensional*, i.e. whose hole does not appear under a box.

A more formal definition of extensional expression and substitution contexts is given in the table below.

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

$$\begin{array}{lrll} \textit{Extensional expression contexts} & \mathscr{E} & ::= & [\,] \mid X \mid x \mid \langle \mathscr{F} \rangle u \mid \lambda x{:}A.\, \mathscr{E} \mid \mathscr{E}_1\, \mathscr{E}_2 \mid \\ & & & \mathbf{box}\ e \mid \mathbf{let\ box}\ u = \mathscr{E}_1\ \mathbf{in}\ \mathscr{E}_2 \mid \\ & & & \nu X{:}A.\, \mathscr{E} \mid \mathbf{choose}\ \mathscr{E} \\ \textit{Extensional substitution contexts} & \mathscr{F} & ::= & \cdot \mid X \to \mathscr{E}, \mathscr{F} \end{array}$$

The decision to restrict extensional contexts so that the hole does not appear under box deserves further explanation: we do this in order to distinguish extensional contextual equivalence from the related notion of intensional contextual equivalence. Intensional contextual equivalence studies the behavior of terms in a language with intensional operations on the syntactic object-level terms – operations like syntactic comparison or pattern-matching against a syntactic term. The associated notion of intensional context would permit the hole to appear in a scope of a box, and allow the intensional operations to act on the boxed hole.

The two contextual relations are obviously different. In a language with intensional operations, conflating them leads to unsoundness, as has already been observed in the case of MetaML in (Taha, 2000). Indeed, two expressions that should be extensionally equal, like a function application and its β-reduction, cannot be considered intensionally equal because they do have observably different shapes. However, both contextual relations can be defined and studied; all it takes is to appropriately restrict extensional contexts so that the context hole does not appear in the scope of a box. This is justified because box turns intensionally related expressions into extensionally related ones (as shown in Lemma 20), but does not necessarily preserve the extensional relation itself.

We return now to our development of extensional equivalence, and prove that the extensional relation on expressions and substitutions, as defined previously, is a congruence with respect to extensional contexts.

Lemma 24 (*Congruence*)
If $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, and $\mathscr{E}$, $\mathscr{F}$ are an expression and substitution context respectively, then the following holds.

1. $\Sigma';\Delta';\Gamma' \vdash \Sigma'_1.\ \mathscr{E}[e_1] \cong \Sigma'_2.\ \mathscr{E}[e_2] : B\,[D]$, if $\mathscr{E}[e_1]$, $\mathscr{E}[e_2]$ are well-typed in their appropriate variable contexts.
2. $\Sigma';\Delta';\Gamma' \vdash \Sigma'_1.\ \langle \mathscr{F}[e_1] \rangle \cong \Sigma'_2.\ \langle \mathscr{F}[e_2] \rangle : [D] \Rightarrow [D']$, if $\mathscr{F}[e_1]$, $\mathscr{F}[e_2]$ are well-typed in their appropriate variable contexts.

Proof
By straightforward simultaneous induction on the structure of $\mathscr{E}$ and $\mathscr{F}$, using Lemma 20. □

The use of an expression in a complete program context of base type defines the contextual equivalence between expressions in the following way.

Definition 25 (*Extensional contextual equivalence*)
Let e_1, e_2 be well-typed expressions such that $\Sigma, \Sigma_1;\Delta;\Gamma \vdash e_1 : A\,[C]$, and $\Sigma, \Sigma_2;\Delta;\Gamma \vdash e_2 : A\,[C]$, where Σ_i are local to e_i. Then e_1 and e_2 are *contextually equivalent*, written

$$\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong_{\mathrm{ctx}} \Sigma_2.\ e_2 : A\,[C]$$

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

if and only if for every extensional expression context $\mathscr{E}$ such that $\vdash \mathscr{E}[e_1] : b$ and $\vdash \mathscr{E}[e_2] : b$, we have

$$\mathscr{E}[e_1] \longmapsto^* v \qquad \text{iff} \qquad \mathscr{E}[e_2] \longmapsto^* v.$$

It is trivial to show that the defined relation is indeed an equivalence. We can now proceed to establish the soundness of the logical relations with respect to contextual equivalence, as we only need to restrict the attention to program contexts of base types.

Lemma 26
If $\Sigma; \Delta; \Gamma \vdash e_1 \cong e_2 : A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash e_1 \cong_{\text{ctx}} e_2 : A\,[C]$.

Proof
By the congruence property of $\cong$ (Lemma 24), for any well-typed extensional context $\mathscr{E}$, we have that $\mathscr{E}[e_1] \cong \mathscr{E}[e_2]$. In the special case when $\mathscr{E}[e_i]$ are closed and of base type b, the relation $\vdash \mathscr{E}[e_1] \cong \mathscr{E}[e_2] : b$ by definition implies that $\mathscr{E}[e_1]$ and $\mathscr{E}[e_2]$ evaluate to the same value. Because $\mathscr{E}$ is chosen arbitrarily, the expressions e_1 and e_2 are contextually equivalent. □

6 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, metaprogramming, modal logic and higher-order abstract syntax.

An early reference to staged computation is Ershov (1977), which introduces 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 others (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 (Nielson & Nielson, 1988; Lee & Leone, 1996; Wickline *et al.*, 1998b), and we described it in section 2.

A significant amount of work on functional metaprogramming today is related to the development of MetaML (Taha & Sheard, 1997; Moggi *et al.*, 1999; Taha, 1999; Taha, 2000) and its variant MetaOCaml (Calcagno *et al.*, 2003b; Taha & Nielsen, 2003), which are themselves extensions of the the $\lambda^{\bigcirc}$-calculus. Formulated by (Davies, 1996), $\lambda^{\bigcirc}$ features a type constructor $\bigcirc$ that classifies open object code. 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 metaprogramming through the development of MetaML.

MetaML builds upon 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 the closed code types are those open

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

code expressions that do not contain any free variables from the present stage. If an expression is typed as a closed code, then it may be evaluated.

It might be of interest here to point out a certain similarity between our concept of supports and the dead-code annotations used in MetaML with references (Calcagno *et al.*, 2003a). MetaML cannot naively allow references to open code, to avoid the extrusion of scope of bound variables. At the same time, limiting references to closed code types is too restrictive because it disallows references to functions. Thus, scope extrusion has to be allowed, but only if the extruding variables are never encountered during evaluation. As a solution, MetaML with references annotates each term with the list of free variables that the term is allowed to contain in dead-code positions.

In contrast to MetaML, in the $\nu^{\Box}$-calculus, free variables are represented by names, and they are built into the calculus from the beginning. As a consequence, only one modal constructor suffices to classify both closed code and code with free variables, leading to a conceptually simpler type system. Furthermore, we do not foresee any significant problems in the extension of $\nu^{\Box}$ with references.

The approach of MetaOCaml to the problem of combining closed and open code is based on *environment classifiers* (Taha & Nielsen, 2003). MetaOCaml has also been extended with type inference for a relatively expressive subset of its type system in Calcagno *et al.* (2004). Intuitively, environment classifiers serve as labels for various object stages of computation; because the stages are labeled, 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 the sense that 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. Syntactically, this implicitness may allow for a more compact programming idiom. For example, the exponentiation function from section 3, can be written in MetaOCaml as shown below. In this example we paraphrase the syntax of MetaOCaml as follows. The type constructor `('a,'b)code` classifies object code of an arbitrary stage `'a` and type `'b`. Here `'a` is an environment classifier. The constructors `.<` and `>.` enclose object code, `.~` splices code into a larger context, and `.!` explicitly evaluates object code.

```
fun exp (n:int) (x:('a, int)code) : ('a, int) code =
   if n = 0 then .<1>. else .< .~ x * .~(exp (n-1) x)>.
```

When applied to an argument 2, the function `exp` generates an object-level code for squaring, which can then be explicitly evaluated using the constructor `.!` to obtain a function for squaring.

```
- sqmeta = .<λ x. ~(exp 2 .<x>.)>.
val sqmeta : .<λx. (x * (x * 1))>. : ('a, int -> int) code
- sq = .! sqcode
val sq = [fn] : int -> int
```

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

In contrast to `.!` from MetaOCaml (which is called `run` in MetaML), the $\nu^{\Box}$-calculus does not need any special constructors for code evaluation. For example, if $e : \Box A$, then a $\nu^{\Box}$ program for evaluating e can be written simply as `let box u = e in u`. In this sense, $\nu^{\Box}$ is more strongly grounded in logic. On the other hand, the explicit support annotations and support polymorphism of $\nu^{\Box}$ certainly make it much more verbose than MetaOCaml, and we plan to address these issues in future work on type and support inference. However, we believe that the current explicitness of type annotations may also prove beneficial. For example, explicit support polymorphism, in addition to the type polymorphism, seems essential for metaprogramming languages that allow recursion over syntactic object expressions. Such recursion ought to be name-polymorphic (and will thus require explicit term constructors for abstraction over sets of names), because scanning past variable binders will have to generate new names to be used as placeholders for these variables.

Coming from the direction of higher-order abstract syntax, probably the first work pointing to the importance of binders like 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). The system presented in Bjørner (1999) is capable of pattern-matching against object-level programs, but is not concerned with their evaluation. 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 inclusion of names in the calculus 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).

On the logical side, the most direct influence 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) andSmoryński (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 & Mendolsohn (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).

https://doi.org/10.1017/S095679680500568X Published online by Cambridge University Press

7 Conclusions and future work

This paper presents the $\nu^{\Box}$-calculus, which is a typed functional language for metaprogramming, employing a novel way to define a modal type of object expressions 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 metaprogramming to handle object programs with free variables (Davies, 1996; Taha & Sheard, 1997; Taha, 1999; Moggi *et al.*, 1999). In our setup, the $\lambda^{\Box}$-calculus provides a way to encode closed object expressions, and names serve to stand for possibly free variables. Names can be operationally thought of as locations that are tracked by the type system, so that a name cannot escape the scope of its 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.

Names in the $\nu^{\Box}$-calculus are second-class objects, and it is an important future work to consider extensions with first-class names and name equality. For example, it may be possible to define a new type constructor

$$\mathbf{N} : \mathit{Type} \rightarrow \mathit{Type},$$

so that $\mathbf{N}(A)$ classifies all the names of type A. The question then becomes how first-class names interact with the modal operators and with explicit substitutions. It is likely that such an extension will require explicit substitutions over (ordinary) variables.

Even when dealing with second-class names, it seems possible that other approaches may be employed for managing dynamic name generation. For example, the variable declaration $u{:}A\,[C]$ may be viewed as *binding* the names listed in C, so that these names have scope local to the explicit substitutions associated to u. This idea has been employed in Nanevski *et al.* (2003) to define a dependently typed calculus for representing metavariables in logical frameworks.

Finally, it is an important future work to investigate embeddings of $\lambda^{\bigcirc}$ and MetaML into $\nu^{\Box}$, in order to formally compare the expressiveness of the three metaprogramming systems. An interesting step in this direction may be to consider the proof-irrelevance of Pfenning (2001) and Awodey & B auer (2001), as a way to represent cross-stage persistence of MetaML.

Acknowledgements

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

References

Attardi, G. and Simi, M. (1995) A formalization of viewpoints. *Fundamenta informaticae*, **23**(3), 149–173.

Awodey, S. and Bauer, A. (2001) *Propositions as [Types]*. Technical report IML-R–34-00/01–SE. Institut Mittag-Leffler, The Royal Swedish Academy of Sciences.

Benaissa, Z. El-Abidine, Moggi, E., Taha, W. and Sheard, T. (1999) Logical modalities and multi-stage programming. *Workshop on Intuitionistic Modal Logics and Applications, IMLA'99*.

Bjørner, Nikolaj. (1999). Type checking meta programs. *Workshop on Logical Frameworks and Meta-languages*.

Calcagno, C., Moggi, E. and Sheard, T. (2003a) Closed types for a safe imperative MetaML. *J. Funct. Program.* **13**(3), 545–571.

Calcagno, C., Taha, W., Huang, L. and Leroy, X. (2003b) Implementing multi-stage languages using ASTs, gensym, and reflection. *Conference on Generative Programming and Component Engineering, GPCE'03*.

Calcagno, C., Moggi, E. and Taha, W. (2004) ML-like inference for classifiers. *European Symposium on Programming, ESOP'04*, pp. 79–93.

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

Davies, R. and Pfenning, F. (2001) A modal analysis of staged computation. *J. ACM*, **48**(3), 555–604.

Despeyroux, J., Pfenning, F. and Schürmann, C. (1997) Primitive recursion for higher-order abstract syntax. In: de Groote, P. and Hindley, J. R. (eds.), *Typed Lambda Calculi and Applications: LNCS 1210*. Springer-Verlag.

Ershov, A. P. (1977) On the partial computation principle. *Infor. Process. Lett.* **6**(2), 38–41.

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

Fiore, M., Plotkin, G. and Turi, D. (1999) Abstract syntax and variable binding. *Symposium on Logic in Computer Science, LICS'99*, pp. 193–202.

Fitting, M. and Mendelsohn, R. L. (1999) *First-order Modal Logic*. Kluwer.

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

Gabbay, M. J. (2000) *A theory of inductive definitions with α-equivalence*. PhD thesis, Cambridge University.

Gabbay, M. J. and Pitts, A. M. (2002) A new approach to abstract syntax with variable binding. *Formal Aspects of Comput.* **13**, 341–363.

Girard, J.-Y. (1986) The system F of variable types, fifteen years later. *Theor. Comput. Sci.* **45**(2), 159–192.

Glück, R. and Jørgensen, J. (1995) Efficient multi-level generating extensions for program specialization. In: Hermenegildo, M. and Swierstra, S. D. (eds.), *Programming Languages: Implementations, logics and programs: LNCS 982*, pp. 259–278. Springer-Verlag.

Glück, R. and Jørgensen, J. (1997) An automatic program generator for multi-level specialization. *Lisp & Symbolic Computation*, **10**(2), 113–158.

Harper, R. (1999) Proof-directed debugging. *J. Funct. Program.* **9**(4), 463–470.

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

Jones, N. D., Sestoft, P. and Søndergaard, H. (1985) An experiment in partial evaluation: the generation of a compiler generator. In: Jouannaud, J.-P. (ed.), *Rewriting techniques and applications: LNCS 202*, pp. 124–140. Springer-Verlag.

Kripke, S. A. (1980) *Naming and Necessity*. Harvard University Press.

Lee, P. and Leone, M. (1996) Optimizing ML with run-time code generation. *Conference on Programming Language Design and Implementation, PLDI'96*, pp. 137–148.

McCarthy, J. (1993) Notes on formalizing context. *International Joint Conference on Artificial Intelligence, IJCAI'93*, pp. 555–560.

Miller, D. (1990) An extension to ML to handle bound variables in data structures. *Proceedings of the First ESPRIT BRA Workshop on Logical Frameworks*, pp. 323–335.

Moggi, E., Taha, W., Benaissa, Zine-El-Abidine and Sheard, T. (1999) An idealized MetaML: Simpler, and more expressive. *European Symposium on Programming, ESOP'99*, pp. 193–207.

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

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

Nanevski, A., Pientka, B. and Pfenning, F. (2003) A modal foundation for meta variables. *Proceedings of MERλIN'03*.

Nielson, F. and Nielson, H. R. (1988) Two-level semantics and code generation. *Theor. Comput. Sci.* **56**(1), 59–133.

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

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

Pfenning, F. and Davies, R. (2001) A judgmental reconstruction of modal logic. *Math. Struct. in Comput. Sci.* **11**(4), 511–540.

Pitts, A. M. and Stark, I. D. B. (1993) Observable properties of higher order functions that dynamically create local names, or: What's new? *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp.: LNCS 711*, pp. 122–141. Gdańsk, Poland. Springer-Verlag.

Pitts, A. M. (2001) Nominal logic: A first order theory of names and binding. In: Kobayashi, N. and Pierce, B. C. (eds.), *Theoretical Aspects of Computer Software: LNCS 2215*, pp. 219–242. Springer-Verlag.

Pitts, A. M. and Gabbay, M. J. (2000) A metalanguage for programming with bound names modulo renaming. In: Backhouse, R. and Oliveira, J. N. (eds.), *Mathematics of Program Construction: LNCS 1837*, pp. 230–255. Springer-Verlag.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In: Mason, R. E. A. (ed.), *Information Processing '83*, pp. 513–523. Elsevier.

Schürmann, C. (2000) *Automating the meta-theory of deductive systems.* PhD thesis, Carnegie Mellon University.

Scott, D. (1970) Advice on modal logic. In: Lambert, K. (ed.), *Philosophical Problems in Logic*, pp. 143–173. Reidel.

Scott, D. (1979) Identity and existence in intuitionistic logic. In: Fourman, M., Mulvey, C. and Scott, D. (eds.), *Applications of Sheaves: LNCS 753*, pp. 660-696. Springer-Verlag.

Sheard, T. (2001) Accomplishments and research challenges in meta-programming. In: Taha, W. (ed.), *Semantics, Applications, and Implementation of Program Generation: LNCS 2196*, pp. 2–44. Springer-Verlag.

Smoryński, C. (1985) *Self-reference and Modal Logic*. Springer-Verlag.

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

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

Taha, W. and Nielsen, M. F. (2003) Environment classifiers. *Symposium on Principles of Programming Languages, POPL'03*, pp. 26–37.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pp. 203–217.

Wickline, P., Lee, P., Pfenning, F. and Davies, R. (1998a) Modal types as staging specifications for run-time code generation. *ACM Comput. Surv.* **30**(3es).

Wickline, P., Lee, P. and Pfenning, F. (1998b) Run-time code generation and Modal-ML. *Conference on Programming Language Design and Implementation, PLDI'98*, pp. 224–235.