From Dynamic Binding to State via Modal Possibility

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

Abstract

In this paper we propose a typed, purely functional calculus for state (with second-class locations) in which types reflect the dichotomy between reading from and writing into the global store. This is in contrast to the usual formulation of state via monads, where the primitives for reading and writing introduce the same monadic type constructor. We hope to argue that making this distinction is useful, simple, and has strong logical foundations.

Our type system is based on the proof-term calculus for constructive modal logic S4, which has two modal type operators: $\Box$ for necessity and $\Diamond$ for possibility. We extend this calculus with the notion of names (which stand for locations) and generalize to indexed families of modal operators (indexed by sets of names). Then, the modal type $\Box_C A$ classifies computations of type A which read from store locations listed in the set C. The dual type $\Diamond_C A$ classifies computations which first write into the locations from C and than use the changed store to obtain a value of type A.

There are several benefits to this development. First, the necessitation fragment of the language is interesting in its own: it formulates a calculus of dynamic binding. Second, the possibility operator $\Diamond$ is a monad, thus forcing the single-threading of memory writes, but not of memory reads (as these are associated with $\Box$). Finally, the different status of reads and writes gives rise to a natural way of expressing the allocation of uninitialized memory while also providing guarantees that only initialized locations are dereferenced.

Categories and Subject Descriptors

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

General Terms

Languages

PPDP'03, August 27-29, 2003, Uppsala, Sweden

Keywords

modal lambda-calculus, effect systems, dynamic binding, state

1 Introduction

Dynamic binding in functional programming languages is a concept by which the value of a certain variable is not fixed statically at the time the variable is introduced, but is determined dynamically from the current scope, each time the variable is used.

This characterization makes dynamically bound variables very similar to memory locations in an appropriate definition of store. Just like dynamic variables, the store locations can be changed arbitrary number of times, and each dereferencing of a location will return the most recent value.

There is, however, a difference between the two. When a store location is changed, that change is supposed to have global scope; it "holds" until the end of the program, or at least until something else is written into the location. When a dynamic variable is changed, that change has only local scope. Once this scope is exited, the old value of the dynamic variable is restored. In this sense, a location in the global store can be described as a dynamically bound variable which is updated by assignments whose scope can never be exited. Conversely, a dynamic variable is a memory location which is updated in a non-destructive way.

Following this intuition, we present in this paper a typed calculus capable of encoding both dynamic binding and global store at the same time. It is based on the modal λ-calculus for a variant of the intuitionistic S4 modal logic [23], which we extend with the concept of *names*. Names are labels that can be dynamically introduced into the computation and we will use them as a theoretic abstraction of memory locations. In this paper, names are explicitly second-class; they cannot be passed as function arguments, so all the memory operations must be over explicitly given locations.

As we have already observed in another paper [21], the operator $\Box$ of modal necessity, in combination with names, can be used to model effects that have local scope and can be handled. When this system is instantiated to treat names as memory locations, name dereference becomes an effect which is handled by corresponding name initializations. For example, similar to indexed monads in [31], we will have a type $\Box_C A$ which classifies *suspended* computations of type A which, in the course of eventual executions, may *read* from the memory locations whose names are listed in the set C. Assignment to memory locations is modeled by *explicit substitutions*, and a suspended computation reading from C

can be evaluated only in an environment in which all the names from C have been initialized by some explicit substitution. Because explicit substitutions have a delimited scope, they can only represent non-destructive update. Consequently, the obtained system models dynamic binding. This fragment is very similar to our meta-programming calculus from [20], with several important differences concerning names and explicit substitutions that make the distinction between meta-programming and dynamic binding; we comment on these in Sections 2 and 3.

As already remarked, global store can be obtained from dynamic binding if the assignments to dynamic variables are single-threaded, and their scope is extended to the end of the program. A natural way to achieve this in a modal calculus is to tie the explicit substitutions to the type of modal possibility (which is a monad). Thus, dually to $\Box_C A$, this will provide us with a type $\Diamond_C A$ classifying suspended computations that *write* into the memory locations represented by the names in C.

The obtained nominal modal calculus will therefore be capable of encoding the following important features related to state: (1) *dynamic allocation of uninitialized locations* which is modeled by the constructors for introduction of names; (2) *non-destructive update* which is modeled by the necessitation fragment of the calculus; (3) *destructive update* which is modeled by the possibility fragment of the calculus; (4) typing guarantees that non-initialized cells will not be dereferenced.

We believe that this makes our system an interesting contribution to the theory of monadic calculi for state, as to the best of our knowledge, most monadic calculi for state usually only model destructive update and allocation of initialized locations, and have not been used to represent non-destructive update or allocation of uninitialized memory [17, 18, 29, 30, 31, 11, 14].

2 Nominal necessity and dynamic binding

In this section, we briefly outline our calculus for dynamic binding. It is a specific instantiation of the more general effect calculus from [21].

The system is based on the proof-term calculus for the necessitation fragment of a variant of modal logic S4 [23]. The idea is to separate the notion of ordinary variables (which we also refer to as *expression* variables) which are statically bound by λ-abstraction, from the notion of dynamic variables whose values depend on the context in which they are used. As we outlined in the introduction, we represent dynamic variables via *names*. Names are labels which can be dynamically introduced into the computation via a separate binding mechanism. Upon the introduction, names are not initialized, and the type system will track the propagation of names in order to ensure that only initialized names are ever dereferenced.

For that purpose, the type system will feature a family $\Box_C$ of modal necessity types, index by a set of names C. A useful operational intuition about the terms of type $\Box_C A$, is to see them as *suspended* computations of type A which may dereference the dynamic variables listed in the set C. On the other hand, a non-modal type A is populated with computations which are *executable*. Values are assigned to dynamic variables by means of explicit substitutions. A suspended computation of type $\Box_C A$ can be forced and executed only in an environment in which all the dynamic variables in C are defined. It is exactly because the explicit substitutions have delimited scope, that our calculus in fact implements non-destructive update of memory locations, i.e., dynamic binding. In Section 4, we will extend the calculus with a mechanism to globalize the scope of explicit substitutions and obtain a calculus for state with destructive update.

The described indexing of the modal operator with names is similar to the one found in the monadic language from [31], where labels are used to identify the effects that may occur under a monad. In our setup, however, we will also allow dynamic introduction of fresh names into the computation (which corresponds to allocation of new dynamic variables), and establish a typing discipline for it.

We start by explaining the syntax and various syntactic conventions.

Names	X	$\in$	$\mathcal{N}$
Supports	C, D	$\in$	$\mathcal{P}_{fin}(\mathcal{N})$
Types	A	$::=$	$b \mid A_1 \to A_2 \mid A_1 \nrightarrow A_2 \mid \Box_C A$
Explicit substitutions	Θ	$::=$	$\cdot \mid X \to e, \Theta$
Terms	e	$::=$	$u \mid X \mid \langle\Theta\rangle e \mid \lambda x{:}A.\, e \mid e_1\ e_2 \mid$ **box** $e \mid$ **let box** $u = e_1$ **in** $e_2 \mid$ $\nu X{:}A.\, e \mid$ **choose** e
Variable contexts	Δ	$::=$	$\cdot \mid \Delta, u{:}A[C]$
Name contexts	Σ	$::=$	$\cdot \mid \Sigma, X{:}A$

The *finite set* C of names that a suspended term of type $\Box_C A$ may dereference, is referred to as *support* of such a term. All the names in a support are drawn from a countably infinite universe of names $\mathcal{N}$. In order to track the use of names, the typing assignments in the context Δ of expression variables must account not only for the typing, but also for the support of a variable. So, for example, the typing $u{:}A[C]$ declares a variable u which can be bound to an expression of type A and support C. When the support of a variable is empty, we will abbreviate $u{:}A[\,]$, simply as $u{:}A$. Similarly, we will simply omit the index support set in the type constructor $\Box_C$ when C is empty.

While the context Δ declares ordinary expression variables, which can be bound to expressions of arbitrary support, the context Σ declares the names (i.e., dynamic variables), and their types. For simplicity purposes, names in Σ have only types, but not supports, associated with them. Because the types of the calculus depend on names, we impose conditions on well-formedness of contexts: the context Σ is well-formed if every type in Σ uses only names declared to the left of it; the variable context Δ is well-formed with respect to Σ, if all the names that appear in the types of Δ are declared in Σ.

The term constructors **box** and **let box** are the introduction and elimination forms for the $\Box$ modality. Operationally, the term constructor **box** suspends the evaluation of its argument expression e, and wraps it into a thunk **box** e which can then be further manipulated by the rest of the program. The elimination form **let box** $u = e_1$ **in** e_2 takes the suspended expression boxed by e_1 and binds it to the expression variable u to be used in e_2.

Example 1 The function `sum` presented below takes an integer argument n and creates a suspension for computing the sum $1 + \cdots + n$.

```
fun sum (n : int) : □int =
    if n = 1 then box 1
    else
      let box u = sum (n - 1)
      in
          box (u + n)
      end

- s5 = sum 5;
val s5 = box (1 + 2 + 3 + 4 + 5) : □int
```

The expression variable u may be used in e_2 in both suspended positions (i.e., under a **box**), and in executable positions. For example, if we would like to force the evaluation of the suspended computation s5 from the above example, we can do it with in the following way.

```
let val u = s5 in u;
- val it = 15 : int
```

A dynamic variable X is dereferenced by simply using its name as part of some term. Assignment to a dynamic variable is done by an explicit substitution. We use the term constructor $\langle\Theta\rangle e$ to apply an explicit substitution Θ over an expression e. This is one of the important distinctions of the calculus for dynamic binding from our meta-programming calculus in [20]. For the purposes of meta-programming we syntactically tied explicit substitutions to expression *variables*, rather than to arbitrary expressions. This enabled us to substitute *open* terms for names within boxed expressions. Such a behavior is not required of a calculus for dynamic binding – here we substitute names at the time they are dereferenced (i.e., the surrounding term is unboxed), and we substitute them only with expressions which are *closed* at run-time.

Explicit substitutions are syntactically defined as lists of assignments of expressions to names. Furthermore, they are simultaneous; there is no ordering between the assignments. Also, no name is assigned twice in the same substitution. In other words, an explicit substitution is a function from the set of names to the set of terms

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

We treat the substitutions in our calculus as providing assignments for all the names; some names are assigned specific terms, and some names are simply unchanged by the substitution. For example, the empty substitution $\langle\,\rangle$ maps every name to itself. Given a substitution Θ, its domain $\mathbf{dom}(\Theta)$ is the set of names that the substitution does not fix. In other words

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

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

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

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

Names are introduced into the computation by means of constructors $\nu X{:}A.\ e$ and **choose** e, which are the introduction and elimination forms for the type constructor $A \twoheadrightarrow B$. As we already pointed out, names stand for dynamic variables, and dynamic variables can be viewed as memory locations. Thus, introduction of new names into the computation intuitively corresponds to *allocation* of new memory cells, where the exact size of the allocated segment would depend on the type of the name. With this in mind, we can describe the term constructor $\nu X{:}A.\ e$ of type $A \twoheadrightarrow B$ as *declaring* (but not allocating) a name $X{:}A$ that may be used in $e{:}B$, and prescribing a certain discipline in the use of X. The actual allocation is the duty of **choose**. So, for example, the redex **choose** $(\nu X{:}A.\ e)$ introduces a new, uninitialized dynamic variable $X{:}A$, and then proceeds to evaluate e.

The pattern of use of X in e is restricted in such a way that upon the evaluation, the value of e will not contain any significant references to X; all the appearances of X will either be substituted away, or otherwise appear in some dead-code part of e. This way, X is forced to be local; it is prevented from escaping the scope of its introducing ν, and making an observable effect.

The usual variable conventions on binding, α-renaming and capture-avoiding substitution, apply here for both ordinary and dynamic variables. The binding forms in the language are $\lambda x{:}A.\ e$, **let box** $u = e_1$ **in** e_2 and $\nu X{:}A.\ e$. Given a term e, we denote by $\mathbf{fv}(e)$ the set of free variables of e. The set of names appearing in the type A is denoted by $\mathbf{fn}(A)$.

Example 2 The following code segment illustrates the interaction between the several binding mechanisms of our language. First, we introduce new *dynamic variables* X and Y of integer type, by means of **choose** and ν. Then we build the polynomial X^2+Y^2 over these dynamic variables, and bind it, via **let box** to an *expression variable* u. Then we use u to create a function f which takes another *expression variable* z and returns a suspended code for computing X^2+Y^2+z. Notice that f is bound using a **let val** form, which has the usual operational behavior, and will be formally introduced shortly. The function f is an ordinary λ-abstraction, and we apply it to 1 to obtain the polynomial $v = X^2+Y^2+1$. Finally, the program instantiates X and Y to 1 and 2, respectively, before evaluating the polynomial $X^2+Y^2+1+2XY$ at the point $(X=1, Y=2)$.

```
- choose νX:int.
    choose νY:int.
      let box u = box (X^2+Y^2)
          val f = λz:int.  box (u + z)
          box v = f 1
      in
          <X->1, Y->2> (v + 2XY)
      end;

  val it = 10 :  int
```

The type system of the calculus for dynamic binding consists of two judgments: one for typing ordinary expressions, and another for typing explicit substitutions. The expression judgment has the form:

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

Given an expression e, the judgment checks whether e has type A, and whether the names that are dereferenced in unsuspended positions in e are accounted for in the support C. The judgment for explicit substitutions has the form:

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

The substitution Θ will be given a type $[C] \Rightarrow [D]$ if it provides definitions for names in C, and those definitions are themselves

Nominal modal λ-calculus

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

$$\frac{\Sigma;\Delta \vdash e_1 : A \to B[C] \quad \Sigma;\Delta \vdash e_2 : A[C]}{\Sigma;\Delta \vdash e_1\ e_2 : B[C]}$$

$$\frac{\Sigma;\Delta \vdash e : A[D]}{\Sigma;\Delta \vdash \mathbf{box}\ e : \Box_D A[C]}$$

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

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

Name dereference and explicit substitution

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

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

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

Figure 1. Typing rules for dynamic binding.

terms with support D. Therefore, one and the same substitution can be given many different types, depending on the supports C and D at which it is considered. For example, the substitution $\Theta = (X \to 1, Y \to 2)$ can be given (among others) the typings: $[\,] \Rightarrow [\,]$, $[X] \Rightarrow [\,]$, as well as $[X,Y,Z] \Rightarrow [Z]$. And indeed, when Θ acts on a term of support $[\,]$, another term with support $[\,]$ is produced; when Θ acts on a term of support $[X]$, it substitutes X away, and the obtained result is a term with empty support; when acting on a term with support $[X,Y,Z]$, X and Y are substituted by concrete terms, and only the dynamic variable Z remains in the support of the residual term. The rules of both judgments are presented in Figure 1, and we explain them next.

One of the important characteristic of the type system is the *support weakening principle*; that is

$$\text{if } \Sigma;\Delta \vdash e : A[C] \text{ and } C \subseteq D, \text{ then } \Sigma;\Delta \vdash e : A[D]$$

Support of the expression e determines which names should be instantiated by means of an explicit substitution before e can be executed. The support weakening principle simply states that instantiating more names than e actually requires, will not influence the typing (and therefore, the evaluation) of e; e could still be safely executed. We make the support weakening principle admissible by explicitly instrumenting the typing rule for hypothesis; a variable can be typed with a support which is larger from the one that the variable is declared with. Furthermore, we allow the values of the calculus, which are the λ- and ν-abstractions, and boxed expressions, to be typed with arbitrary supports. The evaluation of values is already finished, so it does not depend on any particular set of names being initialized; therefore, we can initialize any set we want.

λ-calculus fragment. The most important characteristic of the rule for λ-abstraction is that the body e is required to be pure; that is, e has to match the empty support. Thus, all the dynamic variables that e may dereference must be so dereferenced under a **box** (and correspondingly accounted for in the type of e). This is very similar with the monadic calculi, which require that their functions be pure, and the effects must appear guarded by a monad. This is also one of the points in which the calculus for dynamic binding differs from our meta-programming language from [20]. In [20] we do not insist that λ-abstractions be pure; impurity is handled by explicit substitutions, and explicit substitutions can descend under λ-binders. However, in such a setup, names would be instantiated before they are actually dereferenced – it is not how dynamic variables or state locations should behave.

A further observation about this rule is that λ-terms are values, so we can weaken their support arbitrarily, as explained before. Concerning the rule for function application, it simply checks both the function and the application argument against the same support.

Modal fragment. To type a suspended code **box** e, we must check if e is well-typed and matching the support that is supplied as an index to the $\Box$ constructor. Boxed expressions are values, so their support can be arbitrarily weakened to any well-formed support set C, just like in the case of λ-abstractions. The **let box** construct is supposed to first evaluate the branch e_1 to a boxed expression, and then bind the obtained expression to u before proceeding with e_2. Therefore, **let box** rule requires that both the branch e_1 and the body e_2 be ran in the environment defining the same set of names C.

Names fragment. The rule for $\nu X{:}A.\ e$ must check e for well-typedness in a context Σ extended with the new dynamic variable $X{:}A$. Similar to the λ rule, we require that e has empty support. The ν constructor, however, further requires that X does not appear in the type B. This ensures that X is used only *locally* in e; the process of evaluation can never make X escape the scope of its introducing ν in any observable way. Practically, this typing discipline translates into a requirement that all the uses of X in e are either substituted by an explicit substitution, or appear in some dead code part of e.

While ν-abstraction only declares a name X that can be used in e and enforces the described typing discipline, it does not actually allocate X. As we already pointed out, the allocation is the duty of the term constructor **choose**, which is the elimination form for $A \twoheadrightarrow B$. Operationally, **choose** allocates a fresh name, and substitutes it for the bound name in the ν-abstraction. Since this fresh name is irrelevant for the typing purposes, we don't place it into the context Σ in the conclusion or the typing rule for **choose**, but it will appear in the definition of the operational semantics.

Explicit substitutions The identity explicit substitution $\langle\,\rangle$ does not provide definitions for any names, or rather, as discussed before, each name is simply substituted by itself (in this sense, the substitutions $\langle\,\rangle$ and $\langle X \to X\rangle$ are really the same). Therefore, obviously, if an identity substitution is applied to an expression of support C

it will produce an expression of support C. To preserve the support weakening principle, we allow the target support of the identity substitution to be weakened to an arbitrary $D \supseteq C$. Composite substitutions are typechecked by recursively typechecking all of their assignments.

Last in this fragment are the rules for name dereferencing and substitution application. The rule for name dereferencing allows X to be used only if it is present in the support set C. Substitutions initialize the names in the expression over which they are applied, and so the rule for substitution application requires that the domain support C of the substitution Θ matches the support of the argument expression e.

Example 3 We can introduce **let val** $x = e_1$ **in** e_2 as a syntactic sugar for **let box** $u = (\lambda x.\ \mathbf{box}\ e_2)\ e_1$ **in** u. The boxing of e_2 is required in order to ensure that the purity of the λ-abstraction (as discussed before). The corresponding typing rule is

$$\frac{\Sigma;\Delta \vdash e_1 : A[C] \qquad \Sigma;(\Delta, x{:}A) \vdash e_2 : B[C]}{\Sigma;\Delta \vdash \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 : B[C]}$$

Similarly, we introduce the term constructor **let name** $X{:}A$ **in** e as an abbreviation for **let box** $u =$ **choose** $(\nu X{:}A.\ \mathbf{box}\ e)$ **in** u, with the derived typing rule

$$\frac{(\Sigma, X{:}A);\Delta \vdash e : B[C] \qquad X \notin \mathbf{fn}(B,C)}{\Sigma;\Delta \vdash \mathbf{let\ name}\ X{:}A\ \mathbf{in}\ e : B[C]}$$

Example 4 Assume that C_1, C_2 and D are arbitrary support sets. Then the following terms are well-typed of *empty support*.

$f_1 : \Box_{C_1}A \to \Box_{C_2}A =$ (where $C_1 \subseteq C_2$)
$\lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ u$

$f_2 : \Box A \to A =$
$\lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u$

$f_3 : A \to \Box_D A =$
$\lambda x.\ \mathbf{box}\ x$

$f_4 : \Box_{C_1}A \to \Box_D\Box_{C_2}A =$ (where $C_1 \subseteq C_2$)
$\lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ (\mathbf{box}\ u)$

$f_5 : \Box_{C_1}(A \to B) \to \Box_{C_2}A \to \Box_D B =$ (where $C_1, C_2 \subseteq D$)
$\lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ let\ box}\ v = y\ \mathbf{in\ box}\ (u\ v)$

The function f_1 simply eta-expands its argument. It shows that support weakening in boxed types is derivable. The function f_2 illustrates that we can "unbox" and evaluate suspended expressions which *do not read from dynamic variables*; notice that the argument type of f_2 has empty index support. The function f_3 shows that it is possible to coerce values into suspended computation, by simply boxing them. Because values are name-free, their support can be weakened to an arbitrary support set D. The other two functions generalize the characteristic axioms of S4 modal necessity without names [7, 23].

3 Operational semantics for dynamic binding

The operational semantics of the calculus for dynamic binding is defined through the judgment

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

In this judgment, Σ and Σ' are run-time contexts of currently allocated, but not necessarily initialized names. The judgment relates the expression e with its one-step reduct e', and is defined on expressions which do not contain free expression variables. Expression e can contain names (i.e., dynamic variables), but it must have *empty support*. In other words, we only consider for evaluation those expressions which do not dereference uninitialized names; their names are either initialized by some explicit substitution, or otherwise appear in suspended or dead-code subterms. The reduction can allocate new names, and that is why we must keep track in the judgment of the run-time contexts of allocated names.

The judgment implements call-by-value operational semantics for the calculus in the style of Wright and Felleisen [34]. The idea is to decompose each expression e uniquely as $e = E[r]$ where E is an evaluation context and r is a redex. To define a small-step operational semantics, it is enough to define primitive reduction relation for redexes (which we denote by $\longrightarrow$), and let the evaluation of expressions (which we denote by $\longmapsto$) always first reduce the redex identified by the unique decomposition.

The definition of the judgment relies on the notion of values, redexes and evaluation contexts below.

Values	v	::=	$x \mid \lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e$
Value substitutions	σ	::=	$\cdot \mid X \to v, \sigma$
Redexes	r	::=	$v_1\ v_2 \mid \mathbf{let\ box}\ u = v\ \mathbf{in}\ e \mid$ $\mathbf{choose}\ v \mid \langle\sigma\rangle e$
Evaluation contexts	E	::=	$[\,] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid$ $\mathbf{choose}\ E \mid \langle\sigma, X \to E, \Theta\rangle e$

Notice that the definition of evaluation contexts proscribes a left-to-right evaluation strategy for function applications and for the assignment clauses in explicit substitutions. The primitive reduction and the evaluation relation for the call-by-value version of our calculus are defined below.

$$\begin{aligned}
\Sigma, (\lambda x.\ e)\ v &\longrightarrow \Sigma, [v/x]e \\
\Sigma, \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 &\longrightarrow \Sigma, [e_1/u]e_2 \\
\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) &\longrightarrow (\Sigma, Y{:}A), [Y/X]e \\
&\qquad \text{where } Y \notin \mathbf{dom}(\Sigma) \\
\Sigma, \langle\sigma\rangle e &\longrightarrow \Sigma, \{\sigma\}e
\end{aligned}$$

$$\frac{\Sigma, r \longrightarrow \Sigma', e'}{\Sigma, E[r] \longmapsto \Sigma', E[e']}$$

The operational semantics of the fragment corresponding to the modal λ-calculus is fairly standard: function application and **let box** reduce by performing a capture-avoiding substitution. The more interesting are the reductions concerning name introduction and explicit substitutions.

For example, the operational semantics for **choose** $(\nu X{:}A.\ e)$ prescribes that the run-time name context Σ be extended with a fresh name before proceeding with the evaluation of e. As we pointed out before, this provides our calculus with a formal way to model memory allocation.

Further observe that the operational semantics does not allow evaluations under an explicit substitution, and thus uninitialized names will never be encountered during the evaluation. Rather, the substitution application $\langle\sigma\rangle e$ is reduced by employing the meta-operation $\{\sigma\}e$ to carry out the substitution σ over e, before the evaluation of e can proceed. We next define the meta-operation $\{\sigma\}e$.

DEFINITION 1 (SUBSTITUTION APPLICATION). *Given a substitution Θ 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.*

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

The most important aspect of the above definition is that substitution application does not descend under **box**. Names appearing in a suspended code need not be initialized because suspended code is not evaluated, and hence its names are not dereferenced. However, when a suspension is actually unboxed and executed, this has to be done in a scope of a substitution that initializes the relevant names, as illustrated in Example 2. On the other hand, the other value forms are not of particular interest in this definition, because values are name-free, so substitution application does not descend into λ- and ν-abstractions. As we already commented before, this is in sharp distinction from the meta-programming calculus in [20], but it is the characteristics of dynamic binding, and it will allow us to extend the calculus to model state in Section 4. Further, notice that substitution application over a variable u is explicitly remembered, resulting in a term $\langle\Theta\rangle u$. When the variable u is substituted by a certain expression, the names appearing in this expression will be initialized by Θ.

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

DEFINITION 2 (COMPOSITION OF SUBSTITUTIONS). *Given two substitutions Θ_1 and Θ_2 with finite domains, their composition $\Theta_1\circ\Theta_2$ is the substitution defined as*

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

This operation is obviously well-founded as both the substitutions have finite domains, and substitution application of Θ_1 to the terms from the range of Θ_2 proceeds recursively on terms of decreasing size.

Example 5 Consider the ML-like program below.

```
let val x = ref 0
    fun f (y) = !x + y
    val z = f 1
in
   ((x:=1; f 1), z)
end
```

A similar program can be written in our calculus of dynamic binding as follows.

```
- choose (νX:int.
  <X -> 0>
  let fun f(y: int) : □X int = box (X + y)
      box u = f 1
      val z = u
  in
      (<X -> 1>u, z)
  end);

val it = (2, 1) : int * int
```

Note that in this code segment it is not necessary that the substitution `<X -> 0>` immediately follows the introduction of the name X. Indeed, we could have moved this substitution further down. It is only important that some substitution is active when the variable u is used in unsuspended positions. The variable u is bound to the suspended expression `(X + 1)`, so X must be initialized before u is used. In this particular example, the first unsuspended reference to u (and therefore to X as well) is in the scope of a substitution `<X -> 0>` and the second one is in the scope of `<X -> 1>`.

4 Nominal possibility and state

In the previous sections, we have demonstrated how the modal operator of necessity can be used to obtain a calculus for dynamic binding. We represented dynamic variables by names, and considered a reference to a name to be an effect, while substituting a name is the handler. This way, the calculus keeps track of names which are used, and prevents references to uninitialized names. In this section we build on this calculus to obtain a modal calculus for state. We would like to treat names as locations; dereferencing a name would correspond to a *read*, and substituting a name would correspond to an *update*. But, as the following program formulated in the type system from Section 2 illustrates, explicit substitutions cannot perform the update *destructively*.

```
choose (νX:int.
<X -> 0>
let fun f(y: int) : □X int = box (X + y)
    box u = f 1
in
    (<X -> 1>u, u + 1)
end)
```

Indeed, the subterm `<X -> 1>u` cannot possibly destructively update X to 1 before evaluating u, simply because the old value of X (in this case 0), has to be preserved for the evaluation of the second element of the pair, `u + 1`. Explicit substitutions alone are too weak.

A solution is to single-thread the explicit substitutions, so that once a substitution is attempted, its scope extends to the rest of the program; it is never required to revert back to some previous substitution. Thus, there would always be exactly one substitution "active" at every single moment, and it would play the role of *global store*.

Single-threading and scope extension are exactly the duties of monads, and in modal logic we have the monad $\Diamond$ of modal possibility. Thus, if we want to use explicit substitutions to model destructive state update, we need to tie explicit substitutions to $\Diamond$. Intuitively then, we should obtain a whole family $\Diamond_C$ of possibility operators indexed by support sets, where the type $\Diamond_C A$ classifies an explicit substitution for C *paired up* with a computation of type A – in other words, a closure. More concretely, $\Diamond_C$ types programs of type A

which first *write* into locations C and then compute a value of type A in the changed context. This would pleasantly contrast the type $\Box_C A$ that we already used in Section 2 to type programs which *read* from locations C before computing a value of type A.

We introduce the nominal possibility into the language by defining the following syntactic categories on top of the syntax of the calculus of dynamic binding from Section 2.

$$\begin{array}{llll} \textit{Types} & A & ::= & \ldots \mid \Diamond_C A \\ \textit{Closures} & f & ::= & [\Theta, e] \mid \textbf{let dia}\ x = e\ \textbf{in}\ f \mid \textbf{let box}\ u = e\ \textbf{in}\ f \\ \textit{Terms} & e & ::= & \ldots \mid \textbf{dia}\ f \end{array}$$

As expected, the grammar of types is extended with the family $\Diamond_C A$, whose term constructor is **dia** f, encapsulating a *closure* f. Closures are a new syntactic category intended to describe computations which change the global store. The basic closure constructor is the form $[\Theta, e]$ which ties a substitution Θ and a term e together; this is a computation which first writes into the locations determined by Θ before evaluating e in the new store. When Θ is the identity substitution, we will only write $[e]$ instead of $[\langle\,\rangle, e]$. Closures are deconstructed by the form **let dia** $x = e$ **in** f. It takes a term e which encapsulates a closure, thus carrying a substitution and a term. Intuitively, the term is then bound to x and the substitution carried out, before the evaluation of f is undertaken. The closure form **let box** $u = e$ **in** f takes a suspended expression encapsulated in the term e and binds it to u to be used in the closure f.

Example 6 Assuming that X and Y are integer names and that no other names are defined in the global store, the expression

```
let dia z = dia [<X->1, Y->2>, 2XY]
in
    [<X->X, Y->Y>, X^2+Y^2+z]
end
```

writes 1 and 2 into the locations X and Y respectively, then binds 4 to the local variable `z`, before evaluating to the closure `[<X->1, Y->2>, X^2+Y^2+4]`. Observe that the substitution in `[<X->X, Y->Y>, X^2+Y^2+z]` is the identity; we could have reduced the verbosity by writing the closure simply as `[X^2+Y^2+z]`.

The type system of the calculus for nominal possibility consists of two mutually recursive judgments: one for typing expressions, and another one for typing closures. The expression judgment extends the system from Section 2, and has the form

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

establishing that e may possibly read from locations listed in the support set C. The closure judgment has the form

$$\Sigma; \Delta \vdash f \div_D A[C]$$

This judgment establishes that the closure f consists of a substitution of type $[D] \Rightarrow [C]$, and a term of type A. The term may dereference the names from the support D, because they are initialized by the substitution. We present the type system in Figure 2, and comment on the rules below.

The closure introduction rule simply makes explicit the intuition about closures: the names used in the closure body are initialized by the closure substitution; the substitution closes the body up. Note that this rule is almost identical to the rule for substitution application from Section 2. This is only to be expected since, after all, we introduced closures with the intention to single-thread substitution applications. The two constructs, however, will have different

$$\frac{\Sigma; \Delta \vdash e : A[D] \quad \Sigma; \Delta \vdash \langle\Theta\rangle : [D] \Rightarrow [C]}{\Sigma; \Delta \vdash [\Theta, e] \div_D A[C]}$$

$$\frac{\Sigma; \Delta \vdash e : \Diamond_{D_1} A[C] \quad \Sigma; (\Delta, x{:}A) \vdash f \div_{D_2} B[D_1]}{\Sigma; \Delta \vdash \textbf{let dia}\ x = e\ \textbf{in}\ f \div_{D_2} B[C]}$$

$$\frac{\Sigma; \Delta \vdash e : \Box_{D_1} A[C] \quad \Sigma; (\Delta, u{:}A[D_1]) \vdash f \div_{D_2} B[C]}{\Sigma; \Delta \vdash \textbf{let box}\ u = e\ \textbf{in}\ f \div_{D_2} B[C]}$$

$$\frac{\Sigma; \Delta \vdash f \div_D A[C]}{\Sigma; \Delta \vdash \textbf{dia}\ f : \Diamond_D A[C]}$$

Figure 2. Typing rules for nominal possibility.

operational meaning. The explicit substitution $\langle\Theta\rangle e$ carries out the substitution Θ over the expression e, while the closure $[\Theta, e]$ suspends the substitution, until it is explicitly forced by the **let dia** rule, as would be formalized in the operational semantics of the calculus. The first provides non-destructive location update, while the second should be used when destructive update is desired. What is interesting is that both capabilities harmoniously coexist within the system.

The typing rule for **dia** is a judgmental coercion from closures to terms. When coerced into the category of terms, a closure is given the type of modal possibility $\Diamond_C$. The index support C of this type records the names that the substitution in the closure defines.

The construct **let dia** is an elimination form for closures because **let dia** $e = x$ **in** f is intended to destruct the closure computed by e, and use its parts in to compute f. To give a more specific description of the typing rule for **let dia** $e = x$ **in** f, we start with the observation that the term e is required to be of type $\Diamond_D A$. It is supposed to encode a closure consisting of a substitution Θ of type $[D] \Rightarrow [C]$ and a term $e' : A[D]$. The role of **let dia** is to institute the substitution Θ as a new global store providing definitions for names in the support D, then evaluate e' to a value, bind it to x and proceed with the evaluation of f. Following this semantics, we can allow f to be supported by D, because the new global store in which f is evaluated defines the names from D. We are also free to declare x as being of empty support in the typing of f, because x will always be bound to a value.

Example 7 We will use some further syntactic abbreviations as well: **let val** $x = e$ **in** f is short for

$$\textbf{let dia}\ y = (\textbf{let val}\ x = e\ \textbf{in dia}\ f)\ \textbf{in}\ [y]$$

and **let name** $X{:}A$ **in** f is short for

$$\textbf{let dia}\ y = (\textbf{let name}\ X{:}A\ \textbf{in dia}\ f)\ \textbf{in}\ [y]$$

Operationally, the **let box** rule could also be seen as a similar abbreviation, but it is usually consider primitive because of proof-theoretic reasons: it is required during proof search in order to preserve the subformula property of the calculus. In contrast, **let val**

does not have any significance for proof search, and the same holds for **let name**. The typing rules for the two are easily derived as

$$\frac{\Sigma;\Delta \vdash e : A[C] \qquad \Sigma;(\Delta,x{:}A) \vdash f \div_D B[C]}{\Sigma;\Delta \vdash \textbf{let val } x = e \textbf{ in } f \div_D B[C]}$$

$$\frac{(\Sigma,X{:}A);\Delta \vdash f \div_D B[C] \qquad X \notin \textbf{fn}(B,C,D)}{\Sigma;\Delta \vdash \textbf{let name } X{:}A \textbf{ in } f \div_D B[C]}$$

Example 8 Assume that C and D are support sets. Then the following terms are well-typed of empty support.

$$\begin{array}{ll} f_1 : \Diamond_D A \to \Diamond_C A = & \text{(where } C \subseteq D\text{)} \\ \quad \lambda x.\ \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in}\ [y]) & \\ f_2 : A \to \Diamond A = & \\ \quad \lambda x.\ \textbf{dia}\ [x] & \\ f_3 : \Diamond_C \Diamond_D A \to \Diamond_D A = & \\ \quad \lambda x.\ \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in let dia}\ z = y\ \textbf{in}\ [z]) & \\ f_4 : \Box_C (A \to B) \to \Diamond_D A \to \Diamond_D B = & \text{(where } C \subseteq D\text{)} \\ \quad \lambda e_1.\ \lambda e_2.\ \textbf{dia}\ (\textbf{let box}\ u = e_1\ \textbf{in let dia}\ x = e_2\ \textbf{in}\ [u\,x]) & \end{array}$$

The function f_1 simply eta-expands its argument x. It illustrates that strengthening at the index supports of $\Diamond$ types is derivable. This is not surprising, as it only involves forgetting some entries from the substitution part of the closure x. The rest of the terms generalize the characteristic axioms of a variant of S4 modal possibility introduced in [23], which can be recovered if the involved supports are set to be empty. For example, the function f_2 is a coercion from terms into closures with empty substitution; notice that the range type is $\Diamond A$ with empty index support. Coercions from A to $\Diamond_C A$ with non-empty C are not generally available as they require providing definitions for each name in C. The function f_3 illustrates that it is only the last layer of $\Diamond$'s that matter; all the additional ones can be ignored. The function f_4 takes a function e_1 which needs names C in order to be generated, and a computation e_2 providing a term x and definitions for these names (and possibly some more, since its index support is $D \supseteq C$). Then the definitions provided by e_2 are plugged into e_1 to obtain the function u which is then applied to x to obtain the final result.

It is important to emphasize that the particular typing assigned to an explicit substitution in a closure will have a significant impact on its operational behavior. As explained in Section 2, explicit substitutions have polymorphic typing. For example, the identity substitution <> can have (among others) all of the following types $[\,] \Rightarrow [\,]$, or $[X] \Rightarrow [X]$, or $[X] \Rightarrow [X,Y]$. But, when a substitution is viewed as global store, the specific type assigned to it determines which locations will be used in the rest of the program, and which locations are irrelevant and can therefore be deallocated and garbage-collected. In that sense, the identity substitution with type $[\,] \Rightarrow [\,]$ may be seen an empty store, while the identity substitution with the type $[X] \Rightarrow [X]$ describes a store with a live location X which is filled according to the value of X from the previously active store. For simplicity, we do not consider here a type system and an operational semantics that would make use of this distinction, although it is an interesting future work. Rather, we consider that all the operations on substitutions never "forget" any assignments that the substitution may implicitly contain, depending on the support at which it is considered. When substitutions are viewed as global store, this means that our operational semantics does not prescribe deallocation and garbage-collection of store locations. In a realistic implementation, just as customary, these processes would be performed by a separately specified run-time system.

Example 9 We can use the new type and term constructors for nominal possibility to single-thread and sequence the example given at the beginning of the section.

```
let name X:int
    dia dummy = dia [<X->0>, ()]
    fun f(y : int) : □X int = box (X + y)
    box u = f 1
    val z = u + 1
    let dia w = dia [<X->1>, u]
    in
        [(w, z)]
    end
end
```

The resulting program is typed in the judgment for closures, and evaluates to `[(2, 2)]`. The substitution associated with the result is suppressed as it is equal to the identity. But, even more is true; this substitution can actually be typed as $[\,] \Rightarrow [\,]$. Indeed, the body `(w, z)` of the innermost **let dia** is name-free because it only depends on variables w and z which themselves bind (name free) values. The outcome of the program will, hence, be closed; we can type it in empty global store. In particular, the newly introduced location X could have been deallocated.

5 Operational semantics for state

In this section we develop a call-by-value operational semantics for the modal calculus with possibility. We ignore the closure constructors **let box**, **let val** and **let name**; for operational purposes, they can be viewed as syntactic abbreviations and will not influence the properties we explore here.

The first step is to extend the meta operation of substitution application to account for the new constructs.

$$\begin{array}{lll} \{\Theta\}\ \ \textbf{dia}\ f & = & \textbf{dia}\ \{\Theta\} f \\ \{\Theta\}\ \ [\Theta', e] & = & [\Theta \circ \Theta', e] \\ \{\Theta\}\ \ \textbf{let dia}\ x = e\ \textbf{in}\ f & = & \textbf{let dia}\ x = \{\Theta\} e\ \textbf{in}\ f \end{array}$$

Note that the substitution application is carried out only over the branch e, but not over the body f of a **let dia** construct. This is justified because f is evaluated under a substitution determined by e; any influence that Θ might have over f has to be via e.

The operational semantics is defined by means of two evaluation judgments: one for expressions and one for closures. We adopt a particular formulation of these judgments which emphasizes the relationship between the nominal possibility and global state. The expression evaluation judgment has the form

$$\Sigma, e \overset{\sigma}{\longmapsto} \Sigma', e'$$

and reads: in a context of declared locations Σ and a store σ assigning values to these locations (and some locations may remain uninitialized), the term e steps into e' and possibly introduces new locations Σ'. The evaluation steps cannot change the store σ, as expressions can only read from the store but not write into it. This judgment is a straightforward extension of the evaluation judgment from Section 2.

The judgment for evaluating closures is the default judgment of the operational semantics, because it prescribes evaluation of stateful constructs. It has the form

$$(\Sigma,\sigma), f \longmapsto (\Sigma',\sigma'), f'$$

where f steps into f', changing in the process the set of allocated locations from Σ into Σ' and the global store from σ into σ'. The evaluation strategy that we consider will evaluate under the constructor **dia** only if it is found in a let-branch of a **let dia**. This way, the changes to the global store prescribed under **dia** will take place only when they are single-threaded by a **let dia**. Note that this is not the only possible evaluation strategy, but it is the one that relates nominal possibility to global state and destructive update. Following this idea, we extend the categories of values, evaluation contexts and redexes from Section 2 as summarized below.

Values	v	::=	$\ldots \mid \mathbf{dia}\ f$
Redexes	r	::=	$\ldots \mid X$
Closure contexts	F	::=	$[\,] \mid \mathbf{let\ dia}\ x = \mathbf{dia}\ F\ \mathbf{in}\ f \mid$ $\mathbf{let\ dia}\ x = E\ \mathbf{in}\ f \mid \mathbf{let\ dia}\ x = \mathbf{dia}\ [E]\ \mathbf{in}\ f \mid$ $\mathbf{let\ dia}\ x = \mathbf{dia}\ [\langle \sigma, X \to E, \Theta\rangle, e]\ \mathbf{in}\ f$
Closure redexes	c	::=	$\mathbf{let\ dia}\ x = [\sigma, e]\ \mathbf{in}\ f \mid \mathbf{let\ dia}\ x = [v]\ \mathbf{in}\ f$

The two evaluation judgments require two primitive reduction relations. Primitive reductions for expressions are duplicates of the reductions from Section 3, except that now the reductions are considered in the context of a distinguished substitution σ serving as a global store. We also add a new rule for reducing names.

$$\begin{array}{rcl}
\Sigma, (\lambda x.\ e)\ v & \xrightarrow{\sigma} & \Sigma, [v/x]e \\
\Sigma, \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 & \xrightarrow{\sigma} & \Sigma, [e_1/u]e_2 \\
\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) & \xrightarrow{\sigma} & (\Sigma, X{:}A), e \\
\Sigma, \langle\sigma'\rangle e & \xrightarrow{\sigma} & \Sigma, \{\sigma'\}e \\
\Sigma, X & \xrightarrow{\sigma} & \Sigma, \sigma(X)
\end{array}$$

The primitive reductions for closures are defined as follows.

$$\begin{array}{rcl}
(\Sigma,\sigma), \mathbf{let\ dia}\ x = \mathbf{dia}\ [v]\ \mathbf{in}\ f & \longrightarrow & (\Sigma,\sigma), [v/x]f \\
(\Sigma,\sigma), \mathbf{let\ dia}\ x = \mathbf{dia}\ [\sigma', e]\ \mathbf{in}\ f & \longrightarrow & \\
\multicolumn{3}{r}{(\Sigma, \sigma\circ\sigma'), \mathbf{let\ dia}\ x = \mathbf{dia}\ [e]\ \mathbf{in}\ f} \\
\multicolumn{3}{r}{\text{if } \sigma' \neq (\cdot)}
\end{array}$$

The first rule simply binds v to x if the substitution in the closure of v is the identity, and therefore does not prescribe any changes to the global store. The second rule is more complicated. Its meaning is to change the global store according to the closure substitution and continue evaluating in the new store. Thus, the substitution σ' is moved out of the closure and composed with σ which is the current global store. As discussed before, the composition is performed so that no assignments in the result are omitted. The composition with σ' will change some assignments in the global store, but no assignments will be lost.

The evaluation of expressions and closures are now defined as follows.

$$\frac{\Sigma, r \xrightarrow{\sigma} \Sigma', e'}{\Sigma, E[r] \xmapsto{\sigma} \Sigma', E[e']}$$

$$\frac{\Sigma, r \xrightarrow{\sigma} \Sigma', e'}{(\Sigma,\sigma), F[r] \longmapsto (\Sigma',\sigma), F[e']} \qquad \frac{(\Sigma,\sigma), c \longrightarrow (\Sigma',\sigma'), f'}{(\Sigma,\sigma), F[c] \longmapsto (\Sigma',\sigma'), F[f']}$$

6 Structural properties and type soundness

In this section we establish the basic properties of our type system. We start with the admissibility of support weakening, as discussed in Section 2.

LEMMA 3 (SUPPORT WEAKENING). *1. if $\Sigma;\Delta \vdash e : A[C]$ and $C \subseteq D$, then $\Sigma;\Delta \vdash e : A[D]$*
2. if $\Sigma;\Delta \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C]$ and $C \subseteq D$, then $\Sigma;\Delta \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$
3. if $\Sigma;\Delta \vdash f \div_{C_1} A[C]$ and $C \subseteq D$, then $\Sigma;\Delta \vdash f \div_{C_1} A[D]$

PROOF. By simultaneous induction on the three derivations. □

LEMMA 4 (EXPRESSION SUBSTITUTION PRINCIPLE). *Let $\Sigma;\Delta \vdash e_1 : A[C]$. Then the following holds:*
1. if $\Sigma;(\Delta, u{:}A[C]) \vdash e_2 : B[D]$, then $\Sigma;\Delta \vdash [e_1/u]e_2 : B[D]$
2. if $\Sigma;(\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : [D'] \Rightarrow [D]$, then $\Sigma;\Delta \vdash \langle[e_1/u]\Theta\rangle : [D'] \Rightarrow [D]$
3. if $\Sigma;(\Delta, u{:}A[C]) \vdash f \div_{C_1} B[D]$, then $\Sigma;\Delta \vdash [e_1/u]f \div_{C_1} B[D]$

PROOF. By simultaneous induction on the three derivations. □

LEMMA 5 (EXPLICIT SUBSTITUTION PRINCIPLE). *Let $\Sigma;\Delta \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Then the following holds:*
1. if $\Sigma;\Delta \vdash e : A[C]$ then $\Sigma;\Delta \vdash \{\Theta\}e : A[D]$
2. if $\Sigma;\Delta \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$, then $\Sigma;\Delta \vdash \langle\Theta\circ\Theta'\rangle : [C_1] \Rightarrow [D]$
3. if $\Sigma;\Delta \vdash f \div_{C_1} A[C]$, then $\Sigma;\Delta \vdash \{\Theta\}f \div_{C_1} A[D]$

PROOF. By simultaneous induction on the structure of the derivations. The proof of the second statement is the most interesting, and we present it here. Given the substitutions Θ and Θ', we can always split the representation of $\Theta\circ\Theta'$ into two disjoint sets:

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

Regarding these sets, we could obtain the final result, if we show that

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

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

Indeed, it is easy to establish by induction that the union of these sets, when viewed as a substitution, has the required typing. To establish (a), observe that from the typing of Θ it is clear that $\Theta'_1 : [C \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$. Then, by definition of $\mathbf{dom}(\Theta')$, if $X \in C_1 \setminus \mathbf{dom}(\Theta')$, then X is fixed by Θ'. Thus, either X does not appear in the syntactic representation of Θ', or the syntactic representation of Θ' contains a sequence of mappings $X \to X_1, X_1 \to X_2, \ldots, X_n \to X$. In the second case, X is the substituting term for X_n, and thus $X \in C$. In the first case, $X \in C$ by inductively appealing to the typing rules for substitutions until the empty substitution is reached. Either way, $C_1 \setminus \mathbf{dom}(\Theta') \subseteq C$, and furthermore $C_1 \setminus \mathbf{dom}(\Theta') \subseteq C \setminus \mathbf{dom}(\Theta')$. Now, appealing inductively to the typing rules for substitutions, we easily obtain that $\Sigma;\Delta \vdash \langle\Theta'_1\rangle : [C_1 \setminus \mathbf{dom}(\Theta')] \Rightarrow [D]$.

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

LEMMA 6 (CANONICAL FORMS). *Let v be a closed value such that $\Sigma;\cdot;\vdash v : A[C]$. Then the following holds:*

1. *if* $A = A_1 \to A_2$, *then* $v = \lambda x{:}A_1.\, e$ *and* $\Sigma; x{:}A_1 \vdash e : A_1\,[\,]$
2. *if* $A = \Box_D B$, *then* $v = \mathbf{box}\ e$ *and* $\Sigma; \cdot \vdash e : B\,[D]$
3. *if* $A = A_1 \mathrel{-\!\!\!\rightarrow} A_2$, *then* $v = \nu X{:}A_1.\, e$ *and* $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$
4. *if* $A = \Diamond_D B$, *then* $v = \mathbf{dia}\ f$ *and* $\Sigma; \cdot \vdash f \div_D B\,[C]$

As a consequence, the support of v can be arbitrarily weakened, i.e. $\Sigma; \cdot \vdash v : A\,[D]$, *for any support D.*

PROOF. By case analysis on the structure of closed values. □

LEMMA 7 (REPLACEMENT). *1. If* $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exists a type B such that*
   (a) $\Sigma; \Delta \vdash e : B\,[C]$, *and*
   (b) *if* Σ', Δ' *extend* Σ, Δ, *and* $\Sigma'; \Delta' \vdash e' : B\,[C]$, *then* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

2. *If* $\Sigma; \Delta \vdash F[e] \div_C A\,[D]$, *then there exists a type B such that*
   (a) $\Sigma; \Delta \vdash e : B\,[D]$, *and*
   (b) *if* Σ', Δ' *extend* Σ, Δ *and* $\Sigma'; \Delta' \vdash e' : B\,[D]$, *then* $\Sigma'; \Delta' \vdash F[e'] \div_C A\,[D]$

3. *If* $\Sigma; \Delta \vdash F[f] \div_C A\,[D]$, *then there exists a type B and support* C_1 *such that*
   (a) $\Sigma; \Delta \vdash f \div_{C_1} B\,[D]$, *and*
   (b) *if* Σ', Δ' *extend* Σ, Δ *and* D_1 *is a support set such that* $\Sigma'; \Delta' \vdash f' \div_{C_1} B\,[D_1]$, *then* $\Sigma'; \Delta' \vdash F[f'] \div_C A\,[D_1]$

PROOF. By simultaneous induction on the three derivations, using support weakening. □

LEMMA 8 (SUBJECT REDUCTION). *Let* $\Sigma; \cdot \vdash \langle\sigma\rangle : [C] \Rightarrow [\,]$. *Then the following holds:*

1. *if* $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \xrightarrow{\sigma} \Sigma', e'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : A\,[C]$
2. *if* $\Sigma; \cdot \vdash f \div_D A\,[C]$ *and* $(\Sigma, \sigma), f \longrightarrow (\Sigma', \sigma'), f'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash \langle\sigma'\rangle : [C'] \Rightarrow [\,]$ *and* $\Sigma'; \cdot \vdash f' \div_D A\,[C']$ *for some support set* $C' \subseteq \mathbf{dom}(\Sigma')$

PROOF. By case analysis using canonical forms lemma, support weakening and the substitution principles. □

The following theorem establishes that the evaluation relation is sound with respect to typing.

THEOREM 9 (PRESERVATION). *Let* $\Sigma; \cdot; \vdash \langle\sigma\rangle : [C] \Rightarrow [\,]$. *Then the following holds:*

1. *if* $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \xrightarrow{\sigma} \Sigma', e'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : A\,[C]$
2. *if* $\Sigma; \cdot \vdash f \div_D A\,[C]$ *and* $(\Sigma, \sigma), f \longrightarrow (\Sigma', \sigma'), f'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash \langle\sigma'\rangle : [C'] \Rightarrow [\,]$ *and* $\Sigma'; \cdot \vdash f' \div_D A\,[C']$ *for some support set* $C' \subseteq \mathbf{dom}(\Sigma')$

PROOF. By evaluation rules, the term is decomposed into a context and a redex. Then use the replacement and subject reduction lemmas. □

LEMMA 10 (PROGRESS FOR $\longrightarrow$). *Let* σ *be an arbitrary value substitution. Then the following holds:*

1. *if* $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \xrightarrow{\sigma} \Sigma', e'$.
2. *if* $\Sigma; \cdot \vdash c \div_D A\,[C]$, *then there exist a closure* f', *a value substitution* σ' *and a context* Σ', *such that* $(\Sigma, \sigma), c \longrightarrow (\Sigma', \sigma'), f'$.

PROOF. By case analysis on the structure of redexes, using canonical forms lemma. □

LEMMA 11 (UNIQUE DECOMPOSITION). *1. For every expression e, either:*
   (a) *e is a value, or*
   (b) $e = E[r]$ *for a unique evaluation context E and a redex r.*

2. *For every closure f, either:*
   (a) $f = [\Theta, e]$ *for some substitution* Θ *and expression e, or*
   (b) $f = F[r]$ *for a unique closure context F and term redex r, or*
   (c) $f = F[c]$ *for a unique closure context F and closure redex c.*

PROOF. By simultaneous induction on the structure of *e* and *f*. □

The next theorem proves that a well-typed term can always be reduced at least once. In combination with the preservation theorem, it establishes that well-typed terms do not get stuck.

THEOREM 12 (PROGRESS). *Let* $\Sigma; \cdot \vdash \langle\sigma\rangle : [C] \Rightarrow [\,]$. *Then the following holds:*

1. *if* $\Sigma; \cdot \vdash e : A\,[C]$, *then either*
   (a) *e is a value, or*
   (b) *there exists a term* e' *and a context* Σ', *such that* $\Sigma, e \overset{\sigma}{\longmapsto} \Sigma', e'$.

2. *if* $\Sigma; \cdot \vdash f \div_D A\,[C]$, *then either*
   (a) $f = [\Theta, e]$ *for some substitution* Θ *and an expression e, or*
   (b) *there exists a closure* f', *a context* Σ', *and a value substitution* σ', *such that* $(\Sigma, \sigma), f \longmapsto (\Sigma', \sigma'), f'$

PROOF. By unique decomposition lemma, the term can be decomposed into an evaluation context and a redex. Then appeal to the replacement lemma and progress for primitive reduction. □

The reduct e' and the context Σ' are not necessarily unique for each given *e* and Σ. Because fresh names may be introduced during the course of the computation, two different evaluations of one and the same term may choose the fresh names differently, resulting in different e' and Σ'. The determinacy lemma below shows that the differences between two reducts of one and the same term arise only from this arbitrary choice of fresh names. As customary, we denote by $\longmapsto^n$ the *n*-step reduction relation.

LEMMA 13 (DETERMINACY). *1. If e,* e_1, e_2 *are terms such that* $\Sigma, e \xrightarrow{\sigma}{}^{n} \Sigma_1, e_1$ *and* $\Sigma, e \xrightarrow{\sigma}{}^{n} \Sigma_2, e_2$, *then there exists a permutation of names* $\pi : \mathcal{N} \to \mathcal{N}$, *fixing the domain of* Σ, *such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $e_2 = \pi(e_1)$.

2. *If f,* f_1, f_2 *are closures such that* $(\Sigma, \sigma), f \longmapsto^n (\Sigma_1, \sigma_1), f_1$ *and* $(\Sigma, \sigma), f \longmapsto^n (\Sigma_2, \sigma_2), f_2$, *then there exists a permutation of names* $\pi : \mathcal{N} \to \mathcal{N}$, *fixing the domain of* Σ, *such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $\sigma_2 = \pi(\sigma_1)$, *and* $f_2 = \pi(f_1)$.

PROOF. The most important case is when $n = 1$, the rest follows by induction on *n*, by using the property that if $\Sigma, e \xrightarrow{\sigma}{}^{n} \Sigma', e'$, then $\pi(\Sigma), \pi(e) \xrightarrow{\sigma}{}^{n} \pi(\Sigma'), \pi(e')$ (for expressions), and if $(\Sigma, \sigma), f \longmapsto^n (\Sigma', \sigma'), f'$, then $(\pi(\Sigma), \pi(\sigma)), \pi(f) \longmapsto^n (\pi(\Sigma'), \pi(\sigma')), \pi(f')$ (for closures).

We present the proof for the first statement; the second one is trivial, as there are no primitive closure constructors that introduced fresh names. The proof proceeds by decomposing the term *e* into a context *E* and a redex *r*. The only important case is when

$r = \textbf{choose}\ \nu X{:}A.\ e$. Then it must be $e'_1 = [X_1/X]e$, $e'_2 = [X_2/X]e$, and $\Sigma_1 = (\Sigma, X_1{:}A)$, $\Sigma_2 = (\Sigma, X_2{:}A)$, where X_1 and X_2 are fresh names. Obviously, the involution $(X_1\ X_2)$ which swaps these two names has the required properties. □

7 Related work

Dynamic binding has been inadvertently introduced in the first versions of LISP, but then became a feature, rather than a bug, in the subsequent implementations and dialects. For example, a formalized proposal for dynamic binding in the untyped setting of LISP, can be found in [13]. For semantic treatment of dynamic binding we refer to [19]. Dynamic binding has been considered in typed calculi as well. An example is a calculus λN, developed in [4, 5], for the purposes of explaining certain features of object-oriented programming. The λN-calculus is related to our system in that both use names, but in a slightly different way. The dynamic variables of λN are introduced as ordinary λ-bound variables, but are then indexed by names to distinguish the various values that can be assigned to them. This type system, however, does not prevent reading from uninitialized dynamic variables. Implicit parameters as introduced in [16] are similar to dynamic variables, with the restriction to be used only in a first-order way. Implicit parameters has been proposed in [10] as a convenient mechanism for representing global variables in a monadic language.

The dynamic allocation of memory cells and the typing annotations that we use in the presented calculus somewhat resemble the constructs in calculi with region-based memory management [27, 3, 28, 2]. The exact correspondence between the two remains future work.

Monadic type systems have been used to represent effects related to state, almost since the inception of monads for denotational semantics by Moggi [17, 18]. For example, Wadler in [29, 30, 31], describes how monads can be used to model various effects, among which is state with destructive update. A thorough description of state and it's practical implementation in a monadic language, can be found in Launchbury and Peyton Jones' [14].

The calculus of dynamic binding that we presented in this paper is a specific instantiation of a more general system of effects that we described in [21]. Both calculi use the natural deduction and proof-assignment for a variant of S4 modal logic, that are developed by Pfenning and Davies in [23], and extend them with the notion of names. Pfenning and Davies in [23] and Kobayashi in [12] present a decomposition of a monad in terms of modalities $\Box$ and $\Diamond$. Kobayashi further uses $\Diamond$ to model global state, but the possibility of using $\Box$ for marking state-related effects is not explored.

We have previously considered names in the necessitation fragment of S4 (but not in the possibility fragment), for purposes of meta-programming and higher-order syntax in [20]. The system described in that work is very similar to our calculus for dynamic binding from Section 2, with some important differences: (1) Dynamic binding allows a map with a type $A \to \Box_D A$ as shown in Example 4, which gives a coercion from values into computations; these are prohibited in meta-programming, (2) Dynamic binding only substitutes closed terms for names upon name-dereference, while meta-programming substitutes open terms for names in boxed expressions, (3) Functions and ν-abstractions in dynamic binding may only dereference names in suspended positions. In [20], we did not consider nominal possibility for purposes of meta-programming, but it should be possible to do so. We should also note that the developments in the current paper, as well as in [20], were directly motivated by the work on Nominal Logic and FreshML by Pitts and Gabbay [9, 25, 24, 8] which introduces names as urelements of Fraenkel-Mostowski set theory. The necessitation fragment of S4, and the corresponding $\lambda^{\Box}$-calculus, but without names, have also been considered for purposes of staged computation [6, 33], and run-time code-generation [15, 32].

8 Conclusions and future work

In this paper, we introduce a λ-calculus capable of modeling memory allocation, initialization, destructive and non-destructive update. It is based on the modal λ-calculus corresponding to the variant of intuitionistic modal logic S4, as developed by Pfenning and Davies in [23]. We extend this calculus with names, which are labels that can be dynamically introduced into the computation, and serve as theoretical abstraction for store locations. Store locations in this setup should be understood as chunks of memory of size that is appropriately determined by the type of the location.

In our calculus, the allocation of uninitialized memory is modeled by dynamic introduction of names. Non-destructive update is obtained using the necessitation fragment of the calculus, and destructive update is obtained with the possibility fragment.

Each term is associated by the type system with its support set; that is, the set of names that the term may dereference. The type system features two families of nominal modal types: $\Box_C A$, which classify suspended computation of type A which may dereference names from the set C, and $\Diamond_C A$ which classify suspended computations of type A which assign values to the names in C.

Computations of type $\Box_C A$ may only be executed in an environment in which the names from C are provided with definitions by means of some explicit substitution. Because substitutions have delimited scope, this fragment of the calculus implements non-destructive update, i.e. dynamic binding. When the substitutions are associated with the $\Diamond_C A$ modal types, then their scope is not delimited, but extends till the end of the program. That way, substitutions model global store.

We should mention, however, that our calculus for state is probably not very practical, because the support annotations in types may be too verbose. Therefore, a significant future work will be to investigate type and support inference in the system (which may be similar to region and effect inference algorithms of [26, 2]). In a previous work [20], we have considered explicit support polymorphism as a way to make functions applicable to arguments with various support annotations. This line of work may continue with developing, for example, existential quantification over support sets, or even monadic abstraction in the style of [22, 1]. It would be interesting to see if these abstractions can be used to embed into our calculus the standard monad of state [30, 31, 14] where the typings of allocations, reads and writes are not indexed by names. We should also consider derived language constructs which could capture the patterns of use of the calculus, to provide appropriate levels of abstractions, relevant to the considered applications. All these mentioned research questions are also applicable to the general case of arbitrary effects, and not only to effects related to state.

9 Acknowledgments

The author would like to thank Frank Pfenning for the numerous discussions and his suggestions regarding the topic of this paper.

10 References

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

[2] L. Birkedal and M. Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001.

[3] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Symposium on Principles of Programming Languages, POPL'96*, pages 171–183, St. Petersburg Beach, Florida, 1996.

[4] L. Dami. Functional programming with dynamic binding. In D. Tsichritzis, editor, *Object Applications*, pages 155–172. Technical Report, University of Geneva, 1996.

[5] L. Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1998.

[6] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages, POPL'96*, pages 258–270, St. Petersburg Beach, Florida, 1996.

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

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

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

[10] J. Hughes. Global variables in Haskell. To appear in the Journal of Functional Programming.

[11] R. B. Kieburtz. Taming effects with monadic typing. In *International Conference on Functional Programming, ICFP'98*, pages 51–62, Baltimore, Maryland, 1998.

[12] S. Kobayashi. Monad as modality. *Theoretical Computer Science*, 175(1):29–74, 1997.

[13] J. Lamping. A unified system of parameterization for programming languages. In *Conference on LISP and Functional Programming*, pages 316–326, Snowbird, Utah, 1988.

[14] J. Launchbury and S. L. P. Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[15] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Conference on Programming Language Design and Implementation, PLDI'96*, pages 137–148, 1996.

[16] J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages, POPL'00*, pages 108–118, Boston, Massachusetts, 2000.

[17] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, LICS'89*, pages 14–23, Asilomar, California, 1989.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[19] L. Moreau. A syntactic theory of dynamic binding. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 727–741. Springer, 1997.

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

[21] A. Nanevski. A modal calculus for effect handling. Technical Report CMU-CS-03-149, Computer Science Department, Carnegie Mellon University, June 2003.

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

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

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

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

[26] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[27] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages, POPL'94*, pages 188–201, Portland, Oregon, 1994.

[28] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[29] P. Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages, POPL'92*, pages 1–14, Albequerque, New Mexico, 1992.

[30] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[31] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming, ICFP'98*, pages 63–74, Baltimore, Maryland, 1998.

[32] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *Conference on Programming Language Design and Implementation, PLDI'98*, pages 224–235, Montreal, Canada, 1998.

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

[34] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.