A Nominal Modal Calculus of Effects

Aleksandar Nanevski

June 2003
CMU-CS-??-???

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In their purest formulation, monads are used in functional programming for two purposes: (1) to hygienically propagate effects, and (2) to globalize the effect scope – once an effect occurs, the purity of the surrounding computation cannot be restored. As a consequence, monadic typing does not provide very naturally for the practically important ability to handle effects, and there is a host of previous works directed toward remedying this deficiency. It is mostly based on extending the monadic framework with further extra-logical constructs to support handling.

In this paper we adopt a different approach, founded on the observation of Pfenning and Davies that an abstract monad can be decomposed in terms of S4 modal operators for possibility $\Diamond$ and necessity $\Box$. Our idea is to use the $\Box$ modality for hygienic propagation of effects, and the $\Diamond$ modality for globalization of effect scope. Those effects which admit a natural notion of handling can be encoded using $\Box$; since they are not global, there is no need to push them under $\Diamond$. Those effects which do not need to be handled, can be encoded using $\Diamond$.

To put this idea to work, we require a whole family of modal operators, indexed by the effects that they track. Thus, we further endow the calculus with the semantics category of names which very naturally extend modal logic, and serve to identify individual effects. The type system can then easily track each effect and allow for evaluation only the terms whose effects are guaranteed to be handled.

Keywords: modal logic, effect-systems, exceptions, composable continuations, dynamic binding, state

Contents

1 Introduction

It is a well-established principle in the area of functional programming that monads are a type-theoretic equivalent to effects and effect systems [Wad98]. A monad, as described by Moggi [Mog89, Mog91] and Wadler [Wad92, Wad95], serves two purposes: (1) it marks the impure code segments of a program, supporting in this way a disciplined propagation of effects, and (2) it single-threads the program so that the order

of effects is explicit[1], and more importantly, the scope of the effects becomes global (i.e., once introduced, the effect "holds" till the end of the program).

However, neither monads nor the effect systems, in their purest formulation, can express the important, and in programming practice ubiquitous, ability of *handling effects*; that is, restoring purity to an impure computation by means of some action. In fact, this very definition of handling effects is in direct contradiction to the above property (2) that effects should have global scope. The expressiveness of effect handling, when required, has to be endowed upon the monadic framework by means of additional constructs [Fil94, Fil96, Fil99, Kie98].

In this paper we present a novel effect system, based on modal logic, which is capable of representing effect handling. It is founded on the observation of Pfenning and Davies in [PD01] that an abstract monad can be deconstructed as a composition $\Diamond\Box$ of modalities for possibility $\Diamond$ and necessity $\Box$ of modal logic S4. We build on this result by recognizing that, informally: (1) $\Box$ takes the duty of marking impure code segments and enforcing effect propagation discipline, and (2) $\Diamond$ takes the duty of single-threading the program and globalizing the scope of effects.

We can use the two independent modal type constructors to ascribe typings that are more precise than those of a monadic calculus. Those effects which admit a natural notion of handling may be encoded using only the $\Box$ modality; since the scope of such effects is not global, they need not be forced under a $\Diamond$. For those effects where the actual single-threading and scope extension is required (e.g., destructive state update) the $\Diamond$ modality can be employed.

To label and track specific effects, we use a distinct semantic category of *names* which naturally extends the modal calculus [Nan02]. Thus, similar to indexed monads in [Wad98], we will have a type $\Box_C A$ classifying *suspended* computations of type A which may, in the course of eventual execution, raise effects listed in the (possibly ordered) set of names C. A coercion from $\Box_C A$ to A is possible but only through a *handler* which is equipped to deal with names in C. For example, in the particular case of effects related to state, the type $\Box_C A$ is assigned to terms which may *read* from the locations named in the set C. In this case we can also define the indexed possibility operator $\Diamond_C$ which will classify terms of type A capable of *writing* into the locations from C.

This approach in formulating effects follows the introduction/elimination pattern of natural deduction; effects are introduced by their introduction forms, and are eliminated by their handling forms. This is particularly appealing because it may uncover the logical content of the particular effects in question (although we do not explore this further). Guided by this framework, we develop as an example a novel calculus of exceptions [dG95, PRH+99], in which each exception is associated with a name, raising exceptions is the introduction form and handling exceptions is the elimination form. We also present novel calculi for catch-and-throw [Nak92, KS02], composable continuations [Fel88, DF89, DF90, Mur92, Wad94, HDA94, Kam00a, Kam00b], dynamic binding [Mor97, Dam96, Dam98] and state.

2 Nominal necessity

2.1 $\lambda^{\Box}$-calculus

The starting point for the development of our language of effects is the $\lambda^{\Box}$-calculus of [PD01, DP01]. The $\lambda^{\Box}$ is the proof-term system for the necessitation fragment of modal logic S4, and it was first considered in functional programming in the context of specialization for purposes of run-time code generation [DP96, WLP98, WLPD98]. The syntax of $\lambda^{\Box}$ is summarized below, where we use b to stand for a predetermined set of base types.

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

[1] and preserved in substitution

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

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

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

Figure 1: Typing rules of $\lambda^{\Box}$.

The most important feature of the calculus is the type constructor $\Box$ which is referred to as *modal necessity*, as in the S4 modal logic it is a necessitation modifier on propositions [PD01]. For the purposes of this paper, a useful operational intuition is to consider the type $\Box A$ as a type of *suspended expressions of type A*. In contrast, the non-modal type A would be populated with only *executable expressions*. In this sense, rather than suspending expressions by enclosing them under a λ-binder, as it is customarily done in functional programming, we use a separate language construct for that. This gives us an orthogonal and more appropriate abstraction mechanism that we will build on in subsequent sections.

The type system of $\lambda^{\Box}$ (Figure 1) distinguishes between two variable contexts: Γ for variables bound to executable expressions, and Δ for variables bound to suspended expressions. The introduction and elimination forms of the type constructor $\Box$ are the term constructors **box** and **let box**, respectively. Operationally, the term constructor **box** suspends the evaluation of its argument expression e, and wraps it into a thunk **box** e which can be then be further manipulated by the rest of the program. Note that the typing rule for **box** prohibits e to refer to variables from Γ; it is not possible to coerce values into suspensions. This is counter-intuitive to our interpretation of the modal calculus and we will remedy it in subsequent sections. 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 `exp2` below takes an integer argument n and builds a suspension for computing 2^n.

```
fun exp2 (n : int) : □int =
    if n = 0 then box 1
    else
      let box u = exp (n - 1)
      in
          box (2 * u)
      end

- e5 = exp2 5;
val e5 = box (2 * 2 * 2 * 2 * 2 * 1) : □int
```

■

In the elimination form **let box** $u = e_1$ **in** e_2, the bound variable u belongs to the context Δ of expression variables, but it can be used in e_2 in both suspended positions (i.e., under a box) and executable positions. This way we can compose suspended programs, but also explicitly force their evaluation. In the above example, we can force the evaluation of `e5` in the following way.

```
- let box u = e5 in u;
val it = 32 : int
```

Primitive reduction

$$\frac{}{(\lambda x{:}A.\ e)\ v \longrightarrow [v/x]e} \qquad \frac{}{\textbf{let box}\ u = \textbf{box}\ e_1\ \textbf{in}\ e_2 \longrightarrow [e_1/u]e_2}$$

Evaluation

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

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

To formalize the above operational intuition about the calculus, we employ an evaluation context operational semantics in the style of Wright and Felleisen [WF94]. We have decided on a call-by-value evaluation strategy which, in line with our interpretation of boxed expressions as suspended code, prohibits reductions under box; boxed expressions are considered values. This choice is by no means canonical, but is necessary for the purposes of this paper. The formalization relies on the definitions of redex and evaluation context introduced below.

Values	v	$::=$	$c \mid \lambda x{:}A.\ e \mid \textbf{box}\ e$
Redexes	r	$::=$	$v_1\ v_2 \mid \textbf{let box}\ u = v\ \textbf{in}\ e$
Evaluation contexts	E	$::=$	$[\,] \mid E\ e_1 \mid v_1\ E \mid \textbf{let box}\ u = E\ \textbf{in}\ e$

Each expression e can be decomposed uniquely as $e = E[r]$ where E is an evaluation context and r is a redex. To define a small-step operational semantics of the calculus, 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 primitive reduction and the evaluation relation for call-by-value $\lambda^{\Box}$ are presented in Figure 2.

2.2 Names as markers for effects

The necessitation operator $\Box$ from the $\lambda^{\Box}$-calculus is in fact a *comonad*, and recently several authors have made an argument that comonads too, in addition to monads, can be used to represent certain kinds of effects. For example, [Kie99] argues that comonads are more appropriate than monads to represent effects which arise from the environment. Also, [Par00] identifies comonads as encoding computations which expect additional arguments from the environment.

In this section, we present a calculus that we hope further strengthens the case for comonads. The idea is to extend $\lambda^{\Box}$ with the notion of *names*. Names are labels which provide a formal abstraction for tracking effects. Each effect will be assigned a name, and if an effect appears in a suspended term, then the corresponding $\Box$-type will be indexed by that name. For example, if we have an exception X, then a suspended term of type A which may raise this exception, will be given a type $\Box_X A$, and we also provide coercions from $\Box_X A$ to A which would represent exception handlers for X. In this sense, the comonad of modal necessity, when endowed with names, not only represents effects arising from the environment, but in fact marks effects whose scope is not global – effects which are handleable. Also, the interaction of the environment with the comonad and the aforementioned passing of additional arguments, can be very diverse – different effects will have different notions of handling.

The described indexing of the modal operator with names is similar to the one found in the monadic language from [Wad98], 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 (and hence, generation of new effects), and establish a typing discipline for it. Having mentioned this idea to provide some intuition toward our overall goal, we proceed to introduce our calculus in stages. Rather than formally tying names to effects immediately, we now present a limited fragment that is intended only to account for dynamic introduction of names and for name propagation. This fragment will be a common part of all the effect calculi we develop next. How names relate to effects, and how various effects are raised and handled will be discussed in the subsequent sections.

Our language is very similar to the one we presented in [Nan02]. We start by explaining the syntax and various syntactic conventions.

$$
\begin{array}{lrcl}
\textit{Names} & X & \in & \mathcal{N} \\
\textit{Supports} & C, D & ::= & \cdot \mid C, X \\
\textit{Types} & A & ::= & b \mid A_1 \to A_2 \mid A_1 \nrightarrow A_2 \mid \Box_C A \\
\textit{Terms} & e & ::= & u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \\
 & & & \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \mid \\
 & & & \nu X{:}A.\ e \mid \mathbf{choose}\ e \\
\textit{Variable contexts} & \Delta & ::= & \cdot \mid \Delta, u{:}A[C] \\
\textit{Name contexts} & \Sigma & ::= & \cdot \mid \Sigma, X{:}A
\end{array}
$$

Just like $\lambda^{\Box}$, our calculus makes a distinction between levels of suspended and executable expressions. The two are separated by a modal type constructor $\Box$, except that now we have a whole family of modal type constructors – one for each *finite* sequence of names C, where the names are drawn from a countably infinite universe of names $\mathcal{N}$. As already hinted before, the type $\Box_C A$ classifies suspended expressions which may raise any of the effects whose names are in C. The sequence C is referred to as a *support* of such expressions. We will also consider a partial ordering $\sqsubseteq$ on supports, with the empty support as the smallest element. The definition of the ordering will differ depending on the particular effects being considered.

Because now suspended expressions can contain effects, we extend the typing assignments in the context Δ to keep track not only of the typing, but also of 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. Furthermore, as already commented in the previous section, we would like to enable coercion of values into suspended expressions, so we join the two contexts of the $\lambda^{\Box}$ calculus into one. The context Γ is now considered part of Δ declaring variables with explicitly empty support. Correspondingly, we will frequently abbreviate $x{:}A[\,]$ as $x{:}A$. This decision logically corresponds to identifying the types A and $\Box A$, where the index support in the later type is empty. Equivalently, it can be viewed as imposing monotonicity on the Kripke frame in the possible worlds semantics for the S4 modal logic [PD01]. Of course, because now we have a multitude of modal operators corresponding to various supports that can be used as indices, this move does not collapse the whole hierarchy of modal types.

A further change from $\lambda^{\Box}$ is an addition of the context Σ which declares the names (and their types) which are currently active in the program. Because the types of our calculus depend on names, we must impose some conditions on well-formedness of contexts. A 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 types of the new calculus now include the family $A \nrightarrow B$ whose introduction and elimination forms are $\nu x{:}A.\ e$ and **choose** e. These constructs are used to dynamically introduce fresh names into the calculus. For example, the term $\nu X{:}A.\ e$ binds a name X of type A that can subsequently be used in e. Because names stand for effects, this construct really declares a new effect, and enables e to raise it and handle it. Whatever e does with X, though, we will ensure through the type system that the result of the evaluation of e does not depend on X; we must prevent X to escape the scope of its introduction form. The ν-abstraction will be a value in our calculus. In particular, it will suspend the evaluation of e. If we want to evaluate it, we must **choose** it. The term constructor **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.

Finally, enlarging an appropriate context by a new variable or a name is subject to the usual variable

conventions: the new variables and names are assumed distinct, or are renamed in order not to clash with already existing ones. Terms that differ only in the syntactic representation of their bound variables and names are considered equal. The binding forms in the language are $\lambda x{:}A.\ e$, $\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$ and $\nu X{:}A.\ e$. Capture-avoiding substitution $[e_1/x]e_2$ of expression e_1 for the variable x in the expression e_2 is defined to rename bound variables and names when descending into their scope. Given a term e, we denote by $\mathbf{fv}(e)$ the set of free variables of e. The set of names appearing in the type A is denoted by $\mathbf{fn}(A)$.

The typing judgment of the core fragment is

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

The judgment is hypothetical and works with two contexts: context of names Σ and context of variables Δ. Given an expression e, the judgment checks whether e has type A, and whether its effects are in the support C. The core fragment of the typing rules is presented in Figure 3, and we explained it next.

A pervasive characteristic of the type system is the *support weakening principle*; that is

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

Support of the expression e determines which effects e can raise, and therefore, which handlers can restore its purity. Consequently, the support weakening principle formally models a very intuitive property that if the effects of e can be handled by some handler, then they can be handled by a stronger handler as well. In particular, if e is effect-free, then it can be handled by any and all handlers; the empty support is the smallest element of the partial ordering $\sqsubseteq$.

A further property that we formally represent is that values of the language are effect free. Indeed, values obviously cannot raise any effects, simply because their evaluation is already finished. Therefore, the support of the values of our system will be empty, and according to the support weakening principle, it can then be weakened arbitrarily. This explains the explicit weakening in the hypothesis rule and the arbitrary support in the conclusions of the typing rules for λ- and ν-abstractions and for **box**.

λ-calculus fragment. The rule for λ-abstraction requires that the body e of the abstraction be pure; that is e has to match the empty support. This is not to say that e cannot contain any effects; it can, but only if they are suspended under a **box** (and correspondingly accounted for in the type of e). This parallels exactly the monadic type systems where function bodies must be pure, and effects can be raised only under a monad. On the other hand, because λ-terms are values, the support of the whole abstraction can be arbitrarily weakened, as explained before.

It is implicitly assumed that the argument type A is well-formed in name context Σ before it is introduced into the variable context Δ. Note further that the bound variable x is introduced into Δ with *empty* support, according to our decision to allow coercion of values into suspensions. Thus, x must always be bound to an effect-free expression. This will force us to commit to call-by-value evaluation strategy for the calculus; we must reduce function arguments to values (which are effect-free) before passing them on. The application rule 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. The $\Box$-elimination rule is a straightforward extension of the corresponding $\lambda^{\Box}$ rule. The only difference is that the bound expression variable u from the context Δ now has to be stored with its support annotation.

Names fragment. The rule for $\nu X{:}A.\ e$ must check e for well-typedness in a context Σ extended with the new name $X{:}A$. Similar to the λ rule, we require that e has empty support; all the eventual effects that e may raise must be boxed. The peculiarity of the ν constructor, however, is the further requirement that X does not appear in the type B. This ensures that X remains local to e; it can never escape the scope of its introducing ν in any observable way. The effect corresponding to X will either never be raised in the course of evaluation of e (i.e., it never appears in e or appears in some dead-code part of e), or all the occurrences of X are handled by an appropriate notion of handler.

Hypothesis

$$\frac{C \sqsubseteq D}{\Sigma; (\Delta, u{:}A[C]) \vdash u : A\,[D]}$$

λ-calculus

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

Modality

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

Names

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

Figure 3: Type system of the core fragment.

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.

Example 2 We can introduce the term constructor **let val** $x = e_1$ **in** e_2 into the calculus, with the following rule

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

Note that the construct is syntactic sugar for **let box** $u = (\lambda x.\ \textbf{box}\ e_2)\ e_1$ **in** u, rather than the usual $(\lambda x.\ e_2)\ e_1$. The complication arises because we have to box e_2 and make it pure, before we can put it under a λ-abstraction. Similarly, we introduce the term constructor **new** $X{:}A$ **in** e as an abbreviation for **let box** $u =$ **choose** $(\nu X{:}A.\ \textbf{box}\ e)$ **in** u, with the derived typing rule

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

■

Example 3 If C, C_1, C_2 and D are arbitrary supports such that $C_1 \sqsubseteq C$ and $C_2 \sqsubseteq C$, then the following terms are well-typed of *empty support*.

$$\begin{array}{l}
\lambda x{:}\Box_{C_1} A.\ \textbf{let box } u = x \textbf{ in box } u : \Box_{C_1} A \to \Box_C A \\
\lambda x{:}\Box A.\ \textbf{let box } u = x \textbf{ in } u : \Box A \to A \\
\lambda x{:}\Box_{C_1} A.\ \textbf{let box } u = x \textbf{ in box } (\textbf{box } u) : \Box_{C_1} A \to \Box_D \Box_C A \\
\lambda x{:}\Box_{C_1}(A \to B).\ \lambda y{:}\Box_{C_2} A.\ \textbf{let box } u = x \textbf{ in let box } v = y \textbf{ in box } (u\ v) \\
\qquad : \Box_{C_1}(A \to B) \to \Box_{C_2} A \to \Box_C B
\end{array}$$

The first term is an eta-expansion of the argument x. It shows that support weakening in boxed types is derivable. The second term illustrates that we can "unbox" and evaluate suspended expressions which are *effect free*; notice that the argument type has empty index support. The other two terms generalize the characteristic comonadic axioms of S4 modal necessity without names [DP01, PD01]. ■

Example 4 Anticipating Section 3.1, suppose that our language contains the term constructor **raise**, such that $\textbf{raise}_X\ e$ raises an exception X passing an argument e along (assuming that both X and e have the same type). If X is a name of type A, then the following term is well-typed.

$$\lambda x{:}\Box A.\ \textbf{let box } u = x \textbf{ in box } (\textbf{raise}_X\ u) : \Box A \to \Box_X A$$

Assume further that $e_1{:}B$ is a closed and exception-free term, and $e_2{:}A$ is a closed term which may raise the exception X. Then the expression

$$\textbf{choose } (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y} A.\ e_1)\ (\textbf{box raise}_Y\ e_2))$$

declares a new exception Y and then raises it within a suspension $(\textbf{box raise}_Y\ e_2){:}\Box_{X,Y} A$. In fact, because neither x nor Y appear in e_1, the type of the application will not depend on Y either. Actually, even more is true: the argument suspension will never even be forced; it is dead code. The ν-clause is therefore well-typed, of type $A \nrightarrow B$, and the whole expression is of type B. In Section 3.1 where we introduce exception handling, we would be able to present a more meaningful use of **choose** and ν. ■

Primitive reductions

$$\frac{}{\Sigma, (\lambda x.\ e)\ v \longrightarrow \Sigma, [v/x]e} \qquad \frac{}{\Sigma, \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 \longrightarrow \Sigma, [e_1/u]e_2}$$

$$\frac{Y \text{ fresh}}{\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) \longrightarrow (\Sigma, Y{:}A), [Y/X]e}$$

Evaluation

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

Figure 4: Operational semantics of the core fragment.

2.3 Operational semantics

The operational semantics of this basic fragment of our calculus is defined through the judgment

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

which relates an expression e with its one-step reduct e'. The relation is defined on expressions with no free variables. An expression e can contain effects, whose names must be declared in Σ, but it must have *empty support*. In other words, we only consider for evaluation those expressions whose effects are either suspended, or appear in a dead-code part, or are handled. The reduct e' can introduce new names into the computation, which will be accounted in the extended name context Σ'. However, the new names too, will mark effects which are either suspended, never raised or otherwise handled. The definition of the judgment relies on the notion of redexes and evaluation contexts below.

Values	v	::=	$\lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e$
Redexes	r	::=	$v_1\ v_2 \mid \mathbf{let\ box}\ u = v\ \mathbf{in}\ e \mid \mathbf{choose}\ v$
Evaluation contexts	E	::=	$[\,] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E$

The primitive reduction rules and the evaluation rules of the operational semantics are presented in Figure 4. They are identical to the rules for $\lambda^{\Box}$, except for the new reduction rule for **choose** $(\nu X{:}A.\ e)$, which extends the run-time name context Σ with a fresh name X before proceeding with the evaluation of e.

Example 5 As an illustration of the operational semantics of the calculus, we present the first two steps from the evaluation of the term from Example 4.

$$\begin{array}{ll} & (X{:}A), \mathbf{choose}\ (\nu Y{:}A.\ (\lambda x{:}\Box_{X,Y}A.\ e_1)\ (\mathbf{box\ raise}_Y\ e_2)) \\ \longmapsto & (X{:}A, Z{:}A), (\lambda x{:}\Box_{X,Z}A.\ e_1)\ (\mathbf{box\ raise}_Z\ e_2) \qquad \text{where } Z \text{ is a fresh name} \\ \longmapsto & (X{:}A, Z{:}A), e_1 \\ \longmapsto & \cdots \end{array}$$

■

2.4 Structural properties and type soundness

The rest of this section develops the basic properties of the calculus. We present them here, because the future extensions will all rely on the basic structure of these results.

Lemma 1 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$ *and* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Lemma 2 (Replacement)
If $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exist a type* B *such that*

1. $\Sigma; \Delta \vdash e : B\,[C]$, *and*
2. *if* Σ', Δ' *extend* Σ, Δ, *and* $\Sigma'; \Delta' \vdash e' : B\,[C]$, *then* $\Sigma; \Delta \vdash E[e'] : A\,[C]$

Lemma 3 (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* $A = A_1 \twoheadrightarrow A_2$, *then* $v = \nu X{:}A_1.\ e$ *and* $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$

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

Lemma 4 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longrightarrow \Sigma', e'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 5 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 6 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \longrightarrow \Sigma', e'$.

Lemma 7 (Unique decomposition)
For every expression e, *either:*

1. e *is a value, or*
2. $e = E[r]$ *for a unique evaluation context* E *and a redex* r.

Lemma 8 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, *then either*

1. e *is a value, or*
2. *there exists a term* e' *and a context* Σ', *such that* $\Sigma, e \longmapsto \Sigma', e'$.

Lemma 9 (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 the domain of* Σ, *such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $e_2 = \pi(e_1)$.

3 Example effect systems

3.1 Exceptions

3.1.1 Syntax and typing

The calculus presented thus far did not involve any concrete notions of effects. It was only capable of dynamic introduction and of propagation of effects, but not, in fact, of raising or handling them. In this section we extend our code fragment into a calculus of exceptions. The idea is to assign a name to each exception, which could then be propagated and tracked, by means of the core fragment. To be able to raise and handle exceptions, we need further constructs specific only to exceptions. Thus, we extend the syntax of our language in the following way.

$$\begin{array}{lrcl} \textit{Exception handlers} & \Theta & ::= & \cdot \mid Xz \to e, \Theta \\ \textit{Terms} & e & ::= & \ldots \mid \mathbf{raise}_X\ e \mid e\ \mathbf{handle}\ \langle \Theta \rangle \end{array}$$

Informally, the role of $\mathbf{raise}_X\ e$ is to evaluate e and than raise an exception X, passing the value of e along. On the other hand, $e\ \mathbf{handle}\ \langle \Theta \rangle$ evaluates e (which may raise exceptions), and all the raised exceptions are handled by the exception handler Θ.

An exception handler Θ is syntactically defined as a list of exception patterns, each of which refers to a different exception. More conceptually, we can regard the exception handler as a mapping from the set of names to the set of functions over terms

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

We will treat our handlers as being able to take action on *all* the exceptions; it is just that on some of them the action will be a specifically proscribed term, while on others the action will just involve propagation of the exception. For example, the empty handler $\langle\ \rangle$ handles all the exceptions by propagating them further.

Given a handler Θ, its domain $\mathbf{dom}(\Theta)$ is defined as the set

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

We only consider handlers with *finite* domains. A handler Θ with a finite domain has a finitary syntactical representation as a set of patterns $Xz \to e$ relating a name X from $\mathbf{dom}(\Theta)$ with the function $\Theta(X) : z \mapsto e$. We will frequently equate a handler and the set that represents it when it does not result in ambiguities. The operations that e can do on the term z in the above mapping are limited to only ordinary compositions with the term constructors from our language. Thus, it will be the case that the value of the function $\Theta(X)$ at a term e' is equal to $[e'/z]e$.

Example 6 Assuming X and Y are integer names, the following are well-formed expressions of the exception calculus with empty support.

$$\begin{array}{l} (1 - \mathbf{raise}_X\ \mathbf{raise}_Y\ 10)\ \mathbf{handle}\ \langle Xx \to x + 2, Yy \to y + 3 \rangle \\ (1 - \mathbf{raise}_X\ 0)\ \mathbf{handle}\ \langle Xx \to (2 - \mathbf{raise}_Y\ x) \rangle\ \mathbf{handle}\ \langle Yy \to y \rangle \\ (1 - \mathbf{raise}_X\ 0)\ \mathbf{handle}\ \langle Yy \to (2 - \mathbf{raise}_X\ y) \rangle\ \mathbf{handle}\ \langle Xx \to x + 1 \rangle \end{array}$$

The terms evaluate to 13, 0 and 1, respectively. The first term raises the exception Y and then handles it. The second raises X with value 0, which is then handled by the first handler, but this handler itself raises the exception Y with value 0, ultimately handled by the second handler. The third term raises X with value 0, which is propagated by the first handler, and then handled by the second handler. ■

The type system of the calculus of exceptions consists of two judgments: one for typing expressions, and another one for typing exception handlers. The judgment for expressions has the form

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

Exception handlers

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

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

Exception raising and handling

$$\frac{\Sigma; \Delta \vdash e : A\,[C] \qquad X \in C \qquad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{raise}_X\ e : B\,[C]} \qquad \frac{\Sigma; \Delta \vdash e : A\,[C] \qquad \Sigma; \Delta \vdash \langle\Theta\rangle : A[C] \Rightarrow A[D]}{\Sigma; \Delta \vdash e\ \mathbf{handle}\ \langle\Theta\rangle : A\,[D]}$$

Figure 5: Typing rules for exceptions.

and it simply extends the judgment from the core fragment presented in Section 2.2 with the new rules for **raise** and **handle**. The specific of the calculus is that the support C represents sets, collecting the exceptions that e is *allowed* to raise. Thus, $C \sqsubseteq D$ is defined as $C \subseteq D$ when C and D are viewed as sets (i.e., when the ordering and repetition of elements in these supports are ignored). By support weakening, e need not raise all the exceptions from its support C, but if an exception can be raised, then it must be in C. The judgment for exception handlers has the form

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

and the handler Θ will be given the type $A[C] \Rightarrow A[D]$ if: (1) Θ can handle exceptions from the support set C arising in a term of type A, and (2) during the handling, Θ is allowed to itself raise exceptions only from the support set D. The typing rules of both judgments are presented in Figure 5, and we briefly comment on them below.

An exception X can be raised only if it is accounted for in the support. Thus the rule for **raise** requires $X \in C$. The term $\mathbf{raise}_X\ e$ changes the flow of control, by passing e to the nearest handler. Because of that, the environment in which this term is encountered does not matter; we can type $\mathbf{raise}_X\ e$ by any arbitrary type B. In the rule for **handle**, the type and the support of the expression e must match the type and the domain support of the handler Θ. The exception handler $\langle\,\rangle$ only propagates whichever exceptions it encounters. Thus, if it is supplied an expression of support C it will produce an expression of the same support. To maintain the support weakening property, we allow the range support D of an empty handler to be a superset of C. Notice that the empty support handler may be assigned an arbitrary type A. The rule for nonempty exception handlers simply proscribes inductively checking each of the exception patterns in the handler. The type of each pattern variable z must match the type of the corresponding exception; this is the type of the value that the exception will be raised with. The handling terms e must all have the same type B, which would also be the type assigned to the handler itself.

Example 7 The function `tail` below computes a tail of the argument integer list, raising an exception `EMPTY:unit` if the argument list is empty. The function `length` uses `tail` to compute the length of a list. Note that the range type of `tail` is $\square_{\texttt{EMPTY}}\texttt{intlist}$. This is required because the body of `tail` raises an exception, and, as explained in the previous section, all the effects in function bodies must be boxed.

```
- choose (νEMPTY: unit.
  let fun tail (xs : intlist) : □EMPTYintlist =
      (case xs
         of nil => box (raiseEMPTY ())
          | x::xs => box xs)
      fun length (xs : intlist) : int =
          (1 + length (let box u = tail xs in u))
          handle <EMPTY z -> 0>
  in
    length [1,2,3,4]
  end);
val it = 4;
```

■

There are several points worth emphasizing about our calculus. First of all, exceptions in our calculus are not values and cannot be bound to variables. Correspondingly, they must be explicitly raised; raising a variable exception is not possible. Aside from this fact, when *local* exceptions are concerned (i.e., exceptions which do not originate from a function call, but are raised and handled in the body of the one and the same function), our calculus very much resembles Standard ML [MTHM97]. In particular, exceptions can be raised, and then handled, without forcing any changes to the type of the function. It is only when we want the function to propagate an exception so that it is handled by the caller, that we need to specifically mark the range type of that function with a $\Box$-type. This is in contrast to the general mechanism of monads [Wad95] where effects are not handleable, and therefore adding an effect to the body of a pure function invariably requires that the range type of the function be changed to a monadic type This in turn may require serious restructuring of the programs that use such a function.

Finally, our calculus presents really only a bare-bone theoretical foundation for the treatment of exceptions. As is probably the case with other effect calculi too, it is not very practical when compared to, say, Standard ML, exactly because the types are decorated with exception names. But, we believe that such a hurdle can be overcome; it is easier to hide the excess information, than to overcome the lack thereof. Possible solutions may involve universal and existential abstractions over support sets [Nan02], and deriving new type constructors out of the existing ones in order to obtain the practical level of abstraction.

3.1.2 Operational semantics

The operational semantics of the exception calculus is a simple extension of the semantics of the core fragment. The evaluation judgment has the same form

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

We only need to extend the syntactic categories of evaluation contexts and redexes, and define primitive reductions for the new redexes. First, we define new evaluation contexts.

$$\textit{Evaluation contexts} \quad E \quad ::= \quad \ldots \mid \mathbf{raise}_X\ E \mid E\ \mathbf{handle}\ \langle \Theta \rangle$$

We have already explained that each exception handler can handle all exceptions. It is only that some exceptions are handled in a specified way, while others are handled by simple propagation. This will simplify the operational semantics somewhat, because in order to find the handler capable of handling a particular **raise** we only need to find the nearest handler preceding this **raise**. For that purpose, we define a special subclass of evaluation contexts, called *pure evaluation contexts*.

Definition 10 (Pure evaluation contexts)
An evaluation context E is pure *if it does not contain any exception-handling constructs acting on the hole of the context. In other words, the syntactic category of pure evaluation contexts is defined as*

$$\textit{Pure contexts} \quad P \quad ::= \quad [\,] \mid P\ e_1 \mid v_1\ P \mid \mathbf{let\ box}\ u = P\ \mathbf{in}\ e \mid \mathbf{choose}\ P \mid \mathbf{raise}_X\ P$$

The idea of this definition is to be able to identify, within each evaluation context E, the handling construct (if any) which is the closest to the hole of E, as stated by the following lemma.

Lemma 11 (Evaluation context decomposition)
If E is an evaluation context, then either:

1. *E is a pure context, or*
2. *there exist unique evaluation context E' and pure context P' such that $E = E'[P'\ \mathbf{handle}\ \langle\Theta\rangle]$.*

This definition and lemma provide us with enough equipment to define the new redexes and the primitive reduction on them.

$$\textit{Redexes} \quad r \quad ::= \quad \ldots \mid v\ \mathbf{handle}\ \langle\Theta\rangle \mid P[\mathbf{raise}_X\ v]\ \mathbf{handle}\ \langle\Theta\rangle$$

$$\frac{}{\Sigma, v\ \mathbf{handle}\ \langle\Theta\rangle \longrightarrow \Sigma, v} \qquad \frac{}{\Sigma, P[\mathbf{raise}_X\ v]\ \mathbf{handle}\ \langle\Theta\rangle \longrightarrow \Sigma, \Theta(X)(v)}$$

The first reduction exploits the fact that values are exception free, and therefore simply fall through any handler. The second reduction chooses the closest handler for any particular raise. It also requires that only values be passed along with the exceptions; the operational semantics demands that before an exception is raised, its argument must be evaluated. If it so happens that the evaluation of the argument raises another exception, this later one will take precedence and actually be raised. This is already illustrated in the first term from Example 6, where it is the exception Y which is raised and eventually handled.

3.1.3 Structural properties and type soundness

Lemma 12 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$, 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 $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : B[D'] \Rightarrow B[D]$, then $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : B[D'] \Rightarrow B[D]$*

Lemma 13 (Handler application)
Let $\Sigma; \Delta \vdash \langle\Theta\rangle : A[C] \Rightarrow A[D]$ and $\Sigma; \Delta \vdash e : B\,[D]$, where $X \in C$ and $X{:}B \in \Sigma$. Then $\Sigma; \Delta \vdash \Theta(X)(e) : A\,[D]$

Lemma 14 (Replacement)

1. *If $\Sigma; \Delta \vdash P[e] : A\,[C]$, then there exist a type B and a support $D \sqsubseteq C$ 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 $\Sigma; \Delta \vdash P[e'] : A\,[C]$*
2. *If $\Sigma; \Delta \vdash E[e] : A\,[C]$, then there exist a type B and a support D 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 $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$*

Lemma 15 (Canonical forms)
Let v be a closed value such that $\Sigma; \cdot; \cdot \vdash v : A\,[C]$. 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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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

Lemma 16 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 17 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 18 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, then there exists a term e' and a context Σ', such that $\Sigma, r \longrightarrow \Sigma', e'$.

Lemma 19 (Unique decomposition)
For every expression e, either:

1. *e is a value, or*
2. *$e = P[\mathbf{raise}_X\ v]$, for a unique pure context P, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

Lemma 20 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Lemma 21 (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 the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

3.2 Catch-and-throw calculus

3.2.1 Syntax and typing

In the catch-and-throw calculus, names are associated with labels to which the program can jump. Informally, **catch** establishes a destination point for a jump and assigns a name to it, and **throw** jumps to the established point. The exact syntax of the two constructs is defined as follows.

$$\textit{Terms}\quad e \quad ::= \quad \ldots \mid \mathbf{throw}_X\ e \mid \mathbf{catch}_X\ e$$

The **throw** and **catch** can be viewed as restrictions of **raise** and **handle**; **catch** handles a **throw** by immediately returning the value associated with the throw.

The typing judgment $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$ establishes that e has type A and may throw to destination points whose names are listed in the support C. The supports are sets, rather than sequences, just like in the calculus of exceptions. The typing rules of the calculus are presented in Figure 6. A **throw** to a destination point is allowed only if the destination point is present in the support set. A **catch** establishes a destination point by placing it in the support set against which the argument expression is checked.

Throw and catch

$$\frac{\Sigma;\Delta \vdash e : A\,[C] \qquad X \in C \qquad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{throw}_X\ e : B\,[C]} \qquad \frac{\Sigma;\Delta \vdash e : A\,[C, X] \qquad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{catch}_X\ e : A\,[C]}$$

Figure 6: Typing rules for the catch-and-throw calculus.

Example 8 The following example is adopted from [Kam00a] which introduces a catch-and-throw calculus among its preliminaries. Instead of names, this calculus uses *label variables* bound by **catch**, which admit α-renaming. For example, in the term

$$\mathbf{catch}_X\ ((\lambda f.\ 1 + \mathbf{catch}_X\ (f\ 0))\ (\lambda y.\ \mathbf{throw}_X\ y))$$

the label X used in $\mathbf{throw}_X\ y$ refers to the label X bound by the *outside* **catch**, because it is in the scope of the outside **catch**, and not the inside one. The whole term can then be rewritten as

$$\mathbf{catch}_X\ ((\lambda f.\ 1 + \mathbf{catch}_Y\ (f\ 0))\ (\lambda y.\ \mathbf{throw}_X\ y))$$

in which the label bound by the inside **catch** is α-renamed to avoid capture. The value of the term is 0, because the addition with 1 is skipped over by a **throw**. The evaluation strategy which matches catches and throws in this way is referred to as *static*. In contrast, there is also a *dynamic* strategy which allows dynamic binding of $\mathbf{throw}_X\ y$ to the inside **catch**, resulting in 1 as the value of the term. The dynamic strategy better corresponds to practice, but it is not explored in the framework of [Kam00a].

In our catch-and-throw calculus, we treat labels as names which are introduced by a separate constructor ν, and are not bound by **catch**. Hence, the correspondence between throws and catches does not have to be established based on scoping and α-renaming. In fact, we can write terms in our calculus which behave as either the static or the dynamic version above, while still retaining the safety property that no **throw** remains uncaught. For example, the static translation would be

```
choose (νX:int.
  (λf:int->□_X int.
            let box u = f 0
            in
                catch_X (1 + u)
            end) (λy:int. box (throw_X y)))
```

As in [Kam00a], this term evaluates to 0. In the dynamic variant, the catch is pushed further inside resulting with the following program, whose value is 1.

```
choose (νX:int.
  (λf:int->□_X int.
                let box u = f (box 0)
                in
                    1 + catch_X u
                end) (λy:int. box (throw_X y)))
```

Notice that in the sense of [Kam00a], the name X used in $\mathbf{throw}_X\ y$ was in both the examples outside of the scope of the **catch** that handled it. ■

Example 9 The program segment below defines a recursive function for multiplying elements of an integer list. If an element is found to be equal to 0, then the whole product will be 0, so rather than uselessly performing the remaining computation, we terminate by an explicit **throw** outside of the recursive function.

```
- choose (νEXIT:int.
   let fun mult (xs : intlist) =
          box (case xs
                 of nil => 1
                  | x::xs =>
                     if x = 0 then throw_EXIT 0
                     else (let box u = mult xs in (x * u))
   in
       catch_EXIT (let box u = mult [2, 1, 0, 3] in u)
   end);

val it = 0 : int
```

■

3.2.2 Operational semantics

The evaluation judgment of the catch-and-throw calculus is again a straightforward extension of the evaluation judgment $\Sigma, e \longmapsto \Sigma', e'$ of the core fragment from Section 2.2. We first need to define the new redexes, corresponding to the new **catch** and **throw** constructs, and extend the syntactic category of evaluation contexts.

$$\begin{array}{llll} \textit{Redexes} & r & ::= & \ldots \mid \mathbf{catch}_X\ v \mid \mathbf{catch}_X\ E[\mathbf{throw}_X\ v] \\ \textit{Evaluation contexts} & E & ::= & \ldots \mid \mathbf{catch}_X\ E \mid \mathbf{throw}_X\ E \end{array}$$

In the redex $\mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$ it is assumed that the context E does not contain a $\mathbf{catch}_X$ phrase acting on the hole of E. The primitive reductions on the new redexes are defined as follows.

$$\frac{}{\Sigma, \mathbf{catch}_X\ v \longrightarrow \Sigma, v} \qquad \frac{}{\Sigma, (\mathbf{catch}_X\ E[\mathbf{throw}_X\ v]) \longrightarrow \Sigma, v}$$

Similarly to the exception calculus, values simply fall through the **catch**, and every **throw** is caught by the closes surrounding **catch** with the appropriate name. The operational semantics of catch-and-throw requires that only values be passed along a **throw**. Thus, of possibly nested throws, only the last one will actually be subject to catching.

3.2.3 Structural properties and type soundness

Lemma 22 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$ *and* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Lemma 23 (Replacement)
If $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exist a type* B *and a support* D *such that*

1. $\Sigma; \Delta \vdash e : B\,[D]$, *and*
2. *if* Σ', Δ' *extend* Σ, Δ *and* $\Sigma'; \Delta' \vdash e' : B\,[D]$, *then* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

Lemma 24 (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* $A = A_1 \twoheadrightarrow A_2$, *then* $v = \nu X{:}A_1.\ e$ *and* $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$

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

Lemma 25 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longrightarrow \Sigma', e'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 26 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 27 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \longrightarrow \Sigma', e'$.

Lemma 28 (Unique decomposition)
For every expression e, *either:*

1. e *is a value, or*
2. $e = E[\mathbf{throw}_X\ v]$, *for a unique context* E *which does not catch* X, *or*
3. $e = E[r]$ *for a unique evaluation context* E *and a redex* r.

Lemma 29 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, *then either*

1. e *is a value, or*
2. *there exists a term* e' *and a context* Σ', *such that* $\Sigma, e \longmapsto \Sigma', e'$.

Lemma 30 (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 the domain of* Σ, *such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $e_2 = \pi(e_1)$.

3.3 Composable continuations

3.3.1 Syntax and typing

Similar to the catch-and-throw calculus, composable continuations use names to label destination points to which a program can jump. A destination point for a jump is established with the construct **reset** which also assigns a name to it. The jump is performed by **shift**. The exact syntax of the calculus is defined as follows.

$$\textit{Terms} \quad e \quad ::= \quad \ldots \mid \mathbf{shift}_X\ k.\ e \mid \mathbf{reset}_X\ e$$

The differences from the catch-and-throw calculus arise from the following characteristic: unlike **throw**, the construct $\mathbf{shift}_X\ k.\ e$ captures the part of the environment between the source and the destination points of the jump and binds it to a variable k; k may then be used to compute the value of e that is passed along with the jump. It is important that the evaluation of e is undertaken in the changed environment from which the part captured in k has been *removed*. More specifically, e itself will not be able to shift to destination points which were defined in the captured and removed part.

It is the expressions bound to k that is actually referred to as a *composable continuation* (and other names in use are: partial continuation, delimited continuation and subcontinuation). Ordinary calculus of continuations [Lan65, SW74, Rey72, SF90b, Fil89, Gri90, DHM91, FFKD86, Thi97] can be viewed as a calculus of composable continuations in which all the jumps have a unique destination point, predefined to be at the beginning of the program. In both calculi, continuations are functions whose range type is equal to the type of the destination point. But, in the special case of ordinary continuations, this type is necessarily $\bot$, and that is why ordinary continuations cannot be composed in any non-trivial way.

$$\frac{\Sigma;(\Delta, k{:}\Box_{C,X} B \to \Box_{C,X} A) \vdash e : A\,[C] \qquad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{shift}_X\ k.\ e : B\,[C, X]} \qquad \frac{\Sigma;\Delta \vdash e : A\,[C, X] \qquad C \sqsubseteq D \qquad X{:}A \in \Sigma}{\Sigma;\Delta \vdash \mathbf{reset}_X\ e : A\,[D]}$$

Figure 7: Typing rules for composable continuations.

The typing judgment of our calculus for composable continuations is

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

It establishes that the expression e has type A and may shift to destination points whose names are listed in the support C. The typing rules for composable continuations are presented in Figure 7. In the case of composable continuations, it is the shifting to a name that is the notion of effect, and establishing a destination point is the notion of handling. Therefore, the type system should enable a **shift** to a destination point X only if X is accounted at the support C, placed there by a corresponding **reset**. The situation, however, is a bit more involved. As already mentioned, $\mathbf{shift}_X\ k.\ e$ evaluates e in a changed environment from which the destination points between X and the **shift** have been removed; thus e has to be typechecked against a support that is changed correspondingly.

The above argument indicates that in the calculus of composable continuations, the ordering of names in the support of a term is important. Unlike in the previous effect calculi where supports were simply sets, here we actually must exploit their list-like structure. To simplify matters, we allow a **shift** to a certain name only if that name is at the end of the support. This is accounted for in the typing rule for $\mathbf{shift}_X\ k.\ e$ which demands that X is the *rightmost* name in the support (C, X). After the environment delimited by X has been captured and removed, the destination point X is removed as well, and e is evaluated in the changed environment. Correspondingly, e is typechecked against support from which the rightmost X has been removed. If a shift is required to a name which is deeper down in C, it can still be done by performing a sequence of nested shifts in a last-in-first-out manner to all the names above. In that sense, we can actually view the supports of our calculus of continuations as *stacks*.

There are yet further important aspects of the typing rule for **shift** that need to be explained. The expression e computes the value to be passed along with the jump, so it must have the same type as the destination point X. Because the jump changes the flow of control, the immediate environment of the **shift** does not matter; we can type **shift** by an arbitrary type B. The domain and the range of the continuation k must match the source and the destination points of the **shift**, which in this rule have types B and A, respectively. The **shift** appears in the context of a support stack (C, X) and that is why k is placed into the context with the domain type $\Box_{C,X} B$. The range type of k is $\Box_{C,X} A$, meaning that the captured continuation *will not include* the delimiting $\mathbf{reset}_X$ (otherwise, this $\mathbf{reset}_X$ would have been a handler for X, making the range type $\Box_C A$). This is in contrast to most other calculi for composable continuations [Fel88, DF89, DF90, Mur92, Wad94, HDA94, Kam00a, Kam00b], which capture into the continuation the environment *up to and including* the delimiting **reset**. Our decision actually adds further expressiveness to the calculus, and we illustrate this point at the end of the section.

The typing rule for **reset** is much simpler. The construct $\mathbf{reset}_X\ e$ establishes a destination point X and allows the expression e to shift to X by placing X into the support. If e is a value, it immediately falls through to the destination point X, and thus e and X must have same types. We further allow an arbitrary weakening of supports in the conclusion of this rule, in order to satisfy the support weakening principle.

The partial ordering imposed on the family of supports is the trivial partial ordering with the empty stack as the smallest element: $C \sqsubseteq D$ holds iff $C = (\cdot)$ or $C = D$ as sequences.

Example 10 In this example, we consider the following terms from [DF89, Wad94], which evaluate to 121,

101 and 1121 respectively.

$$\begin{aligned} e_1 &= 1 + \mathbf{reset}\ (10 + \mathbf{shift}\ f.\ f\ (f\ 100)) \\ e_2 &= 1 + \mathbf{reset}\ (10 + \mathbf{shift}\ f.\ 100) \\ e_3 &= 1 + \mathbf{reset}\ (10 + \mathbf{shift}\ f.\ f\ 100 + f\ 1000) \end{aligned}$$

In our calculus, they are translated as follows (assuming X is a name of integer type).

```
e1 =  1 + reset_X (10 + shift_X f.
             let box u = f (f (box 100))
             in
                 reset_X u
             end)

e2 = 1 + reset_X (10 + shift_X f. 100)

e3 = 1 + reset_X (10 + shift_X f.
            let box u1 = f (box 100)
                box u2 = f (box 1000)
            in
                reset_X (u1 + u2)
            end)
```

■

Example 11 We next illustrate the intended operational semantics of the calculus by showing the evaluation of the expression e_1 from Example 10 in full.

```
    1 + reset_X (10 + shift_X f.
                    let box u = f (f (box 100))
                    in
                        reset_X u
                    end)
⟼ 1 + (let box u = f (f (box 100))
        in
            reset_X u
        end), where f = λx. let box v = x in box (10 + v)
⟼ 1 + (let box u = f (box (10 + 100))
        in
            reset_X u
        end)
⟼ 1 + (let box u = box (10 + (10 + 100))
        in
             reset_X u
        end)
⟼ 1 + reset_X (10 + (10 + 100))
⟼ 1 + reset_X (10 + 110)
⟼ 1 + reset_X 120
⟼ 1 + 120
⟼ 121
```

■

Notice in the above examples, that our translation of the original terms from [DF89, Wad94] use additional resets in the bodies of **let box**. These arise because our process of capturing discards the delimiting **reset** from the environment, but does not capture it into the continuation either (as we pointed out in our discussion of the typing rules), so we must explicitly put it back. While this may seem as an unnecessary complication at the moment, it does give us freedom to choose the place for putting back the delimiter.

Example 12 The program below is a particularly convoluted way of reversing a list, adopted from [DF89]. To reduce clutter, we abbreviate `(let box u = (-) in u)` simply as `unbox(-)`.

```
fun reverse (l : intlist) : intlist =
 choose (νX: intlist.
  let fun rev' (l : intlist) : □X intlist =
      case l
        of nil => box nil
      | (x::xs) =>
          let val y = rev' xs
          in
              box (shiftX c:□X intlint -> □X intlist.
                    resetX x :: unbox (c y))
          end
      box v = rev' l
  in
     resetX v
  end)
```

To understand `reverse`, it is instructive to view a particular evaluation of the helper function `rev'`. For example, `rev' [2, 1, 0]` produces

```
box (shiftX c3.
      resetX 2 :: unbox c3 (box shiftX c2.
        resetX 1 :: unbox c2 (box shiftX c1.
          resetX 0 :: unbox c1 (box nil)))
```

When prepended by a $\texttt{reset}_X$, unboxed and evaluated, this code uses the continuations c_i to accumulate the reversed prefix of the list. For example, c_3 is bound to the identity function, corresponding to the initial empty prefix; c_2 is bound to λ`x.let box u = x in box(2::u)`; c_1 is bound to λ`x.let box u = x in box(1::2::u)`, until finally the reversed list `[0,1,2]` is produced. ■

3.3.2 Operational semantics

As in the development of operational semantics for the previous effect calculi, here too we start by extending the notion of evaluation contexts from the core language.

$$\textit{Evaluation contexts} \quad E \quad ::= \quad \ldots \mid \mathbf{reset}_X\ E$$

Because each **shift** is handled by the nearest **reset** (and the typing rules ensure that these are labeled by the same name), we need to identify within each evaluation context E that **reset** (if any) which is closest to the hole of E. Thus, we identify the specific subclass of evaluation contexts which are pure, in the sense that they do not contain any resets.

Definition 31 (Pure evaluation contexts)
An evaluation context E is pure if it does not contain any **reset** *constructors acting on the hole of the context. In other words, the syntactic category of pure evaluation contexts is defined as*

$$\textit{Pure contexts} \quad P \quad ::= \quad [\,] \mid P\ e_1 \mid v_1\ P \mid \textbf{let box}\ u = P\ \textbf{in}\ e \mid \textbf{choose}\ P$$

This is exactly the notion of evaluation contexts from the core language.

Lemma 32 (Evaluation context decomposition)
If E is an evaluation context, then either:

1. *E is a pure context, or*
2. *there exist unique evaluation context E' and pure context P' such that $E = E'[\textbf{reset}_X\ P']$.*

The syntactic category of redexes is extended with the new cases involving **shift** and **reset**, and we define the primitive reduction relation for the new redexes.

$$\textit{Redexes} \quad r \quad ::= \quad \ldots \mid \textbf{reset}_X\ v \mid \textbf{reset}_X\ P[\textbf{shift}_Y\ k.\ e]$$

$$\frac{}{\Sigma, \textbf{reset}_X\ v \longrightarrow \Sigma, v}$$

$$\frac{}{\Sigma, (\textbf{reset}_X\ P[\textbf{shift}_X\ k.\ e]) \longrightarrow \Sigma, [(\lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ P[u])/k]e}$$

The first reduction rule is simple; it just serves to let the values pass through a **reset**. Indeed, values are effect free (in this case, shift-free), so no resets are really relevant for them. The second primitive reduction deserves more comment. It proscribes that the evaluation context P be captured and substituted for a continuation variable k in e. But notice that the reset which delimits the context P is *discarded* altogether from the further evaluation. It does not survive the capturing as part of the reduct, and it does not survive in the captured continuation either (which only includes P). We have already pointed this fact out in our discussion of the type system, and this reduction rule makes it formal.

There are merits to both of these choices. The more obvious one is that because the **reset** is removed from the reduct, we can shift to names which are further down in the support stack.[2]

It is a bit more difficult to explain the benefits of the decision that the delimiting **reset** be discarded from the captured continuation as well. From a purely technical aspect, it certainly seems more flexible to discard the **reset** than to include it in the continuation. After all, if the **reset** is missing, we can always put it back; if it is retained, we can never eliminate it. But the important point is that it also improves the expressiveness of the calculus. A particular example we have in mind concerns the application of composable continuations to neatly encode *bounded nondeterminism* [DF89, DF90]. What we would be able to do in our system, thanks to this particular design decision, is to express in the same manner the *unbounded nondeterminism* as well. We illustrate this argument in the following two examples.

Example 13 [Bounded nondeterminism] Composable continuations have been used to conveniently express "nondeterministic computation"; that is, computation which can return many results [DF89, DF90]. We paraphrase from these papers the following program for finding all the triples (i, j, k) of distinct positive integers smaller than n that sum up to s, which is very effectively phrased in terms of a primitive function `choice`.

[2]This idea is not new; it has already been noticed in Appendix C of [DF89], but it was not further explored.

```
fun choice n =
     shift c.
        let fun loop (s:int) : int =
            if s = 0 then shift c. ()
            else (c s; loop (s - 1))
        in
            loop n
        end
fun triples (n, s) =
    let val i = choice n
        val j = choice (i - 1)
        val k = choice (j - 1)
    in
        if (i + j + k = s) then (i, j, k)
        else shift c. ()
    end
- reset (print triples (9, 15));
val it =
(9, 5, 1)
(9, 4, 2)
(8, 6, 1)
(8, 5, 2)
(8, 4, 3)
(7, 6, 2)
(7, 5, 3)
(6, 5, 4)
```

The program works by maintaining a list of composable continuations, which is represented at run-time as a sequentially composed sequence of expressions of unit type. Each call to `choice` picks the *top element* of the list and stores this top element into the composable continuation `c`, while at the same time removing it from the list. This element is then expanded into `(c n; c (n - 1); ... c (1); ())` which is placed back at the top of the list. In this sense, the strategy that the above program uses for exploring the search space is *depth-first*. The depth of the search tree must be *bounded* (and it is in this particular example) if the program is to *enumerate* all the solutions. This is the relation of composable continuations in the style of [DF89, DF90] to bounded non-determinism.

In our calculus, we can write a very similar program that employs the depth-first-search strategy, except that now we need a name to represent destination points associated to **reset**. We use one name X of type `unit` to mark the beginning of each of the continuations on the stack.

```
fun choice (n : int) : □_X int =
    box (shift_X c:□_X int -> □_X unit.
       let fun loop (s : int) : unit =
            if s = 0 then ()
            else
              let box u = c (box s)
              in
                  (reset_X u); loop (s - 1)
              end
       in
          loop (n)
       end)
fun triple (n : int, s : int) : unit =
reset_X
   let val i = (let box u = choice n
                in u end)
       val j = (let box u = choice (i - 1)
                in u end)
       val k = (let box u = choice (j - 1)
                in u end)
   in
       if (i + j + k = s) then
         print (i, j, k)
       else ()
   end
```

■

Example 14 [Unbounded nondeterminism] In our calculus of composable continuations we are not limited to the depth-first-search strategy. We can, for example, employ *breadth-first-search* to enumerate the solutions even if the search space is of unbounded depth. As outlined before, our continuations capture the environment up to, but not including the resets, and this provides the needed expressiveness. In particular, when expanding the top element from the list of computations, we do not need to push the newly expanded cases on the top of the list (as in the previous example); we can place them at the bottom. How does this work? First, we capture the whole list of computations into a continuation function of its own. The argument of this function abstracts the top of the list, and therefore, adding new elements at the top of the list represented by the continuation (i.e. doing depth-first-search) is not influenced by whether the

continuation is delimited by a **reset** of not. But, the continuations must not be delimited if elements are to be added at the bottom, as otherwise the delimiter gets in the way. Because we do not capture the delimiting **reset** into the continuation, we can first extend the represented list at the bottom (thus encoding breadth-first-search strategy), and only then place a **reset** back, to delimit the new list. The scope of this **reset** will then include the newly expanded search cases.

To paraphrase, even though our continuations are not delimited at the time of capturing, the type system will force us to delimit them before they are invoked, but not before we had a chance to modify them first to suit our needs.

As an example, we present a version of `triple` that uses breadth-first-search strategy. We now need two names to delimit the continuations: the name X:*unit* which delimits individual entries in the list, and the name Y:*unit* for the whole list. The new function `bfs-choice` picks the top element `c1` expands it into (and we are being imprecise about the types here): `(reset`$_X$ `c1(n); reset`$_X$ `c1(n - 1);...; reset`$_X$ `c1(1); reset`$_X$ `())`. Then this whole expanded segment is attached to the bottom of the list, which itself has been captured into the continuation `c2`.

```
fun bfs-choice (n : int) : □Y,Xint =
    box (shiftX c1:□Y,Xint -> □Y,Xunit.
          shiftY c2:□Yunit -> □Yunit.
           resetY
            let fun loop (s : int) : unit =
                if s = 0 then ()
                else
                  let box u = c1 (box s)
                  in
                      (resetX u); loop (s - 1)
                  end
                box v = c2 (box resetX ())
            in
                v; loop (n)
            end)
fun triple (n : int, s : int) : unit =
resetY
 resetX
   let val i = (let box u =
                  bfs-choice n
                in u end)
       val j = (let box u =
                  bfs-choice (i - 1)
                in u end)
       val k = (let box u =
                  bfs-choice (j - 1)
                in u end)
   in
       if (i + j + k = s) then
         print (i, j, k)
       else ()
   end
```

■

3.3.3 Structural properties and type soundness

Lemma 33 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$ *and* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Lemma 34 (Replacement)

1. *If* $\Sigma; \Delta \vdash P[e] : A\,[C]$, *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* $\Sigma'; \Delta' \vdash P[e'] : A\,[C]$
2. *if* $\Sigma; \Delta; \Gamma \vdash E[e] : A\,[C]$, *then there exist a type* B *and a support* D *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* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

Lemma 35 (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* $A = A_1 \twoheadrightarrow A_2$, *then* $v = \nu X{:}A_1.\ e$ *and* $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$

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

Lemma 36 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longrightarrow \Sigma', e'$, *then* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 37 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 38 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \longrightarrow \Sigma', e'$.

Lemma 39 (Unique decomposition)
For every expression e, *either:*

1. e *is a value, or*
2. $e = P[\mathbf{shift}_X\ k.\ e']$, *for a unique pure context* P, *or*
3. $e = E[r]$ *for a unique evaluation context* E *and a redex* r.

Lemma 40 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, *then either*

1. e *is a value, or*
2. *there exists a term* e' *and a context* Σ', *such that* $\Sigma, e \longmapsto \Sigma', e'$.

Lemma 41 (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 the domain of* Σ, *such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $e_2 = \pi(e_1)$.

3.4 Calculus for dynamic binding

3.4.1 Syntax and typing

The type system that we develop in this section is intended to model memory allocation, lookup and *non-destructive* update. The idea is to view names as memory locations of arbitrary type, and track their dereferencing through the mechanism of supports. Looking up a name will be an effect, and substituting a name with a term by means of an *explicit substitution* will be its handler. The operational semantics evaluates expressions with empty support, and hence will permit dereferencing of only those names that are in the scope of a substitution which defines them. Thus, we can only dereference *initialized* names.

In a sense, this system is a middle way between a calculus with local variables and let-definitions on one side, and a calculus of state on the other side. Names are really allocated memory locations, but at the same time, assigning definitions to names via explicit substitutions is not a destructive operation. Each name can be defined arbitrary number of times (including zero), but the definitions only have local scope, and dereferencing a name will use the nearest definition. Thus, the obtained calculus is really a type-safe version of *dynamic binding*, much in the style of LISP and Scheme. We will build on this system in Section 4 to obtain a full-fledged calculus of state with destructive update.

The syntax of the calculus for dynamic binding extends the core fragment with new constructs for name lookup and substitution. The resulting language is very similar to the one we presented in [Nan02].

$$\begin{array}{lrcl} \textit{Explicit substitutions} & \Theta & ::= & \cdot \mid X \to e, \Theta \\ \textit{Terms} & e & ::= & \ldots \mid X \mid \langle \Theta \rangle e \end{array}$$

An explicit substitution Θ is syntactically defined as a list of assignments of expressions to names. A name X is referenced by simply using it in some term. The construct $\langle \Theta \rangle e$ denotes an application of an explicit substitution Θ over the expression e.

Example 15 Assuming that X and Y are integer names, the following code segment defines a polynomial in X and Y, then instantiates X and Y to 1 and 2, respectively, before evaluating the polynomial.

```
- let box u = box (X^2 + Y^2)
  in
      <X->1, Y->2> (u + 2XY)
  end;

val it = 9 :  int
```

■

The explicit substitutions in our calculus 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

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

Just like in the case of exception handlers, we treat the substitutions 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 $\mathbf{range}(\Theta)$.

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

Definition 42 (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.

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

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 15. 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. On the other hand, values are not of a particular interest in this definition, because they are effect-free (in this case, name-free), so substitution application does not descend into λ- and ν-abstractions.

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

Definition 43 (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 $\mathbf{range}(\Theta_2)$ proceeds recursively on terms of decreasing size.

The type system of calculus consists of two mutually recursive judgments: one for typing expressions, and one for typing explicit substitutions. The judgment for expressions has the form

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

and is a simple extension of the expression judgment from the core fragment. The expression e will have support C only if all the names dereferenced in e (and therefore, appearing in *unsuspended* positions in e) are listed in C. The support C is simply a set, just like in the calculi for exceptions and catch-and-throw.

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 terms with support D. Therefore, one and the same substitution can be given many different types, depending at which supports C and D 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, Θ does map a term of support $[\,]$ into another term with support $[\,]$, a term of support $[X]$ into a term with support $[\,]$, and a term with support $[X, Y, Z]$ into a term with support $[Z]$. We present the typing rules of the calculus in Figure 8, and comment on them briefly.

The identity substitution $\langle\,\rangle$ does not provide a definition for any names (or, rather assigns every name to itself). Thus, if applied to an expression of support C it will result with 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 $D \sqsupseteq C$. Composite substitutions are typechecked by recursively typechecking all of their assignments.

In line with the described semantics of the typing judgment for expressions, 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 16 Consider the ML-like program below.

Explicit substitutions

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

Name dereference and substitution

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

Figure 8: Typing rules for the calculus of dynamic binding.

```
let val xref = ref 0
    fun f (y) = !xref + 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>`. ■

3.4.2 Operational semantics

The evaluation judgment for the calculus for dynamic binding extends the core fragment with the new construct for substitution application. The judgment still has the form

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

where Σ and Σ' are run-time contexts of currently allocated, but not necessarily initialized, names. However, just like in the previous systems, we only consider for evaluation the expressions e which have empty support,

i.e., expressions whose names are dereferenced only in dead-code subterms, or in the scope of some initializing substitution.

We adopt a call-by-value strategy for evaluating substitutions; that is, all the assignments in a substitutions are first reduced to values, before the substitution itself is applied. To formalize this policy, we define the notion of value substitutions, and use it to extend the evaluation contexts.

$$\begin{array}{llll} \textit{Value substitutions} & \sigma & ::= & \cdot \mid X \to v, \sigma \\ \textit{Evaluation contexts} & E & ::= & \ldots \mid \langle \sigma, X \to E, \Theta \rangle e \end{array}$$

The category of redexes and the primitive reduction relation are extended to admit substitution applications.

$$\textit{Redexes} \quad r \quad ::= \quad \ldots \mid \langle \sigma \rangle e$$

$$\frac{}{\Sigma, \langle \sigma \rangle e \longrightarrow \Sigma, \{\sigma\} e}$$

Notice 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.

3.4.3 Structural properties and type soundness

Lemma 44 (Explicit substition 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 $\Sigma; \Delta \vdash \langle \Theta \circ \Theta' \rangle : [C_1] \Rightarrow [D]$*

Lemma 45 (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 $\Sigma; \Delta \vdash \langle [e_1/u]\Theta \rangle : [D'] \Rightarrow [D]$*

Lemma 46 (Replacement)
If $\Sigma; \Delta \vdash E[e] : A\,[C]$, then there exist a type B such that

1. *$\Sigma; \Delta \vdash e : B\,[C]$, and*
2. *if Σ', Δ' extend Σ, Δ, and $\Sigma'; \Delta' \vdash e' : B\,[C]$, then $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$*

Lemma 47 (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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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

Lemma 48 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Lemma 49 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$*, then* Σ' *extends* Σ*, and* $\Sigma'; \cdot \vdash e' : A\,[C]$*.*

Lemma 50 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$*, then there exists a term* e' *and a context* Σ'*, such that* $\Sigma, r \longrightarrow \Sigma', e'$*.*

Lemma 51 (Unique decomposition)
For every expression e*, either:*

1. e *is a value, or*
2. $e = E[X]$*, for a unique evaluation context* E *and a name* X*, or*
3. $e = E[r]$ *for a unique evaluation context* E *and a redex* r*.*

Lemma 52 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$*, then either*

1. e *is a value, or*
2. *there exists a term* e' *and a context* Σ'*, such that* $\Sigma, e \longmapsto \Sigma', e'$*.*

Lemma 53 (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 the domain of* Σ*, such that* $\Sigma_2 = \pi(\Sigma_1)$ *and* $e_2 = \pi(e_1)$*.*

4 Nominal possibility

4.1 Calculus for state

In the previous sections, we have demonstrated how the modal operator of necessity can be used to demarcate handleable effects. For example, in the calculus of dynamic binding from Section 3.4, we 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 3.4 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*.

As we already mentioned in the introduction, in the modal decomposition of monads from [PD01], the single-threading and scope extension are exactly the duties of the $\Diamond$ modality of 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. This would pleasantly contrast the type $\Box_C A$ that we already used in Section 3.4 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 3.4.

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

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 17 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 cut down on verbosity, and simply wrote the closure 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 3.4, 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 9, 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 3.4. This is only to be expected since, after all, we introduced closures with the intention to single-thread substitution applications. The

Closures

$$\frac{\Sigma;\Delta \vdash e : A\,[D] \qquad \Sigma;\Delta \vdash \langle\Theta\rangle : [D] \Rightarrow [C]}{\Sigma;\Delta \vdash [\Theta, e] \div_D A\,[C]} \qquad \frac{\Sigma;\Delta \vdash e : \Diamond_{D_1} A\,[C] \qquad \Sigma;(\Delta, x{:}A) \vdash f \div_{D_2} B\,[D_1]}{\Sigma;\Delta \vdash \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f \div_{D_2} B\,[C]}$$

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

Possibility

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

Figure 9: Typing rules for nominal possibility.

two constructs, however, will have different 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[3] 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.

The closure construct **let box** $u = e$ **in** f is a syntactic sugar for **let dia** $y = (\mathbf{let\ box}\ u = e\ \mathbf{in\ dia}\ f)$ **in** $[y]$, and its typing rule is accordingly derived. The construct is introduced into the calculus in order to maintain the subformula property [PD01].

Example 18 We will use some further syntactic abbreviations as well: **let val** $x = e$ **in** f is short for

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

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

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

[3]Although it can be viewed as an elimination for $\Diamond$ as well.

In contrast to the **let box** rule, however, these two constructs do not have any proof-theoretic significance. The typing rules for the two are easily derived as

$$\frac{\Sigma;\Delta \vdash e : A\,[C] \qquad \Sigma;(\Delta, x{:}A) \vdash f \div_D B\,[C]}{\Sigma;\Delta \vdash \mathbf{let\ val}\ x = e\ \mathbf{in}\ f \div_D B\,[C]} \qquad \frac{(\Sigma, X{:}A);\Delta \vdash f \div_D B\,[C] \qquad X \notin \mathbf{fn}(B,C,D)}{\Sigma;\Delta \vdash \mathbf{new}\ X{:}A\ \mathbf{in}\ f \div_D B\,[C]}$$

■

Example 19 Assume that C_1, C_2 and D are support sets, such that $C_1 \sqsubseteq C_2$. Then the following terms are well-typed of empty support.

$$\begin{array}{l}
\lambda x{:}\Diamond_{C_2} A.\ \mathbf{dia}\ (\mathbf{let\ dia}\ y = x\ \mathbf{in}\ [y]) : \Diamond_{C_2} A \to \Diamond_{C_1} A \\
\lambda x{:}A.\ \mathbf{dia}\ [x] : A \to \Diamond A \\
\lambda x{:}\Diamond_D \Diamond_{C_2} A.\ \mathbf{dia}\ (\mathbf{let\ dia}\ y = x\ \mathbf{in\ let\ dia}\ z = y\ \mathbf{in}\ [z]) : \Diamond_D \Diamond_{C_2} A \to \Diamond_{C_2} A \\
\lambda e_1{:}\Box_{C_1}(A \to B).\ \lambda e_2{:}\Diamond_{C_2} A.\ \mathbf{dia}\ (\mathbf{let\ box}\ u = e_1\ \mathbf{in\ let\ dia}\ x = e_2\ \mathbf{in}\ [u\ x]) \\
\qquad : \Box_{C_1}(A \to B) \to \Diamond_{C_2} A \to \Diamond_{C_2} B
\end{array}$$

The first function 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 S4 modal possibility [PD01], which can be recovered if the involved supports are set to be empty[4]. For example, the second term 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 third term illustrates that it is only the last layer of $\Diamond$'s that matter; all the additional ones can be ignored. The fourth term takes a function e_1 which needs names C_1 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 $C_2 \sqsupseteq C_1$). 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 3.4, 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 garbage-collected. In that sense, the identity substitution with type $[\,] \Rightarrow [\,]$ is 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.

The most important consequence of substitution typing concerns operational semantics of substitution composition, on which we remained uncommitted in Section 3.4, but have to comment on here, because substitutions are now promoted into global store and garbage-collection becomes an issue. Obviously, a composition of two substitution can be given different syntactic representations, depending on the type at which it is performed. Consider for example the composition $\langle X \to 1\rangle \circ \langle\,\rangle$. If the identity substitution is typed as $[\,] \Rightarrow [\,]$, then the result may be represented simply as $\langle\,\rangle$. If the identity substitution is typed as $[X] \Rightarrow [X]$, then the result is represented as $\langle X \to 1\rangle$.

In the current system, the type of a substitution cannot be determined from its syntax, so it cannot be decided at run-time (when the types are erased) which locations should be garbage-collected, and which should not. This can be remedied, for example, by storing the domain type of a substitution as part of its syntax, but we do not do that here for simplicity. Rather, we avoid tying garbage collection to the type

[4]But not quite, as there is a twist in the rule for closure introduction

system at all, and consider that substitution compositions are performed so that the result always preserves all the assignments.

Example 20 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.

```
new X:int in
    let 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
```

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 garbage-collected. ■

4.2 Operational semantics

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 **new** as they are only syntactic sugar and do 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

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

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 3.4.

The judgment for evaluating closures is the default judgment of the operational semantics, because it proscribes 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 proscribed 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 3.4 as summarized below.

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

The two evaluation judgments require two primitive reduction relations. We define them in Figure 10. All the rules are fairly straightforward, except the one for primitive reduction of closures with nonempty substitution. The meaning of this rule 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.

4.3 Structural properties and type soundness

Lemma 54 (Support weakening)

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

Lemma 55 (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* $\Sigma; \Delta \vdash [e_1/u]f \div_{C_1} B\,[D]$

Lemma 56 (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* $\Sigma; \Delta \vdash \{\Theta\}f \div_{C_1} A\,[D]$

Lemma 57 (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* $A = A_1 \twoheadrightarrow A_2$, *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* $\Sigma; \cdot \vdash f \div_D B\,[C]$

Primitive reductions for expressions

$$\frac{}{\Sigma, (\lambda x.\ e)\ v \xrightarrow{\sigma} \Sigma, [v/x]e} \qquad \frac{}{\Sigma, \textbf{let box } u = \textbf{box } e_1 \textbf{ in } e_2 \xrightarrow{\sigma} \Sigma, [e_1/u]e_2}$$

$$\frac{}{\Sigma, \textbf{choose } (\nu X{:}A.\ e) \xrightarrow{\sigma} (\Sigma, X{:}A), e} \qquad \frac{}{\Sigma, \langle\sigma'\rangle e \xrightarrow{\sigma} \Sigma, \{\sigma'\}e} \qquad \frac{}{\Sigma, X \xrightarrow{\sigma} \Sigma, \sigma(X)}$$

Primitive reductions for closures

$$\frac{\sigma' \neq (\cdot)}{(\Sigma, \sigma), \textbf{let dia } x = \textbf{dia } [\sigma', e] \textbf{ in } f \longrightarrow (\Sigma, \sigma \circ \sigma'), \textbf{let dia } x = \textbf{dia } [e] \textbf{ in } f}$$

$$\frac{}{(\Sigma, \sigma), \textbf{let dia } x = \textbf{dia } [v] \textbf{ in } f \longrightarrow (\Sigma, \sigma), [v/x]f}$$

Evaluation for expressions

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

Evaluation for closures

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

Figure 10: Operational semantics for the modal calculus with possilibity.

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

Lemma 58 (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 $\Sigma'; \Delta' \vdash F[f'] \div_C A\,[D_1]$*

Lemma 59 (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 $C' \subseteq \mathbf{dom}(\Sigma')$*

Lemma 60 (Preservation)
Let $\Sigma; \cdot; \vdash \langle \sigma \rangle : [C] \Rightarrow [\,]$. Then the following holds:

1. *if $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \overset{\sigma}{\longmapsto} \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$*
2. *if $\Sigma; \cdot \vdash f \div_D A\,[C]$ and $(\Sigma, \sigma), f \longmapsto (\Sigma', \sigma'), f'$, 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 $C' \subseteq \mathbf{dom}(\Sigma')$*

Lemma 61 (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 $(\Sigma, \sigma), c \longrightarrow (\Sigma', \sigma'), f'$.*

Lemma 62 (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.*

Lemma 63 (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* $\Sigma, e \stackrel{\sigma}{\longmapsto} \Sigma', e'$.
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* $(\Sigma, \sigma), f \longmapsto (\Sigma', \sigma'), f'$

Lemma 64 (Determinacy)
1. *If* e, e_1, e_2 *are terms such that* $\Sigma, e \stackrel{\sigma}{\longrightarrow}^n \Sigma_1, e_1$ *and* $\Sigma, e \stackrel{\sigma}{\longrightarrow}^n \Sigma_2, e_2$, *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* $f_2 = \pi(f_1)$.

5 Related work

Integrating effects into functional calculi has quite a long history, and this section is bound to be very incomplete. Numerous systems have been proposed, treating various effects and with various levels of precision and verbosity of typing. As an example, we only list [GL86, LG88, JG89, JG91, TJ92, TJ94, NNA97, TT97, BT01, Thi03].

A treatment of exceptions in a lazy language is given by Peyton Jones at al. in [PRH+99]. This paper associates exceptions with values, so that a value of any type is either "normal" or "exceptional". This is to some extent similar with our calculus where values and expressions, rather than function calls, are marked by the type system as exceptional. And our exceptions share the universe of types with the other values, too. Another exception calculi is presented by de Groote in [dG95]. It is a call-by-value calculus which uses separate binding mechanisms to introduce exceptions into the computation. However, because of the lack of modal or monadic types, it has to specifically require that values of the language are effect-free, in which case it implements the Standard ML exception mechanism. This paper also discuses the logical content of exceptions, and relationship with classical logic. Exception mechanism of Java relates to our calculus as well, as Java methods must be labeled by the exceptions they can raise [GJS97]. Catch-and-throw calculus is a specific simplifications of exceptions, and theoretical analysis of catch and throw can be found in [Nak92, Kam00a, KS02].

Composable continuations were probably first considered by Felleisen in [Fel88], in an untyped setting and with shifting only to the nearest reset (or prompt). A generalization to a whole family of control operators for shifting, each of which is indexed by a numeral proscribing how many closest resets should be jumped over, appeared in [SF90a]. Also in untyped setting, Hieb, Dybvig and Anderson in [HDA94] introduce labels insted of numerals to describe the destination points for a hierarchy of shifts. Danvy and Filinski in [DF89] develop a type system for composable continuations with a single shift operator. The resets are not labeled. In the Appendix C, they also briefly discuss the idea which we have employed here: upon capturing, remove the resets from the environment, so that jumps can be made to the resets further down in the context stack. Logical content of composable continuations is studied by Murthy in [Mur92]. This paper develops a type system for composable continuation with a hierarchy of shifting operators, which is based on monads indexed by sets of types, but has to restrict the resets to only implication-free types in order to preserve soundness. Wadler in [Wad94] further analyses the above type systems for composable continuations with a single shift

operator, and with a hierarchy of shift operators, and presents them in terms of indexed monads. Most recently, Kameyama in [Kam00a, Kam00b] works with labels instead of numerals to provide a hierarchy of shifting operators. These calculi lack modal or monadic types and must (like the abovementioned system of exceptions [dG95]) limit the form of values in order to avoid scope extrusion for labels and preserve soundness.

Dynamic binding has been inadvertantly introduced in the first versions of LISP, but then became a feature, rather than a bug, in the subsequent implementations and dialects. For semantical treatment of dynamic binding we refer to [Mor97]. A typed calculus for dynamic binding, called λN, is developed in [Dam96, Dam98]. 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. However, the type system does not have a notion of support, so it cannot prevent reading from uninitialized dynamic variables.

Coming from the side of logic and type theory, effect calculi are directly representable by monads. Monads were invented for use in denotational semantics by Moggi [Mog89, Mog91], and were adopted for functional programming by Wadler [Wad92, Wad95, Wad98] to represent effectful computations. We further point to the work of Filinski [Fil94, Fil96, Fil99] which develops the concepts of monadic reflection and reification, with intentions similar to ours: to increase the flexibility of monadic programming and introduce the notion of handling in the framework. Similar motivation lies behind Kieburtz's introduction of lexical scoping for effects in [Kie98].

On the other hand, our paper was specifically motivated by the work of Pfenning and Davies [PD01] which formulates natural deduction for S4 modal logic and the $\lambda^{\Box}$-calculus as proof terms for this logic. It also shows how a monadic system can be embedded using modalities. The necessitation segment $\lambda^{\Box}$ of the modal calculus has been used for purposes of staged computation [DP96, WLPD98], and run-time code-generation [LL96, WLP98]. The modal necessity, as defined in these papers, is a comonad, and the fact that comonads may represent effects have already been noticed by Kieburtz [Kie99] and Pardo [Par00], but it was not related to the notion of handling.

That modal necessity can be very naturally extended with the notion of names was argued in [Nan02]. The calculus from that paper is a direct precursor to the effect system we presented here. It is based on the work on Nominal Logic and FreshML by Pitts and Gabbay[GP02, PG00, Pit01, Gab00] which introduce the concept of names as urelements of Fraenkel-Mostowski set theory.

6 Conclusions and future work

This paper introduced a logically motivated effect system, based on modal logic S4, which is designed to naturally support the process of effect handling. It is founded on the observation by Pfenning and Davies in [PD01] that a monad can be decomposed in terms of S4 modal operators for necessity and possibility. In our calculus, the necessity operator $\Box$ serves to mark impure program segments and accounts for a hygenic propagation of effects; it is used to represent handleable effects. The possibility operator $\Diamond$ is used to represent effects which have global scope; that is, effects which are not handleable.

The presented system extends the modal calculus from [PD01] with names, which are labels that can be dynamically introduced into the computation and serve to identify effects. The (possibly oredered) collection of effects that may appear in a certain term is refered to as support of the term. The typing judgment relates terms to their supports, and only terms of empty support (which are, hence, guranteed to be effect-free) will be considered for evaluation. The notion of support in the calculus is rather flexible; depending on the particular effect being model, different definitions of support may be used. For example, in case of exceptions, catch-and-throw and state, the supports are simply sets of names listing the exceptions, jump destinations, or locations read by the term, respectivelly. In case of composable continuations, supports are stacks of names depicting the environments of nested continuation-delimiting points. It is an interesting future work to investigate how different definitions of support may interact in a system combining several effects that are heterogenious in this respect.

In our system, the modalities are indexed by supports. For example, the type $\Box_C A$ clasifies computations

of type A capable of raising effects whose names are listed in the support C. It is possible to project a value of type A out of a computation of type $\Box_C A$ if the effects in C can be handled (in a way specified by the particular effects in question). In the special case when C is the empty support, the computation is effect-free, and the value can be projected out trivially; this is the behaviour of the propositional S4 necessity, recovered. We further impose monotonicity on the new logic with multiple modalities, resulting in the identification of the types A and $\Box A$, where the later type has empty index support. Operationally, this allow us to coerce values into effectful computations, much like a monadic system would do.

The presented system consists of a core effect calculus which can easily be extended to account for various effects, simply by providing the introduction and elimination (i.e., handling) forms and specifying the particular notions of support suitable for that effect. We demonstrate this by extending the core fragment into new calculi for exceptions, catch-and-throw, composable continuations, dynamic binding and state. We should mention, however, that the obtained systems are probably not very practical (and certainly not in cases of exceptions and state) as the verbosity of support annotations on types may significantly complicate any serious programming effort. It may be of some comfort that a similar criticism is in fact applicable to most other effect calculi as well. At any rate, it is a very important future work to investigate the questions of type and support inference in this calculus (probably in the style of [TJ92, BT01]), and develop new abstraction mechanism which can hide the unwanted support and result in a more practical language (perhaps combining support polymorphism [Nan02] and proof irrelevance [Pfe01, AB01]). The system presented here is logically motivated and fully explicit about the supports of its terms, and is therefore a solid theoretical basis for such investigations.

References

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

[BT01] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001.

[Dam96] Laurent Dami. Functional programming with dynamic binding. In Dennis Tsichritzis, editor, *Object Applications*, pages 155–172. Technical Report, University of Geneva, 1996.

[Dam98] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1998.

[DF89] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU - Computer Science Department, University of Copenhagen, 1989.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Conference on LISP and Functional Programming*, pages 151–160, Nice, France, 1990.

[dG95] Philippe de Groote. A simple calculus of exception handling. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1995.

[DHM91] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Symposium on Principles of Programming Languages, POPL'91*, pages 163–173, Orlando, Florida, 1991.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Symposium on Principles of Programming Languages, POPL'96*, pages 258–270, St. Petersburg Beach, Florida, 1996.

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

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages, POPL'88*, pages 180–190, San Diego, California, 1988.

[FFKD86] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. Reasoning with continuations. In *Symposium on Logic in Computer Science, LICS'86*, pages 131–141, Cambridge, Massachusetts, 1986.

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, Copenhagen, Denmark, 1989. DIKU Report 89/11.

[Fil94] Andrzej Filinski. Representing monads. In *Symposium on Principles of Programming Languages, POPL'94*, pages 446–457, Portland, Oregon, 1994.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.

[Fil99] Andrzej Filinski. Representing layered monads. In *Symposium on Principles of Programming Languages, POPL'99*, pages 175–188, San Antonio, Texas, 1999.

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

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Conference on LISP and Functional Programming*, pages 28–38, Cambridge, Massachusetts, 1986.

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

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Symposium on Principles of Programming Languages, POPL'90*, pages 47–58, San Francisco, California, 1990.

[HDA94] Robert Hieb, Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[JG89] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Conference on Programming Language Design and Implementation, PLDI'89*, pages 218–226, Portland, Oregon, 1989.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Symposium on Principles of Programming Languages, POPL'91*, pages 303–310, Orlando, Florida, 1991.

[Kam00a] Yukiyoshi Kameyama. Towards logical understanding of delimited continuations. In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations, CW'01*, pages 27–33, 2000. Technical Report No. 545, Computer Science Department, Indiana University.

[Kam00b] Yukiyoshi Kameyama. A type-theoretic study on partial continuations. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2000.

[Kie98] Richard B. Kieburtz. Taming effects with monadic typing. In *International Conference on Functional Programming, ICFP'98*, pages 51–62, Baltimore, Maryland, 1998.

[Kie99] Richard B. Kieburtz. Codata and comonads in Haskell. Unpublished. Available from `http://www.cse.ogi.edu/~dick`, 1999.

[KS02] Yukiyoshi Kameyama and Masahiko Sato. Strong normalizability of the non-deterministic catch/throw calculi. *Theoretical Computer Science*, 272(1–2):223–245, 2002.

[Lan65] Peter J. Landin. A correspondence between ALGOL-60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 1965.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Symposium on Principles of Programming Languages, POPL'88*, pages 47–57, San Diego, California, 1988.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Conference on Programming Language Design and Implementation, PLDI'96*, pages 137–148, 1996.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, LICS'89*, pages 14–23, Asilomar, California, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mor97] Luc Moreau. A syntactic theory of dynamic binding. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 727–741. Springer, 1997.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mur92] Chetan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations, CW'92*, pages 49–71, 1992. Technical Report STAN-CS-92-1426, Stanford University.

[Nak92] Hiroshi Nakano. A constructive formalization of the catch and throw mechanism. In *Symposium on Logic in Computer Science, LICS'92*, pages 82–89, Santa Cruz, California, 1992.

[Nan02] Aleksandar 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.

[NNA97] Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: the static semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 141–171. Springer, 1997.

[Par00] Alberto Pardo. Towards merging recursion and comonads. In *Proceedings of the 2nd Workshop on Generic Programming, WGP'00*, pages 50–68, Ponte de Lima, Portugal, 2000.

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

[Pfe01] Frank 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.

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

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

[PRH+99] Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Conference on Programming Language Design and Implementation, PLDI'99*, pages 25–36, Atlanta, Georgia, 1999.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *25th National ACM Conference*, pages 717–740, Boston, Massachusetts, 1972.

[SF90a] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[SF90b] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Conference on LISP and Functional Programming*, pages 161–175, Nice, France, 1990.

[SW74] Christopher Strachey and Christopher Wadsworth. A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, 1974.

[Thi97] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.

[Thi03] Hayo Thielecke. From control effects to typed continuation passing. In *Symposium on Principles of Programming Languages, POPL'03*, pages 139–149, New Orleans, Louisiana, 2003.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Wad92] Philip Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages, POPL'92*, pages 1–14, Albequerque, New Mexico, 1992.

[Wad94] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[Wad98] Philip Wadler. The marriage of effects and monads. In *International Conference on Functional Programming, ICFP'98*, pages 63–74, Baltimore, Maryland, 1998.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *Conference on Programming Language Design and Implementation, PLDI'98*, pages 224–235, Montreal, Canada, 1998.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), 1998.

7 Proofs for exceptions

Lemma 65 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$*, 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* $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : B[D'] \Rightarrow B[D]$*, then* $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : B[D'] \Rightarrow B[D]$

Proof:

1. By induction on the derivation of e_2.

   case $e_2 = x, v$ obvious.

   case $e_2 = u$ follows by support weakening.

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

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

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

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

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

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

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

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

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

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

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

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

   case $e_2 = \mathbf{raise}_X\ e'$, where $X{:}B' \in \Sigma$, and $X \in D$.

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

   case $e_2 = e'\ \mathbf{handle}\ \Theta$.

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

(b) by first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D']$
(c) by second induction hypothesis, $\Sigma; \Delta \vdash \langle [e_1/u]\Theta \rangle : B[D'] \Rightarrow B[D]$
(d) result follows by typing rule for handle

2. case $\Theta = (\cdot)$. Obvious.
   case $\Theta = (Xz \to e, \Theta')$, where $X{:}B' \in \Sigma$.
   (a) by derivation, $\Sigma; (\Delta, u{:}A[C], z{:}B') \vdash e : B\,[D]$, and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle \Theta' \rangle : B[D' \setminus X] \Rightarrow B[D]$
   (b) by the first induction hypothesis, $\Sigma; (\Delta, z{:}B') \vdash [e_1/u]e : B\,[D]$
   (c) by the second induction hypothesis, $\Sigma; \Delta \vdash \langle [e_1/u]\Theta' \rangle : B[D' \setminus X] \Rightarrow B[D]$
   (d) result follows by the typing rule for composite handlers

■

Lemma 66 (Handler application)
Let $\Sigma; \Delta \vdash \langle \Theta \rangle : A[C] \Rightarrow A[D]$ *and* $\Sigma; \Delta \vdash e : B\,[D]$, *where* $X \in C$ *and* $X{:}B \in \Sigma$. *Then* $\Sigma; \Delta \vdash \Theta(X)(e) : A\,[D]$

Proof: By induction on the syntactic representation of Θ.

case $\Theta = (\cdot)$. Then $\Theta(X)(z) = \mathbf{raise}_X\ z$, and so $\Theta(X)(e) = \mathbf{raise}_X\ e$, and thus $\Sigma; \Delta; \Gamma \vdash \Theta(X)e : A\,[C]$, because $X \in C$.

case $\Theta = (Xz \to e_2, \Theta')$.

1. by derivation, $\Sigma; (\Delta, z{:}B) \vdash e_2 : A\,[D]$
2. by substitution principle, $\Sigma; \Delta \vdash [e/x]e_2 : A\,[D]$
3. but $\Theta(X)(e)$ is exactly $[e/x]e_2$, so the proof of the case is finished

case $\Theta = (Yz \to e_2, \Theta')$, and $X \neq Y$.

1. by derivation, $\Sigma; \Delta \vdash \langle \Theta' \rangle : A[C \setminus Y] \Rightarrow A[D]$
2. conslusion follows by induction hypothesis

■

Lemma 67 (Evaluation context decomposition)
If E *is an evaluation context, then either:*

1. E *is a pure context, or*
2. *there exist unique evaluation context* E' *and pure context* P' *such that* $E = E'[P'\ \mathbf{handle}\ \Theta]$

Proof: By induction on the structure of E.

case $E = []$ is obviously pure.

case $E = E_1\ e_2$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[P'\ \mathbf{handle}\ \Theta]$ in which case pick $E' = E_1'\ e_2$.

case $E = v_1\ E_2$.

By induction hypothesis, E_2 is either pure, in which case E is pure as well, or $E_2 = E_2'[\P'\ \mathbf{handle}\ \Theta]$ in which case pick $E' = v_1\ E_2'$.

case $E = \textbf{let box}\ u = E_1\ \textbf{in}\ e_2$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[P'\ \textbf{handle}\ \Theta]$ in which case pick $E' = \textbf{let box}\ u = E_1'\ \textbf{in}\ e_2$.

case $E = \textbf{choose}\ E_1$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[P'\ \textbf{handle}\ \Theta]$ in which case pick $E' = \textbf{choose}\ E_1'$.

case $E = \textbf{raise}_X\ E_1$.

By induction hypothesis, E_1 is either pure, in which case pick E is pure as well, or $E_1 = E_1'[P'\ \textbf{handle}\ \Theta]$ in which case pick $E' = \textbf{raise}_X\ E_1'$.

case $E = E_1\ \textbf{handle}\ \Theta$.

By induction hypothesis, E_1 is either pure, in which case pick $E' = []$ and $P' = E_1$, or $E_1 = E_2'[P_2'\ \textbf{handle}\ \Theta_2]$, in which case pick $E' = E_2'\ \textbf{handle}\ \Theta$ and $P' = P_2'$.

■

Lemma 68 (Replacement)

1. *If* $\Sigma; \Delta \vdash P[e] : A\,[C]$, *then there exist a type* B *and a support* $D \sqsubseteq C$ *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* $\Sigma; \Delta \vdash P[e'] : A\,[C]$
2. *If* $\Sigma; \Delta \vdash E[e] : A\,[C]$, *then there exist a type* B *and a support* D *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* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

Proof:

1. By induction on the structure of P.

   case $P = [\,]$. obvious

   case $P = P_1\ e_2$.
   (a) by derivation, $\Sigma; \Delta \vdash P_1[e] : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash e_2 : A_1\,[C]$
   (b) by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
   (c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : A_1 \to A\,[C]$
   (d) by weakening $\Sigma'; \Delta' \vdash e_2 : A_1\,[C]$
   (e) result follows by typing

   case $P = v_1\ P_2$.
   (a) by derivation, $\Sigma; \Delta \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash P_2[e] : A_1\,[C]$
   (b) by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
   (c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P_2[e'] : A_1\,[C]$
   (d) by weakening, $\Sigma'; \Delta' \vdash v_1 : A_1 \to A\,[C]$
   (e) result follows by typing

   case $P = \textbf{let box}\ u = P_1\ \textbf{in}\ e_2$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : \Box_{C_1} A_1\,[C]$, and $\Sigma; (\Delta, u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
(b) by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : \Box_{C_1} A_1\,[C]$
(d) by weakening, $\Sigma'; (\Delta', u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
(e) result follows by typing

case $P = \mathbf{choose}\ P_1$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : A_1 \twoheadrightarrow A\,[C]$
(b) by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : A_1 \twoheadrightarrow A\,[C]$
(d) result follows by typing

case $P = \mathbf{raise}_X\ P_1$, where $X{:}B' \in \Sigma$, and $X \in C$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : B'\,[C]$
(b) by induction hypothesis, there exist B and $D \sqsubseteq C$ such that $\Sigma; \Delta \vdash e : B\,[D]$
(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : B'\,[C]$
(d) result follows by typing

2. By Evaluation context decomposition, we only need to consider two cases.

case $E = P$. Follows from the first statement.

case $E = E_1[P\ \mathbf{handle}\ \Theta]$.

(a) by induction hypothesis, there exist B' and D' such that $\Sigma; \Delta \vdash P[e]\ \mathbf{handle}\ \Theta : B'\,[D']$
(b) by typing, $\Sigma; \Delta \vdash P[e] : B'\,[D'']$, and $\Sigma; \Delta \vdash \langle\Theta\rangle : B'[D''] \Rightarrow B'[D']$
(c) by first part of the lemma, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
(d) again by first part of the lemma, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash P[e'] : B'\,[D'']$
(e) result now follows by typing for handle

■

Lemma 69 (Canonical forms)
Let v be a closed value such that $\Sigma; \cdot; \cdot \vdash v : A\,[C]$. 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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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 70 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

case $e = (\lambda x.\ e')\ v$ follows by value substitution principle.

case $e = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$ follows by expression substitution principle.

case $e = \mathbf{choose}\ \nu X.\ e'$ follows by typing.

case $e = v\ \mathbf{handle}\ \Theta$.

1. by derivation, $\Sigma; \cdot \vdash v : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : A[C'] \Rightarrow A[C]$
2. by canonical forms lemma, the support of v can be arbitrary, and in particular $\Sigma; \cdot \vdash v : A\,[C]$

case $e = P[\mathbf{raise}_X\ v]\ \mathbf{handle}\ \Theta$, where $X{:}B' \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash P[\mathbf{raise}_X\ v] : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : A[C'] \Rightarrow A[C]$
2. by replacement lemma, there exist B and $D \sqsubseteq C'$ such that $\Sigma; \cdot \vdash \mathbf{raise}_X\ v : B\,[D]$
3. by typing rules, there must be $X \in D$, and $\Sigma; \cdot \vdash v : B'\,[D]$
4. in particular, $X \in C'$
5. by canonical forms lemma, support of a value can be arbitrary; in particular, $\Sigma; \cdot \vdash v : B'\,[C]$
6. by handler application lemma, $\Sigma; \cdot \vdash \Theta(X)(v) : A\,[C]$
7. since $\Sigma, e \longrightarrow \Sigma, \Theta(X)(v)$, this finishes the proof

■

Lemma 71 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

1. by evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$
2. by replacement lemma, there exist B and D such that $\Sigma; \cdot; \cdot \vdash r : B\,[D]$
3. by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash r' : B\,[D]$
4. by relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$
5. since $e' = E[r']$ this proves the lemma

■

Lemma 72 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, then there exists a term e' and a context Σ', such that $\Sigma, r \longrightarrow \Sigma', e'$.

Proof: By induction on the derivation of e.

case $r = v_1\ v_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \cdot \vdash v_2 : A_1\,[C]$
2. by canonical forms lemma, $v_1 = \lambda x{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, v_1\ v_2 \longrightarrow \Sigma, [v_2/x]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [v_2/x]e_1$

case $e = \textbf{let box}\ u = v_1\ \textbf{in}\ e_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : \Box_{C_1} B\,[C]$, and $\Sigma; u{:}B[C_1] \vdash e_2 : A\,[C]$
2. by canonical forms lemma, $v_1 = \textbf{box}\ e_1$
3. by reduction rules, $\Sigma, \textbf{let box}\ u = v_1\ \textbf{in}\ e_2 \longrightarrow \Sigma, [e_2/u]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [e_2/u]e_1$

case $e = \textbf{choose}\ v_1$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \twoheadrightarrow A\,[C]$
2. by canonical forms lemma, $v_1 = \nu X{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, \textbf{choose}\ v_1 \longrightarrow (\Sigma, X{:}A_1), e_1$
4. pick $\Sigma' = (\Sigma, X{:}A_1)$ and $e' = e_1$

case $e = v\ \textbf{handle}\ \Theta$.

1. by reduction rules, $\Sigma, v\ \textbf{handle}\ \Theta \longrightarrow \Sigma, v$
2. pick $\Sigma' = \Sigma$ and $e' = v$

case $e = P[\textbf{raise}_X\ v]\ \textbf{handle}\ \Theta$, where $X{:}B \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash P[\textbf{raise}_X\ v] : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : A[C'] \Rightarrow A[C]$
2. by replacement lemma, there exist B' and $D' \sqsubseteq C'$ such that $\Sigma; \cdot \vdash \textbf{raise}_X\ v : B'\,[D']$
3. by typing rules, it must be $X \in D' \sqsubseteq C'$
4. because $X \in C'$, $\Theta(X)(v)$ is well-defined.
5. pick $\Sigma' = \Sigma$ and $e' = \Theta(X)(v)$

■

Lemma 73 (Unique decomposition)
For every expression e, either:

1. *e is a value, or*
2. *$e = P[\textbf{raise}_X\ v]$, for a unique pure context P, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

Proof: By induction on the structure of e.

case $e = c, x, u$ are values (albeit the last two are not closed).

case $e = \lambda x.\ e', \nu X.\ e', \textbf{box}\ e'$ are all values.

case $e = e_1\ e_2$, and e_1 is not a value, nor a raise in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = E_1\ e_2$

case $e = v_1\ e_2$, and e_2 is not a value, nor a raise in pure context.

1. by induction hypothesis, $e_2 = E_2[r]$

2. pick $E = v_1\ E_2$

case $e = v_1\ v_2$. Pick $E = []$ and $r = e$.

case $e = v_1\ P_1[\mathbf{raise}_X\ v]$, Pick $P = v_1\ P_1$ and the second statement of the lemma holds.

case $e = E_1[\mathbf{raise}_X\ v]\ e_2$. Pick $P = P_1\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$, where e_1 is not a value nor a raise in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$. Pick $E = []$ and $r = e$.

case $e = \mathbf{let\ box}\ u = P_1[\mathbf{raise}_X\ v]\ \mathbf{in}\ e_2$. Pick $P = \mathbf{let\ box}\ u = P_1\ \mathbf{in}\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{choose}\ e_1$ and e_1 is not a value nor a raise in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{choose}\ E_1$

case $e = \mathbf{choose}\ v_1$. Pick $E = []$ and $r = e$.

case $e = \mathbf{choose}\ P_1[\mathbf{raise}_X\ v]$. Pick $E = \mathbf{choose}\ P_1$, and the second statement holds.

case $e = \mathbf{raise}_X\ e_1$, and e_1 is not a value, nor a raise in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{raise}_X\ E_1$

case $e = \mathbf{raise}_X\ v$. Pick $P = []$ and the second statement holds.

case $e = \mathbf{raise}_X\ P_1[\mathbf{raise}_Y\ v]$. Pick $P = \mathbf{raise}_X\ P_1$ and the second statement holds.

case $e = e_1\ \mathbf{handle}\ \Theta$ and e_1 is not a value, nor a raise in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = E_1\ \mathbf{handle}\ \Theta$

case $e = v\ \mathbf{handle}\ \Theta$. Pick $E = []$ and $r = e$.

case $e = P[\mathbf{raise}_X\ v]\ \mathbf{handle}\ \Theta$. Pick $E = []$ and $r = e$.

■

Lemma 74 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof:

1. because e has empty support, by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$
2. if e is not a value, by replacement lemma, there exists B and C such that $\Sigma; \cdot \vdash r : B\,[C]$
3. by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$
4. by evaluation rules, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$
5. pick $e' = E[e_1]$

■

Lemma 75 (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 the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof: The most important case is when $n = 1$, the reset follows by induction on n, by using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$.

1. by evaluation rules, there exists E and r such that $e = E[r]$
2. by unique decomposition lemma, these are unique.
3. by evaluation rules, $\Sigma, r \longrightarrow \Sigma_i, e_i'$ and $e_i = E[e_i']$.
4. The proof proceeds by analysis of possible reduction cases:
   (a) If $r = (\lambda x.\ e)\ v$, or $r = \textbf{let box}\ u = \textbf{box}\ e_1\ \textbf{in}\ e_2$, or $r = \textbf{reset}_X\ v$, or $r = v\ \textbf{handle}\ \Theta$, or $r = P[\textbf{raise}_X\ v]\ \textbf{handle}\ \Theta$, the reducts are unique, i.e. $e_1' = e_2'$, and thus $e_1 = e_2$, so the identity permutation satisfies the conditions.
   (b) If $r = \textbf{choose}\ \nu X{:}A.\ e$, 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.

■

8 Proofs for catch-and-throw calculus

Lemma 76 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$ and $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, then $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Proof:
By induction on the derivation of e_2.

case $e_2 = x, v$ obvious.

case $e_2 = u$ follows by support weakening.

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C], x{:}B') \vdash e' : B''\,[\,]$
2. by induction hypothesis, $\Sigma; (\Delta, x{:}B') \vdash [e_1/u]e' : B''\,[\,]$
3. result follows by typing

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B' \to B\,[D]$ and also $\Sigma; (\Delta, u{:}A[C]) \vdash e'' : B'\,[D]$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B' \to B\,[D]$, and also $\Sigma; \Delta \vdash [e_1/u]e'' : B'\,[D]$
3. result follows by typing

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D']$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D']$
3. result follows by typing

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

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

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

1. by derivation, $(\Sigma, X{:}B'); (\Delta, u{:}A[C]) \vdash e' : B''\,[\,]$ and also $X \notin \mathbf{fn}(B'')$
2. by induction hypothesis, $(\Sigma, X{:}B'); \Delta \vdash [e_1/u]e' : B''\,[\,]$
3. result follows by typing

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

1. by derivation, $\Sigma'; (\Delta, u{:}A[C]) \vdash e' : B' \nrightarrow B\,[D]$
2. by induction hypothesis, $\Sigma'; \Delta \vdash [e_1/u]e' : B' \nrightarrow B\,[D]$
3. result follows by typing

case $e_2 = \mathbf{throw}_X\ e'$, where $X{:}B' \in \Sigma$, and $X \in D$.

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D]$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D]$
3. result follows by typing

case $e_2 = \mathbf{catch}_X\ e'$, where $X{:}B \in \Sigma$.

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B\,[D, X]$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D, X]$
3. result follows by typing

■

Lemma 77 (Replacement)
If $\Sigma; \Delta \vdash E[e] : A\,[C]$, then there exist a type B and a support D such that

1. *$\Sigma; \Delta \vdash e : B\,[D]$, and*
2. *if Σ', Δ' extend Σ, Δ and $\Sigma'; \Delta' \vdash e' : B\,[D]$, then $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$*

Proof: By induction on the structure of E.

case $E = [\,]$. obvious

case $E = E_1\ e_2$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash e_2 : A_1\,[C]$
2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \to A\,[C]$
4. by weakening $\Sigma'; \Delta' \vdash e_2 : A_1\,[C]$
5. result follows by typing

case $E = v_1\ E_2$.

1. by derivation, $\Sigma; \Delta \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash E_2[e] : A_1\,[C]$
2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_2[e'] : A_1\,[C]$
4. by weakening, $\Sigma'; \Delta' \vdash v_1 : A_1 \to A\,[C]$
5. result follows by typing

case $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : \Box_{C_1} A_1\,[C]$, and $\Sigma; (\Delta, u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : \Box_{C_1} A_1\,[C]$
4. by weakening, $\Sigma'; (\Delta', u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
5. result follows by typing

case $E = \mathbf{choose}\ E_1$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \twoheadrightarrow A\,[C]$
2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \twoheadrightarrow A\,[C]$
4. result follows by typing

case $E = \mathbf{throw}_X\ E_1$, where $X{:}B' \in \Sigma$, and $X \in C$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : B'\,[C]$
2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : B'\,[C]$
4. result follows by typing

case $E = \mathbf{catch}_X\ E_1$, and $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : A\,[C, X]$

2. by induction hypothesis, there exist B and D such that $\Sigma; \Delta \vdash e : B\,[D]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[D]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A\,[C, X]$
4. result follows by typing

■

Lemma 78 (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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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 79 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

case $e = (\lambda x.\ e')\ v$ follows by value substitution principle.

case $e = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$ follows by expression substitution principle.

case $e = \mathbf{choose}\ \nu X.\ e'$ follows by typing.

case $e = \mathbf{catch}_X\ v$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash v : A\,[C, X]$
2. by canonical forms lemma, the support of v can be arbitrary, and in particular $\Sigma; \cdot \vdash v : A\,[C]$

case $e = \mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash E[\mathbf{throw}_X\ v] : A\,[C, X]$
2. by replacement lemma, there exist B and D such that $\Sigma; \cdot \vdash \mathbf{throw}_X\ v : B\,[D]$
3. by typing rules, there must be $X \in D$, and $\Sigma; \cdot \vdash v : A\,[D]$
4. by canonical forms lemma, support of a value can be arbitrary; in particular, $\Sigma; \cdot \vdash v : A\,[C]$
5. Since $\Sigma, e \longrightarrow \Sigma, v$, this finishes the proof

■

Lemma 80 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

1. by evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$

2. by replacement lemma, there exist B and D such that $\Sigma; \cdot; \cdot \vdash r : B\,[D]$
3. by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash r' : B\,[D]$
4. by relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$
5. since $e' = E[r']$ this proves the lemma

■

Lemma 81 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, then there exists a term e' and a context Σ', such that $\Sigma, r \longrightarrow \Sigma', e'$.

Proof: By induction on the derivation of e.

case $r = v_1\ v_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \cdot \vdash v_2 : A_1\,[C]$
2. by canonical forms lemma, $v_1 = \lambda x{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, v_1\ v_2 \longrightarrow \Sigma, [v_2/x]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [v_2/x]e_1$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : \Box_{C_1} B\,[C]$, and $\Sigma; u{:}B[C_1] \vdash e_2 : A\,[C]$
2. by canonical forms lemma, $v_1 = \mathbf{box}\ e_1$
3. by reduction rules, $\Sigma, \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2 \longrightarrow \Sigma, [e_2/u]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [e_2/u]e_1$

case $e = \mathbf{choose}\ v_1$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \mathrel{-\!\!\!\rightarrow} A\,[C]$
2. by canonical forms lemma, $v_1 = \nu X{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, \mathbf{choose}\ v_1 \longrightarrow (\Sigma, X{:}A_1), e_1$
4. pick $\Sigma' = (\Sigma, X{:}A_1)$ and $e' = e_1$

case $e = \mathbf{catch}_X\ v$, where $X{:}A \in \Sigma$.

1. by reduction rules, $\Sigma, \mathbf{catch}_X\ v \longrightarrow \Sigma, v$
2. pick $\Sigma' = \Sigma$ and $e' = v$

case $e = \mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash E[\mathbf{throw}_X\ ke_1] : A\,[C, X]$
2. by replacement lemma, there exist B and D such that $\Sigma; \cdot \vdash \mathbf{throw}_X\ v : B\,[D]$
3. by typing rules, it must be $B = A$ and $X \in D$ and $\Sigma; \cdot \vdash v : A\,[D]$
4. by cannonical forms lemma, v can have arbitrary support; in particular $\Sigma; \cdot \vdash v : A\,[C]$
5. pick $\Sigma' = \Sigma$ and $e' = v$

■

Lemma 82 (Unique decomposition)
For every expression e, either:

1. *e is a value, or*
2. *$e = E[\mathbf{throw}_X\ v]$, for a unique context E which does not catch X, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

Proof: By induction on the structure of e.

case $e = c, x, u$ are values (albeit the last two are not closed).

case $e = \lambda x.\ e', \nu X.\ e', \mathbf{box}\ e'$ are all values.

case $e = \mathbf{throw}_X\ e'$.

1. if e' is a value, then pick $E = []$ and the second statement of the lemma holds
2. if $e' = E_1[\mathbf{throw}_Y\ v]$ (and E_1 doesn't catch Y), pick $E = \mathbf{throw}_X\ E_1$, and the second statement of the lemma holds
3. if $e' = E_1[r]$, pick $E = \mathbf{throw}_X\ E_1$ and the third statement of the lemma holds

case $e = e_1\ e_2$, and e_1 is not a value, nor a throw in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = E_1\ e_2$

case $e = v_1\ e_2$, and e_2 is not a value, nor a throw in pure context.

1. by induction hypothesis, $e_2 = E_2[r]$
2. pick $E = v_1\ E_2$

case $e = v_1\ v_2$. Pick $E = []$ and $r = e$.

case $e = v_1\ E_1[\mathbf{throw}_X\ v]$, where E_1 doesn't catch X. Pick $E = v_1\ E_1$ and the second statement of the lemma holds.

case $e = E_1[\mathbf{throw}_X\ v]\ e_2$, where E_1 doesn't catch X. Pick $E = E_1\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$, where e_1 is not a value nor a throw in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$. Pick $E = []$ and $r = e$.

case $e = \mathbf{let\ box}\ u = E_1[\mathbf{throw}_X\ v]\ \mathbf{in}\ e_2$. Pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{choose}\ e_1$ and e_1 is not a value nor a throw in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{choose}\ E_1$

case $e = \mathbf{choose}\ v_1$. Pick $E = []$ and $r = e$.

case $e = \mathbf{choose}\ E_1[\mathbf{throw}_X\ v]$. Pick $E = \mathbf{choose}\ E_1$, and the second statement holds.

case $e = \mathbf{catch}_X\ e_1$, and e_1 is not a value, nor a throw in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{reset}_X\ E_1$

case $e = \mathbf{catch}_X\ v$. Pick $E = []$ and $r = e$.

case $e = \mathbf{catch}_X\ E_1[\mathbf{throw}_Y\ v]$, and E_1 doesn't catch X.

1. if $Y = X$ then pick $E = []$ and $r = e$
2. otherwise, pick $E = \mathbf{catch}_X\ E_1$, and the second statement of the lemma holds

■

Lemma 83 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof:

1. because e has empty support, by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$
2. if e is not a value, by replacement lemma, there exists B and C such that $\Sigma; \cdot; \cdot \vdash r : B\,[C]$
3. by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$
4. by evaluation rules, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$
5. pick $e' = E[e_1]$

■

Lemma 84 (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 the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof: The most important case is when $n = 1$, the reset follows by induction on n, by using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$.

1. by evaluation rules, there exists E and r such that $e = E[r]$
2. by unique decomposition lemma, these are unique.
3. by evaluation rules, $\Sigma, r \longrightarrow \Sigma_i, e_i'$ and $e_i = E[e_i']$.
4. The proof proceeds by analysis of possible reduction cases:
   (a) If $r = (\lambda x.\ e)\ v$, or $r = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$, or $r = \mathbf{catch}_X\ v$, or $r = \mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$, the reducts are unique, i.e. $e_1' = e_2'$, and thus $e_1 = e_2$, so the identity permutation satisfies the conditions.

(b) If $r = \textbf{choose}\ \nu X{:}A.\ e$, 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.

■

9 Proofs for composable continuations

Lemma 85 (Expression substitution principle)
If $\Sigma; \Delta \vdash e_1 : A\,[C]$ *and* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$.

Proof:
By induction on the derivation of e_2.

case $e_2 = x, v$ obvious.

case $e_2 = u$ follows by support weakening.

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C], x{:}B') \vdash e' : B''\,[\,]$
2. by induction hypothesis, $\Sigma; (\Delta, x{:}B') \vdash [e_1/u]e' : B''\,[\,]$
3. result follows by typing

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B' \to B\,[D]$ and also $\Sigma; (\Delta, u{:}A[C]) \vdash e'' : B'\,[D]$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B' \to B\,[D]$, and also $\Sigma; \Delta \vdash [e_1/u]e'' : B'\,[D]$
3. result follows by typing

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

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D']$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D']$
3. result follows by typing

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

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

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

1. by derivation, $(\Sigma, X{:}B'); (\Delta, u{:}A[C]) \vdash e' : B''\,[\,]$ and also $X \notin \textbf{fn}(B'')$
2. by induction hypothesis, $(\Sigma, X{:}B'); \Delta \vdash [e_1/u]e' : B''\,[\,]$
3. result follows by typing

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

1. by derivation, $\Sigma'; (\Delta, u{:}A[C]) \vdash e' : B' \nrightarrow B\,[D]$

2. by induction hypothesis, $\Sigma'; \Delta \vdash [e_1/u]e' : B' \nrightarrow B\,[D]$
3. result follows by typing

case $e_2 = \mathbf{shift}_X\ k.\ e'$, where $X{:}B' \in \Sigma$, and $D = (D', X)$.

1. by derivation, $\Sigma; (\Delta, u{:}A[C], k{:}\Box_{D',X} B \to \Box_{D',X} B') \vdash e' : B'\,[D']$
2. by induction hypothesis, $\Sigma; (\Delta, k{:}\Box_{D',X} B \to \Box_{D',X} B') \vdash [e_1/u]e' : B'\,[D']$
3. result follows by typing

case $e_2 = \mathbf{reset}_X\ e'$, where $X{:}B \in \Sigma$.

1. by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B\,[D', X]$, where $D' \sqsubseteq D$
2. by induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D', X]$
3. result follows by typing

■

Lemma 86 (Evaluation context decomposition)
If E is an evaluation context, then either:

1. *E is a pure context, or*
2. *there exist unique evaluation context E' and pure context P' such that $E = E'[\mathbf{reset}_X\ P']$*

Proof: By induction on the structure of E.

case $E = []$ is obviously pure.

case $E = E_1\ e_2$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[\mathbf{reset}_X\ P']$ in which case pick $E' = E_1'\ e_2$.

case $E = v_1\ E_2$.

By induction hypothesis, E_2 is either pure, in which case E is pure as well, or $E_2 = E_2'[\mathbf{reset}_X\ P']$ in which case pick $E' = v_1\ E_2'$.

case $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[\mathbf{reset}_X\ P']$ in which case pick $E' = \mathbf{let\ box}\ u = E_1'\ \mathbf{in}\ e_2$.

case $E = \mathbf{choose}\ E_1$.

By induction hypothesis, E_1 is either pure, in which case E is pure as well, or $E_1 = E_1'[\mathbf{reset}_X\ P']$ in which case pick $E' = \mathbf{choose}\ E_1'$.

case $E = \mathbf{reset}_X\ E_1$.

By induction hypothesis, E_1 is either pure, in which case pick $E' = []$ and $P = E_1$, or $E_1 = E_1'[\mathbf{reset}_X\ P']$ in which case pick $E' = \mathbf{reset}_X\ E_1'$.

■

Lemma 87 (Replacement)
1. *If $\Sigma; \Delta \vdash P[e] : A\,[C]$, 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* $\Sigma'; \Delta' \vdash P[e'] : A\,[C]$

2. *if* $\Sigma; \Delta; \Gamma \vdash E[e] : A\,[C]$, *then there exist a type* B *and a support* D *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* $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$

Proof:

1. By induction on the structure of P.

case $P = [\,]$. obvious

case $P = P_1\ e_1$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash e_1 : A_1\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : A_1 \to A\,[C]$

(d) by weakening $\Sigma'; \Delta' \vdash e_1 : A_1\,[C]$

(e) result follows by typing

case $P = v_1\ P_1$.

(a) by derivation, $\Sigma; \Delta \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash P_1[e] : A_1\,[C]$

(b) by induction hypothesis, there exist B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : A_1\,[C]$

(d) by weakening, $\Sigma'; \Delta' \vdash v_1 : A_1 \to A\,[C]$

(e) result follows by typing

case $P = \mathbf{let\ box}\ u = P_1\ \mathbf{in}\ e_1$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : \Box_{C_1} A_1\,[C]$, and $\Sigma; (\Delta, u{:}A_1[C_1]) \vdash e_1 : A\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : \Box_{C_1} A_1\,[C]$

(d) by weakening, $\Sigma'; (\Delta', u{:}A_1[C_1]) \vdash e_1 : A\,[C]$

(e) result follows by typing

case $P = \mathbf{choose}\ P_1$.

(a) by derivation, $\Sigma; \Delta \vdash P_1[e] : A_1 \twoheadrightarrow A\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash P_1[e'] : A_1 \twoheadrightarrow A\,[C]$

(d) result follows by typing

2. by evaluation context decomposition lemma, it is enough to consider two cases:

case E is pure follows from the first statement of the lemma.

case $E = E_1[\mathbf{reset}_X\ P]$, where $X{:}B' \in \Sigma$.

(a) by second induction hypothesis, there exists B_1 and D_1 such that $\Sigma; \Delta \vdash \mathbf{reset}_X\ P[e] : B_1\,[D_1]$

(b) by typing, it must be $B_1 = B'$ and $\Sigma; \Delta \vdash P[e] : B_1\,[D', X]$, where $D' \sqsubseteq D_1$

(c) by the first induction hypothesis, there exist B such that $\Sigma; \Delta \vdash e : B\,[D', X]$
(d) pick $D = (D', X)$ for the first part of the proof
(e) also by the first induction hypothesis, if $\Sigma'; \Delta' \vdash e' : B\,[D', X]$ then $\Sigma'; \Delta' \vdash P[e'] : B_1\,[D', X]$
(f) by typing, $\Sigma'; \Delta' \vdash \mathbf{reset}_X\ P[e'] : B_1\,[D_1]$
(g) by induction hypothesis, $\Sigma'; \Delta' \vdash E_1[\mathbf{reset}_X\ P[e']] : A\,[C]$

■

Lemma 88 (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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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 89 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

case $e = (\lambda x.\ e')\ v$ follows by value substitution principle.

case $e = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$ follows by expression substitution principle.

case $e = \mathbf{choose}\ \nu X.\ e'$ follows by typing.

case $e = \mathbf{reset}_X\ v$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash v : A\,[C', X]$, where $C' \sqsubseteq C$
2. by canonical forms lemma, the support of v can be arbitrary, and in particular $\Sigma; \cdot \vdash v : A\,[C]$

case $e = \mathbf{reset}_X\ P[\mathbf{shift}_X\ k.\ e']$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash P[\mathbf{shift}_X\ k.\ e'] : A\,[C', X]$, where $C' \sqsubseteq C$
2. by replacement lemma for pure contexts, there exists B such that $\Sigma; \cdot \vdash \mathbf{shift}_X\ k.\ e' : B\,[C', X]$
3. also by replacement lemma, $\Sigma; u{:}B[C', X] \vdash P[u] : A\,[C', X]$
4. thus $\Sigma; \cdot \vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u] : (\Box_{C',X} B \to \Box_{C',X} A)\,[\,]$
5. from the typing (2), $\Sigma; k{:}(\Box_{C',X} B \to \Box_{C',X} A) \vdash e' : A\,[C']$
6. from (4) and (5), if we set $K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]$, by substitution principle, we get $\Sigma; \cdot \vdash [K/k]e' : A\,[C']$
7. by support weakening, $\Sigma; \cdot \vdash [K/k]e' : A\,[C]$, because $C' \sqsubseteq C$
8. since it is exactly $\Sigma, e \longrightarrow \Sigma, [K/k]e'$, this proves the case

■

Lemma 90 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

1. by evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$
2. by replacement lemma, there exist B and D such that $\Sigma; \cdot \vdash r : B\,[D]$
3. by subject reduction lemma, Σ' extends Σ, and $\Sigma'; \cdot \vdash r' : B\,[D]$
4. by relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$
5. since $e' = E[r']$ this proves the lemma

■

Lemma 91 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \longrightarrow \Sigma', e'$.

Proof: By induction on the derivation of e.

case $r = v_1\ v_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \cdot \vdash v_2 : A_1\,[C]$
2. by canonical forms lemma, $v_1 = \lambda x{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, v_1\ v_2 \longrightarrow \Sigma, [v_2/x]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [v_2/x]e_1$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : \Box_{C_1} B\,[C]$, and $\Sigma; u{:}B[C_1] \vdash e_2 : A\,[C]$
2. by canonical forms lemma, $v_1 = \mathbf{box}\ e_1$
3. by reduction rules, $\Sigma, \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2 \longrightarrow \Sigma, [e_2/u]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [e_2/u]e_1$

case $e = \mathbf{choose}\ v_1$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \twoheadrightarrow A\,[C]$
2. by canonical forms lemma, $v_1 = \nu X{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, \mathbf{choose}\ v_1 \longrightarrow (\Sigma, X{:}A_1), e_1$
4. pick $\Sigma' = (\Sigma, X{:}A_1)$ and $e' = e_1$

case $e = \mathbf{reset}_X\ v$, where $X{:}A \in \Sigma$.

1. by reduction rules, $\Sigma, \mathbf{reset}_X\ v \longrightarrow \Sigma, v$
2. pick $\Sigma' = \Sigma$ and $e' = v$

case $e = \mathbf{reset}_X\ (P[\mathbf{shift}_Y\ k.\ e_1])$, where $X{:}A \in \Sigma$.

1. by derivation, $\Sigma; \cdot \vdash P[\mathbf{shift}_Y\ k.\ e_1] : A\,[C', X]$, where $C' \sqsubseteq C$
2. by replacement lemma, there exists B such that $\Sigma; \cdot \vdash \mathbf{shift}_Y\ k.\ e_1 : B\,[C', X]$

3. by typing rules, $Y = X$
4. by reduction rules, $\Sigma, \mathbf{reset}_X\ (P[\mathbf{shift}_X\ k.\ e_1]) \longrightarrow \Sigma, [K/k]e_1$, where $K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]$
5. pick $\Sigma' = \Sigma$ and $e' = [K/k]e_1$

■

Lemma 92 (Unique decomposition)
For every expression e, either:

1. *e is a value, or*
2. *$e = P[\mathbf{shift}_X\ k.\ e']$, for a unique pure context P, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

Proof: By induction on the structure of e.

case $e = c, x, u$ are values (albeit the last two are not closed).

case $e = \lambda x.\ e', \nu X.\ e', \mathbf{box}\ e'$ are all values.

case $e = \mathbf{shift}_X\ k.\ e'$. Pick $P = []$.

case $e = e_1\ e_2$, and e_1 is not a value, nor a shift in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = E_1\ e_2$

case $e = v_1\ e_2$, and e_2 is not a value, nor a shift in pure context.

1. by induction hypothesis, $e_2 = E_2[r]$
2. pick $E = v_1\ E_2$

case $e = v_1\ v_2$. Pick $E = []$ and $r = e$.

case $e = v_1\ (P_2[\mathbf{shift}_X\ k.\ e'])$. Pick $P = v_1\ P_2$.

case $e = (P_1[\mathbf{shift}_X\ k.\ e'])\ e_2$. Pick $P = P_1\ e_2$.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$, where e_1 is not a value nor a shift in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$. Pick $E = []$ and $r = e$.

case $e = \mathbf{let\ box}\ u = (P_1[\mathbf{shift}_X\ k.\ e'])\ \mathbf{in}\ e_2$. Pick $P = \mathbf{let\ box}\ u = P_1\ \mathbf{in}\ e_2$.

case $e = \mathbf{choose}\ e_1$ and e_1 is not a value nor a shift in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{choose}\ E_1$

case $e = \mathbf{choose}\ v_1$. Pick $E = []$ and $r = e$.

case $e = \mathbf{choose}\ (P_1[\mathbf{shift}_X\ k.\ e'])$. Pick $P = \mathbf{choose}\ P_1$.

case $e = \mathbf{reset}_X\ e_1$, and e_1 is not a value, nor a shift in pure context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{reset}_X\ E_1$

case $e = \mathbf{reset}_X\ v$. Pick $E = []$ and $r = e$.

case $e = \mathbf{reset}_X\ (P[\mathbf{shift}_Y\ k.\ e_1])$. Pick $E = []$ and $r = e$.

■

Lemma 93 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof:

1. because e has empty support, by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$
2. if e is not a value, by replacement lemma, there exists B and C such that $\Sigma; \cdot \vdash r : B\,[C]$
3. by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$
4. by evaluation rules, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$
5. pick $e' = E[e_1]$

■

Lemma 94 (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 the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof: The most important case is when $n = 1$, the reset follows by induction on n, by using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$.

1. by evaluation rules, there exists E and r such that $e = E[r]$
2. by unique decomposition lemma, these are unique
3. By evaluation rules, $\Sigma, r \longrightarrow \Sigma_i, e_i'$ and $e_i = E[e_i']$.
4. The proof proceeds by analysis of possible reduction cases:
   (a) If $r = (\lambda x.\ e)\ v$, or $r = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$, or $r = \mathbf{reset}_X\ v$, or $r = \mathbf{reset}_X\ P[\mathbf{shift}_X\ k.\ e]$, the reducts are unique, i.e. $e_1' = e_2'$, and thus $e_1 = e_2$, so the identity permutation satisfies the conditions.
   (b) If $r = \mathbf{choose}\ \nu X{:}A.\ e$, then it must be $e_1' = [X_1/X]e$, $e_2' = [X_2/X]e$, and $\Sigma_1 = (\Sigma, X_1{:}A)$, $\Sigma_2 = (\Sigma, X_2{:}A)$, where X_1 and X_2 are fresh names. Obviously, the involution $(X_1\ X_2)$ which swaps these two names has the required properties.

■

10 Proofs for calculus of dynamic binding

Lemma 95 (Explicit substition 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* $\Sigma; \Delta \vdash \langle \Theta \circ \Theta' \rangle : [C_1] \Rightarrow [D]$

Lemma 96 (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* $\Sigma; \Delta \vdash \langle [e_1/u]\Theta \rangle : [D'] \Rightarrow [D]$

Proof:

1. By induction on the derivation of e_2.

   case $e_2 = x, v$ obvious.

   case $e_2 = u$ follows by support weakening.

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

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

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

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

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

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

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

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

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

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

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

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

   case $e_2 = X$, where $X{:}B \in \Sigma$, and $X \in D$. Trivial, because the substitution is vacuous.

case $e_2 = \langle\Theta\rangle e'$.

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B\,[D']$, and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : [D'] \Rightarrow [D]$
(b) by first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D']$
(c) by second induction hypothesis, $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : [D'] \Rightarrow [D]$
(d) result follows by typing rule for handle

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

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

(a) by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e : B\,[D]$, and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta'\rangle : [D' \setminus \{X\}] \Rightarrow [D]$
(b) by the first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e : B\,[D]$
(c) by the second induction hypothesis, $\Sigma; \Delta \vdash \langle[e_1/u]\Theta'\rangle : [D' \setminus \{X\}] \Rightarrow [D]$
(d) result follows by the typing rule for substitution application

■

Lemma 97 (Replacement)
If $\Sigma; \Delta \vdash E[e] : A\,[C]$, then there exist a type B such that

1. *$\Sigma; \Delta \vdash e : B\,[C]$, and*
2. *if Σ', Δ' extend Σ, Δ, and $\Sigma'; \Delta' \vdash e' : B\,[C]$, then $\Sigma'; \Delta' \vdash E[e'] : A\,[C]$*

Proof:
By induction on the structure of E.

case $E = [\,]$. obvious

case $E = E_1\ e_2$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash e_2 : A_1\,[C]$
2. by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \to A\,[C]$
4. by weakening $\Sigma'; \Delta' \vdash e_2 : A_1\,[C]$
5. result follows by typing

case $E = v_1\ E_2$.

1. by derivation, $\Sigma; \Delta \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash E_2[e] : A_1\,[C]$
2. by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_2[e'] : A_1\,[C]$
4. by weakening, $\Sigma'; \Delta' \vdash v_1 : A_1 \to A\,[C]$
5. result follows by typing

case $E = \textbf{let box}\ u = E_1\ \textbf{in}\ e_2$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : \Box_{C_1} A_1\,[C]$, and $\Sigma; (\Delta, u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
2. by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : \Box_{C_1} A_1\,[C]$
4. by weakening, $\Sigma'; (\Delta', u{:}A_1[C_1]) \vdash e_2 : A\,[C]$
5. result follows by typing

case $E = \mathbf{choose}\ E_1$.

1. by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \twoheadrightarrow A\,[C]$
2. by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$
3. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \twoheadrightarrow A\,[C]$
4. result follows by typing

case $E = \langle \sigma, X \to E_1, \Theta \rangle e_1$, where $X{:}B' \in \Sigma$.

1. by derivation, $\Sigma; \Delta \vdash e_1 : A'C'$, and $\Sigma; \Delta \vdash \langle \sigma, X \to E_1[e], \Theta \rangle : [C'] \Rightarrow [C]$
2. by typing of substitutions, $\Sigma; \Delta \vdash E_1[e] : B'\,[C]$
3. first statement now follows by induction hypothesis.
4. again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : B'\,[C]$
5. result follows by typing of substitutions

■

Lemma 98 (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 $A = A_1 \twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

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 99 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

case $e = (\lambda x.\ e')\ v$ follows by value substitution principle.

case $e = \mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2$ follows by expression substitution principle.

case $e = \mathbf{choose}\ \nu X.\ e_1$ follows by typing.

case $e = \langle \sigma \rangle e_1$.

1. by derivation, $\Sigma; \cdot \vdash e_1 : A\,[C']$, and $\Sigma; \cdot \vdash \langle \sigma \rangle : [C'] \Rightarrow [C]$
2. by substitution principle, $\Sigma; \cdot \vdash \{\sigma\} e_1 : A\,[C]$

3. by definition, $e' = \{\sigma\}e_1$, which finishes the proof

■

Lemma 100 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[C]$ *and* $\Sigma, e \longmapsto \Sigma', e'$, *then* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : A\,[C]$.

Proof:

1. by evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$
2. by replacement lemma, there exists B such that $\Sigma; \cdot; \cdot \vdash r : B\,[C]$
3. by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot \vdash r' : B\,[C]$
4. by relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$
5. since $e' = E[r']$ this proves the lemma

■

Lemma 101 (Progress for $\longrightarrow$)
If $\Sigma; \cdot \vdash r : A\,[C]$, *then there exists a term* e' *and a context* Σ', *such that* $\Sigma, r \longrightarrow \Sigma', e'$.

Proof: By induction on the derivation of e.

case $r = v_1\ v_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \cdot \vdash v_2 : A_1\,[C]$
2. by canonical forms lemma, $v_1 = \lambda x{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, v_1\ v_2 \longrightarrow \Sigma, [v_2/x]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [v_2/x]e_1$

case $e = \textbf{let box}\ u = v_1\ \textbf{in}\ e_2$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : \Box_{C_1} B\,[C]$, and $\Sigma; u{:}B[C_1] \vdash e_2 : A\,[C]$
2. by canonical forms lemma, $v_1 = \textbf{box}\ e_1$
3. by reduction rules, $\Sigma, \textbf{let box}\ u = v_1\ \textbf{in}\ e_2 \longrightarrow \Sigma, [e_2/u]e_1$
4. pick $\Sigma' = \Sigma$ and $e' = [e_2/u]e_1$

case $e = \textbf{choose}\ v_1$.

1. by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \twoheadrightarrow A\,[C]$
2. by canonical forms lemma, $v_1 = \nu X{:}A_1.\ e_1$
3. by reduction rules, $\Sigma, \textbf{choose}\ v_1 \longrightarrow (\Sigma, X{:}A_1), e_1$
4. pick $\Sigma' = (\Sigma, X{:}A_1)$ and $e' = e_1$

case $e = \langle\sigma\rangle e_1$.

1. by reduction rules, $\Sigma, \langle\sigma\rangle e_1 \longrightarrow \Sigma, \{\sigma\}e_1$
2. pick $\Sigma' = \Sigma$ and $e' = \{\sigma\}e_1$

■

Lemma 102 (Unique decomposition)
For every expression e, either:

1. *e is a value, or*
2. *$e = E[X]$, for a unique evaluation context E and a name X, or*
3. *$e = E[r]$ for a unique evaluation context E and a redex r.*

Proof: By induction on the structure of e.

case $e = c, x, u$ are values (albeit the last two are not closed).

case $e = \lambda x.\ e', \nu X.\ e', \mathbf{box}\ e'$ are all values.

case $e = X$. Pick $E = []$ and the second statement of the lemma holds.

case $e = e_1\ e_2$, and e_1 is not a value, nor a name in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = E_1\ e_2$

case $e = v_1\ e_2$, and e_2 is not a value, nor a name in context.

1. by induction hypothesis, $e_2 = E_2[r]$
2. pick $E = v_1\ E_2$

case $e = v_1\ v_2$. Pick $E = []$ and $r = e$.

case $e = v_1\ E_2[X]$, Pick $E = v_1\ E_2$ and the second statement of the lemma holds.

case $e = E_1[X]\ e_2$. Pick $E = E_1\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$, where e_1 is not a value nor a name in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$. Pick $E = []$ and $r = e$.

case $e = \mathbf{let\ box}\ u = E_1[X]\ \mathbf{in}\ e_2$. Pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$, and the second statement of the lemma holds.

case $e = \mathbf{choose}\ e_1$ and e_1 is not a value nor a name in context.

1. by induction hypothesis, $e_1 = E_1[r]$
2. pick $E = \mathbf{choose}\ E_1$

case $e = \mathbf{choose}\ v_1$. Pick $E = []$ and $r = e$.

case $e = \mathbf{choose}\ E_1[X]$. Pick $E = \mathbf{choose}\ E_1$, and the second statement holds.

case $e = \langle \sigma, X \to e_1, \Theta \rangle e_2$, where e' is not a value nor a name in context.

1. by induction hypothesis, $e_1 = E_1[r]$

2. pick $E = \langle \sigma, X \to E_1, \Theta \rangle e_2$

case $e = \langle \sigma, X \to E_1[Y], \Theta \rangle e_2$. Pick $E = \langle \sigma, X \to E_1[], \Theta \rangle e_2$, and the second statement holds.

case $e = \langle \sigma \rangle e_2$. Pick $E = []$ and $r = e$.

■

Lemma 103 (Progress)
If $\Sigma; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*
2. *there exists a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof:

1. because e has empty support, by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$
2. if e is not a value, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[\,]$
3. by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$
4. by evaluation rules, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$
5. pick $e' = E[e_1]$

■

Lemma 104 (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 the domain of Σ, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof: The most important case is when $n = 1$, the reset follows by induction on n, by using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$.

1. by evaluation rules, there exists E and r such that $e = E[r]$
2. by unique decomposition lemma (NUMBER), these are unique.
3. by evaluation rules, $\Sigma, r \longrightarrow \Sigma_i, e_i'$ and $e_i = E[e_i']$.
4. The proof proceeds by analysis of possible reduction cases:
   (a) If $r = (\lambda x.\ e)\ v$, or $r =$ **let box** $u =$ **box** e_1 **in** e_2, or $r = \langle \sigma \rangle e$, the reducts are unique, i.e. $e_1' = e_2'$, and thus $e_1 = e_2$, so the identity permutation satisfies the conditions.
   (b) If $r =$ **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.

■

11 Proofs for nominal possibility

Lemma 105 (Support weakening)

1. *if* $\Sigma; \Delta \vdash e : A\,[C]$ *and* $C \sqsubseteq D$, *then* $\Sigma; \Delta \vdash e : A\,[D]$
2. *if* $\Sigma; \Delta \vdash \langle \Theta \rangle : [C_1] \Rightarrow [C]$ *and* $C \sqsubseteq D$, *then* $\Sigma; \Delta \vdash \langle \Theta \rangle : [C_1] \Rightarrow [D]$
3. *if* $\Sigma; \Delta \vdash f \div_{C_1} A\,[C]$ *and* $C \sqsubseteq D$, *then* $\Sigma; \Delta \vdash f \div_{C_1} A\,[D]$ *(This one needed in the proof of Replacemenet, case 2)*

Lemma 106 (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:

1. Extend with case for dia.
2. Unchanged
3. By induction on the structure of f.

   case $f = [\Theta, e]$.
   (a) by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e : B\,[C_1]$, and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle \Theta \rangle : [C_1] \Rightarrow [D]$
   (b) by first induction hypothesis,, $\Sigma; \Delta \vdash [e_1/u]e : B\,[C_1]$
   (c) by second induction hypothesis, $\Sigma; \Delta \vdash \langle [e_1/u]\Theta \rangle : [C_1] \Rightarrow [D]$
   (d) result follows by typing rules for closures

   case $f = \textbf{let dia } x = e \textbf{ in } f'$.
   (a) by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e : \Diamond_{C'} A'\,[D]$, and $\Sigma; (\Delta, u{:}A[C], x{:}A') \vdash f' \div_{C_1} B\,[C']$
   (b) by first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e : \Diamond_{C'} A'\,[D]$
   (c) by third induction hypothesis, $\Sigma; (\Delta, x{:}A') \vdash [e_1/u]f' \div_{C_1} B\,[C']$
   (d) result follows by typing

■

Lemma 107 (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:

1. *Application to terms has to be extended to cover the case* $e = \textbf{dia } f$ *which is easy as it immediately refers to the third statement of the lemma.*
2. *Composition case is unchanged*

3. *Proof by induction on the structure of* f.

   case $f = [\Theta', e]$.

   (a) *by derivation,* $\Sigma; \Delta \vdash e : A\,[C_1]$ *and* $\Sigma; \Delta \vdash \langle \Theta' \rangle : [C_1] \Rightarrow [C]$

   (b) *by second induction hypothesis,* $\Sigma; \Delta \vdash \langle \Theta \circ \Theta' \rangle : [C_1] \Rightarrow [D]$

   (c) *result follows by typing rule for phrases*

   case $f = \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f'$.

   (a) *by derivation,* $\Sigma; \Delta \vdash e : \Diamond_{C'} A'\,[C]$, *and* $\Sigma; (\Delta, x{:}A') \vdash f' \div_{C_1} A\,[C']$

   (b) *by first induction hypothesis,* $\Sigma; \Delta \vdash \{\Theta\}e : \Diamond_{C'} A'\,[D]$

   (c) *result follows by typing*

■

Lemma 108 (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* $A = A_1 \twoheadrightarrow A_2$, *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 109 (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:

1. By induction on the structure of E.

   case $E = [\,]$. obvious

   case $E = E_1\ e_2$.

   (a) by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash e_2 : A_1\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \to A\,[C]$

(d) by weakening $\Sigma'; \Delta' \vdash e_2 : A_1\,[C]$

(e) result follows by typing

case $E = v_1\ E_2$.

(a) by derivation, $\Sigma; \Delta \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \Delta \vdash E_2[e] : A_1\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_2[e'] : A_1\,[C]$

(d) by weakening, $\Sigma'; \Delta' \vdash v_1 : A_1 \to A\,[C]$

(e) result follows by typing

case $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$.

(a) by derivation, $\Sigma; \Delta \vdash E_1[e] : \Box_{C_1} A_1\,[C]$, and $\Sigma; (\Delta, u{:}A_1[C_1]) \vdash e_2 : A\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : \Box_{C_1} A_1\,[C]$

(d) by weakening, $\Sigma'; (\Delta', u{:}A_1[C_1]) \vdash e_2 : A\,[C]$

(e) result follows by typing

case $E = \mathbf{choose}\ E_1$.

(a) by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1 \twoheadrightarrow A\,[C]$

(b) by induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[C]$

(c) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : A_1 \twoheadrightarrow A\,[C]$

(d) result follows by typing

case $E = \langle \sigma, X \to E_1, \Theta \rangle e_1$, where $X{:}B' \in \Sigma$.

(a) by derivation, $\Sigma; \Delta \vdash e_1 : A'C'$, and $\Sigma; \Delta \vdash \langle \sigma, X \to E_1[e], \Theta \rangle : [C'] \Rightarrow [C]$

(b) by typing of substitutions, $\Sigma; \Delta \vdash E_1[e] : B'\,[C]$

(c) first statement now follows by induction hypothesis.

(d) again by induction hypothesis, for every e' such that $\Sigma'; \Delta' \vdash e' : B\,[C]$, we have $\Sigma'; \Delta' \vdash E_1[e'] : B'\,[C]$

(e) result follows by typing of substitutions

2. case $F = \mathbf{let\ dia}\ x = \mathbf{dia}\ F_1\ \mathbf{in}\ f$.

(a) by derivation, $\Sigma; \Delta \vdash F_1[e] \div_{C_1} A_1\,[D]$, and $\Sigma; (\Delta, x{:}A_1) \vdash f \div_C A\,[C_1]$

(b) by second induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[D]$

(c) also, if $\Sigma'; \Delta' \vdash e' : B\,[D]$, then $\Sigma'; \Delta' \vdash F_1[e'] \div_{C_1} A_1\,[D]$

(d) result follows by typing for let-dia

case $F = \mathbf{let\ dia}\ x = E_1\ \mathbf{in}\ f$.

(a) by derivation, $\Sigma; \Delta \vdash E_1[e] : \Diamond_{C_1} A_1\,[D]$, and $\Sigma; (\Delta, x{:}A_1) \vdash f \div_C A\,[C_1]$

(b) by first induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[D]$

(c) also, if $\Sigma'; \Delta' \vdash e' : B\,[D]$, then $\Sigma'; \Delta' \vdash E_1[e'] : \Diamond_{C_1} A_1\,[D]$

(d) result follows by typing rule for let-dia

case $F = \mathbf{let\ dia}\ x = \mathbf{dia}\ [E_1]\ \mathbf{in}\ f$.

(a) by derivation, $\Sigma; \Delta \vdash E_1[e] : A_1\,[C_1]$, where $C_1 \sqsubseteq D$, and $\Sigma; (\Delta, x{:}A_1) \vdash f \div_C A\,[C_1]$
(b) by support weakening, $\Sigma; \Delta \vdash E_1[e] : A_1\,[D]$ and $\Sigma; (\Delta, x{:}A_1) \vdash f \div_C AD$
(c) by first induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[D]$
(d) also, if $\Sigma'; \Delta' \vdash e' : B\,[D]$, then $\Sigma'; \Delta' \vdash E_1[e'] : A_1\,[D]$
(e) result follows by typing for let-dia

case $F = \textbf{let dia}\ x = \textbf{dia}\ [\langle \sigma, X \to E_1, \Theta \rangle, e]\ \textbf{in}\ f$, where $X{:}B_1 \in \Sigma$.

(a) by derivation, $\Sigma; \Delta \vdash E_1[e] : B_1\,[D]$, and $\Sigma; (\Delta, x{:}A_1) \vdash f \div_C AC_1$
(b) by first induction hypothesis, there exists B such that $\Sigma; \Delta \vdash e : B\,[D]$
(c) also, if $\Sigma'; \Delta' \vdash e' : B\,[D]$, then $\Sigma'; \Delta' \vdash E_1[e'] : B_1\,[D]$
(d) result follows by typing for let-dia

3. case $F = [\ ]$. Obviously, pick $B = A$, and $C_1 = C$.

case $F = \textbf{let dia}\ x = \textbf{dia}\ F_1\ \textbf{in}\ f_1$.

(a) by derivation, $\Sigma; \Delta \vdash F_1[f] \div_{C'} A'\,[D]$, and $\Sigma; (\Delta, x{:}A') \vdash f_1 \div_C A\,[C']$
(b) by third induction hypothesis, there exist B and C_1 such that $\Sigma; \Delta \vdash f \div_{C_1} B\,[D]$
(c) also, if $\Sigma'; \Delta' \vdash f' \div_{C_1} B\,[D_1]$, then $\Sigma'; \Delta' \vdash F_1[f'] \div_{C'} A'\,[D_1]$
(d) result follows by typing rules for let-dia

■

Lemma 110 (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 $C' \subseteq \textbf{dom}(\Sigma')$*

Proof:

1. case $e = (\lambda x.\ e')\ v$ follows by substitution principle for expressions with empty support (here v) and cannonical forms lemma to weaken to C.

case $e = \textbf{let box}\ u = \textbf{box}\ e_1\ \textbf{in}\ e_2$ follows by expression substitution principle.

case $e = \textbf{choose}\ \nu X.\ e_1$ follows by typing and support weakening.

case $e = \langle \sigma' \rangle e_1$.

(a) by derivation, $\Sigma; \cdot \vdash e_1 : A\,[C']$, and $\Sigma; \cdot \vdash \langle \sigma' \rangle : [C'] \Rightarrow [\ ]$
(b) by explicit substitution principle, $\Sigma; \cdot \vdash \{\sigma'\}e_1 : A\,[\ ]$
(c) by weakening, $\Sigma; \cdot \vdash \{\sigma'\}e_1 : A\,[\ ]$
(d) by definition, $e' = \{\sigma\}e_1$, which finishes the proof

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

(a) by derivation, $X \in C$, and thus by typing for substitutions $\Sigma; \cdot \vdash \sigma(X) : A\,[\ ]$
(b) furthermore, because σ is a value substitution, $\sigma(X)$ is a value, so by cannonical forms lemma, its support can be arbitrarily weakend; in particular $\Sigma; \cdot \vdash \sigma(X) : A\,[C]$

2. case $f = \textbf{let dia}\ x = \textbf{dia}\ [\sigma_1, e]\ \textbf{in}\ f_1$.

(a) by definition, $\Sigma' = \Sigma$ and $\sigma' = \sigma \circ \sigma_1$
(b) by derivation, $\Sigma; \cdot \vdash e : B\,[C']$, and $\Sigma; \cdot \vdash \langle \sigma_1 \rangle : [C'] \Rightarrow [C]$, and $\Sigma; x{:}B \vdash f_1 \div_D A\,[C']$

(c) by explicit substitution principle, $\Sigma; \cdot \vdash \langle \sigma \circ \sigma_1 \rangle : [C'] \Rightarrow [\,]$

(d) result follows by typing rule for let-dia

case $f = \textbf{let dia } x = \textbf{dia } [v] \textbf{ in } f_1$.

(a) by definition, $\Sigma' = \Sigma$ and $\sigma' = \sigma$ and $C_1 = C$

(b) by derivation, $\Sigma; \cdot \vdash v : B\,[C_1]$ for some $C_1 \sqsubseteq C$, and $\Sigma; x{:}B \vdash f_1 \div_D A\,[C']$

(c) by cannonical forms lemma, $\Sigma; \cdot \vdash v \div_B [\,]$

(d) by support weakening, $\Sigma; x{:}B \vdash f_1 \div_D A\,[C]$

(e) by substitution principle, $\Sigma; \cdot \vdash [v/x]f_1 \div_D A\,[C]$

■

Lemma 111 (Preservation)
Let $\Sigma; \cdot; \vdash \langle \sigma \rangle : [C] \Rightarrow [\,]$. Then the following holds:

1. *if $\Sigma; \cdot \vdash e : A\,[C]$ and $\Sigma, e \overset{\sigma}{\longmapsto} \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[C]$*
2. *if $\Sigma; \cdot \vdash f \div_D A\,[C]$ and $(\Sigma, \sigma), f \longmapsto (\Sigma', \sigma'), f'$, 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 $C' \subseteq \textbf{dom}(\Sigma')$*

Proof:

1. - by evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \xrightarrow{\sigma} \Sigma', r'$ and $e' = E[r']$
   - by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$
   - by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot \vdash r' : B\,[C]$
   - by relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$
   - since $e' = E[r']$ this proves the case
2. - By evaluation rules, it is either $f = F[r]$ for some closure context F and term redex r, or $f = F[c]$ for some closure redex c.
   - if $f = F[r]$, then $\Sigma, r \xrightarrow{\sigma} \Sigma', e'$ and $f' = F[e']$, and $\sigma' = \sigma$ and $C_1 = C$
   - by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$
   - by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : B\,[C]$
   - by replacement lemma $\Sigma'; \cdot \vdash F[e'] : A\,[C]$
   - if $f = F[c]$, then $(\Sigma, \sigma), c \longrightarrow (\Sigma', \sigma'), c'$ and $f' = F[c']$
   - by replacement lemma, there exists B and D_1 such that $\Sigma; \cdot \vdash c \div_{D_1} B\,[C]$
   - by subject reduction, Σ' extends Σ, and $\Sigma'; \cdot \vdash \langle \sigma' \rangle : [C'] \Rightarrow [\,]$, and $\Sigma'; \cdot \vdash c' \div_{D_1} B\,[C']$
   - by replacement lemma, $\Sigma'; \cdot \vdash F[c'] \div_D A\,[C']$

■

Lemma 112 (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:

1. case $r = v_1\ v_2$.
   (a) by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \to A\,[C]$, and $\Sigma; \cdot \vdash v_2 : A_1\,[C]$
   (b) by canonical forms lemma, $v_1 = \lambda x{:}A_1.\ e_1$
   (c) by reduction rules, $\Sigma, v_1\ v_2 \xrightarrow{\sigma} \Sigma, [v_2/x]e_1$
   (d) pick $\Sigma' = \Sigma$ and $e' = [v_2/x]e_1$

   case $r = \textbf{let box}\ u = v_1\ \textbf{in}\ e_2$.
   (a) by derivation, $\Sigma; \cdot \vdash v_1 : \Box_{C_1} B\,[C]$, and $\Sigma; u{:}B[C_1] \vdash e_2 : A\,[C]$
   (b) by canonical forms lemma, $v_1 = \textbf{box}\ e_1$
   (c) by reduction rules, $\Sigma, \textbf{let box}\ u = v_1\ \textbf{in}\ e_2 \xrightarrow{\sigma} \Sigma, [e_2/u]e_1$
   (d) pick $\Sigma' = \Sigma$ and $e' = [e_2/u]e_1$

   case $r = \textbf{choose}\ v_1$.
   (a) by derivation, $\Sigma; \cdot \vdash v_1 : A_1 \nrightarrow A\,[C]$
   (b) by canonical forms lemma, $v_1 = \nu X{:}A_1.\ e_1$
   (c) by reduction rules, $\Sigma, \textbf{choose}\ v_1 \xrightarrow{\sigma} (\Sigma, X{:}A_1), e_1$
   (d) pick $\Sigma' = (\Sigma, X{:}A_1)$ and $e' = e_1$

   case $r = \langle\sigma_1\rangle e_1$.
   (a) by reduction rules, $\Sigma, \langle\sigma_1\rangle e_1 \xrightarrow{\sigma} \Sigma, \{\sigma\}e_1$
   (b) pick $\Sigma' = \Sigma$ and $e' = \{\sigma\}e_1$

   case $r = X$, where $X{:}A \in \Sigma$.
   (a) pick $\Sigma' = \Sigma$ and $e' = \sigma(X)$

2. case $c = \textbf{let dia}\ x = \textbf{dia}\ [\sigma_1, e]\ \textbf{in}\ f_1$.
   (a) pick $\Sigma' = \Sigma$ and $\sigma' = \sigma \circ \sigma_1$ (which is easily a value substitution), and pick a $f' = \textbf{let dia}\ x = \textbf{dia}\ [e]\ \textbf{in}\ f_1$

   case $c = \textbf{let dia}\ x = \textbf{dia}\ [v]\ \textbf{in}\ f_1$
   (a) pick $\Sigma' = \Sigma$, and $\sigma' = \sigma$ and $f' = [v/x]f_1$

■

Lemma 113 (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 induction on the structure of e and f.

1. case $e = x, u$ are values (albeit the last two are not closed).

case $e = \lambda x.\ e', \nu X.\ e', \mathbf{box}\ e'$ are all values.

case $e = X$. Pick $E = []$ and $r = X$ and the second statement of the lemma holds.

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

(a) by induction hypothesis, $e_1 = E_1[r]$

(b) pick $E = E_1\ e_2$

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

(a) by induction hypothesis, $e_2 = E_2[r]$

(b) pick $E = v_1\ E_2$

case $e = v_1\ v_2$. Pick $E = []$ and $r = e$.

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

(a) by induction hypothesis, $e_1 = E_1[r]$

(b) pick $E = \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ e_2$

case $e = \mathbf{let\ box}\ u = v_1\ \mathbf{in}\ e_2$. Pick $E = []$ and $r = e$.

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

(a) by induction hypothesis, $e_1 = E_1[r]$

(b) pick $E = \mathbf{choose}\ E_1$

case $e = \mathbf{choose}\ v_1$. Pick $E = []$ and $r = e$.

case $e = \langle \sigma, X \to e_1, \Theta \rangle e_2$, where e_1 is not a value.

(a) by induction hypothesis, $e_1 = E_1[r]$

(b) pick $E = \langle \sigma, X \to E_1, \Theta \rangle e_2$

case $e = \langle \sigma \rangle e_2$. Pick $E = []$ and $r = e$.

case $e = \mathbf{dia}\ f$ is a value.

2. case $f = [\Theta_1, e_1]$. Trivially true.

case $f = \mathbf{let\ dia}\ x = e_1\ \mathbf{in}\ f_1$, where e_1 is not a value.

(a) by first induction hypothesis, $e_1 = E_1[r]$

(b) pick $F = \mathbf{let\ dia}\ x = E_1\ \mathbf{in}\ f_1$

case $f = \mathbf{let\ dia}\ x = \mathbf{dia}\ f_1\ \mathbf{in}\ f_2$, where f_1 is not $[\Theta, e]$.

(a) by second induction hypothesis, either $f_1 = F_1[r]$, or $f_1 = F_1[c]$

(b) pick $F = \mathbf{let\ dia}\ x = \mathbf{dia}\ F_1\ \mathbf{in}\ f_2$, and either second or third statement holds

case $f = \mathbf{let\ dia}\ x = \mathbf{dia}\ [e]\ \mathbf{in}\ f_1$, where e is not a value.

(a) by first induction hypothesis, $e = E_1[r]$

(b) pick $F = \mathbf{let\ dia}\ x = \mathbf{dia}\ [E_1]\ \mathbf{in}\ f_1$, and the second statement holds.

case $f = \mathbf{let\ dia}\ x = \mathbf{dia}\ [v]\ \mathbf{in}\ f_1$. Pick $F = [\ ]$ and $c = f$.

case $f = \mathbf{let\ dia}\ x = \mathbf{dia}\ [\langle \sigma, X \to e_1, \Theta \rangle, e_2]\ \mathbf{in}\ f_2$.

(a) by induction hypothesis, $e_1 = E_1[r]$

(b) pick $F = \mathbf{let\ dia}\ x = \mathbf{dia}\ [\langle \sigma, X \to E_1, \Theta \rangle, e_2]\ \mathbf{in}\ f_2$, and the second statement holds.

case $f = \mathbf{let\ dia}\ x = \mathbf{dia}\ [\sigma, e]\ \mathbf{in}\ f_2$. This is a redex, so pick $F = [\ ]$, and $c = f$

■

Lemma 114 (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:

1. (a) by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$
   (b) if e is not a value, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$
   (c) by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \xrightarrow{\sigma} \Sigma', e_1$
   (d) by evaluation rules, $\Sigma, E[r] \overset{\sigma}{\longmapsto} \Sigma', E[e_1]$
   (e) pick $e' = E[e_1]$
2. (a) by unique decomposition lemma, f is either equal to $[\Theta, e]$, or there exists unique F and r such that $f = F[r]$, or there exists unique F and c such that $f = F[c]$
   (b) in the second case, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$
   (c) by progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \xrightarrow{\sigma} \Sigma', e_1$
   (d) pick $f' = F[e_1]$
   (e) in the third case, then by replacement lemma, there exists a type B and support C_1 such that $\Sigma; \cdot \vdash c \div_{C_1} B\,[C]$
   (f) by progress for $\longrightarrow$, there exists a phrase f_1, a context Σ' and a substitution σ', such that $(\Sigma, \sigma), c \longmapsto (\Sigma', \sigma'), f_1$
   (g) pick $f' = F[f_1]$

■

Lemma 115 (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 prove only the first statement of the lemma. The second one is trivial, as there are no primitive closure constructors that introduce fresh names.

1. by evaluation rules, there exists E and r such that $e = E[r]$

2. by unique decomposition lemma, these are unique.

3. by evaluation rules, $\Sigma, r \longrightarrow \Sigma_i, e_i'$ and $e_i = E[e_i']$.

4. The proof proceeds by analysis of possible reduction cases:

   (a) If $r = (\lambda x.\ e)\ v$, or $r =$ **let box** $u =$ **box** e_1 **in** e_2, or $r = \langle\sigma\rangle e$, the reducts are unique, i.e. $e_1' = e_2'$, and thus $e_1 = e_2$, so the identity permutation satisfies the conditions.

   (b) If $r =$ **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.

■