Functional Programming with Names and Necessity

Aleksandar Nanevski

June 9, 2004

Contents

Introduction

It is becoming increasingly important today to execute programs in very complex run-time environments. Modern programs are often required to run in parallel, be mobile, use distributed data owned by different authorities, accommodate dynamically to changing run-time conditions. Moreover, as the run-time environments are becoming more complex, so is the programming for these environments.

When approaching complex programming problems, a language-enforced programming discipline is crucial, and a natural way to enforce this discipline is through the type mechanism of functional languages. Types express assumptions and guarantees required of expressions, and usually correspond to propositions in some logic. The compiler can mechanically check if the expression matches its specified type, thereby aiding the debugging process.

The type systems of most languages today usually ensure that functions are invoked with matching arguments but, unfortunately, ignore how programs interact with run-time environments. In order to manage the increased complexity of programming, a language-enforced typing discipline that takes environments into account seems like a critical component. Indeed, if types could capture important aspects of run-time environments, then the type system may also ensure that expressions are always executed in matching environments.

What does it mean for an expression and an environment to match? The definition may be given in many different ways, depending on the particular application. However, just as an illustration of the concept, consider the following simple, but concrete example. Assume that an environment consists on a number of allocated, but not necessarily initialized memory locations. An expression interacts with this environment by reading or writing into the locations. One possible definition of matching may, for example, insist that each expression reading from a number of locations is always executed in a state of memory where these locations are actually initialized.

A related issue is whether an expression only depends on the environment in which it executes, or perhaps the execution of the expression may cause a change in the environment. To refer to the previous example, a program that does not interact explicitly with the memory locations will produce the same result irrespectively of the particular values stored into the locations. Such a program is *pure*. If the program reads from a certain location, then changing that locations value may certainly change the result of the program. If the program actually writes into a location, than not only the program depends on the memory environment, but it also *changes* it. It may be beneficial in several ways to make a typing distinction between expressions that are pure, expressions that depend on the environment, and expressions that may change the environment. A pure expression is self-contained. One can easily optimize it and reason about it. If the expression is impure, optimizations and reasoning are much harder, because interactions with unknown environments must be taken into account. The reasoning is made easier if types could restrict the kinds of environments that may be encountered, and also inform about the nature of the interaction.

A natural question then becomes: which logic may capture the properties of run-time environments, and thus may serve as a foundation for type systems with above properties? The proposed answer in this dissertation is: *modal logic*. More specifically, the thesis statement of the dissertation is:

Partial modal logic with names provides an appropriate type theoretic foundation for expressing diverse aspects of the interaction between a functional program and the environment in which this program executes.

Modal logic is designed for reasoning about truth across various – abstract – worlds. A proposition may be true in some world, but not true in some other. The versions of modal logic that will be consider here feature two operators on propositions: $\Box$ (box) and $\Diamond$ (diamond). The operator $\Box$ is a universal quantifier: $\Box A$ is true at the current world iff A is *necessary*, i.e. true at *all* worlds. The operator $\Diamond$ is an existential quantifier: $\Diamond A$ is true at the current world iff A is *possible*, i.e. true in at least *some* world.

For the application to programming languages, we may assume that, intuitively, the worlds from modal logic stand for the run-time environments in which the programs execute. Then, according to the proofs-as-programs paradigm of type theory, deriving truth of a proposition A in a particular world, computationally corresponds to producing a value of type A in a particular run-time environment.

Partial modal logic further introduces an additional condition C that may or my not be satisfied by any given world. Then instead of two modal operators $\Box$ and $\Diamond$, we consider two *indexed families* of operators $\Box_C$ and $\Diamond_C$ which correspond to *bounded* universal and bounded existential quantification over worlds, respectively. The proposition $\Box_C A$ is true at the current world iff A is true at every world in which C holds. The proposition $\Diamond_C A$ is true at the current world if there exists a world in which both C holds and A is true.

Computationally, the condition C represents properties of interests that the run-time environment must satisfy in order for the considered expression to be evaluated. In the previously mentioned example with memory reads and writes, C may be a list of currently initialized memory locations. The type system may ensure that expressions reading from locations listed in C are always executed in environments in which locations from C are initialized.

The computational interpretation of the modal type $\Box_C A$ parallels its logical meaning: $\Box_C A$ classifies expressions of type A that may execute in *any* environment satisfying the condition C. The results of the execution may differ depending on the particular environment, but it is important that the environment is not changed as result of the execution. In our example with memory, $\Box_C A$ will classify expressions that do not write into any locations, but may read from locations in C, before computing a value of type A.

The interpretation of the modal type $\Diamond_C A$ is dual: $\Diamond_C A$ classifies expressions that may change the current environment (and the condition C captures the aspects that are subject to change) before producing a value of type A in the changed environment. Such expressions correspond to bounded existential quantification. Indeed, they are the witness that there *exists* an environment (i.e. the one obtained after the change has been carried out) in which a value of type A can be computed. In the example with memory, $\Diamond_C A$ will classify expressions that may first write into the memory locations C before computing a value of type A in the changed state.

Names are objects that are used to formally represent the partiality condition C. In the example with memory, each memory location is associated with a *name* which uniquely identifies this location. The condition C is a set of names, representing the set of locations that are currently initialized. Names may be dynamically allocated and introduced into the computation.

The idea to use types to differentiate pure from effectful expressions has certainly been studied before. Here we only mention the most popular approaches: *type-and-effect systems* [GL86, LG88, Wad98, JG91, TJ94, TT97], and *monads* [Mog91, Wad92, Wad95, Wad98]. In modal logic, however, the emphasis is not on the effects themselves, but is rather on the environments (as the reader have undoubtedly already noticed). For example, in the framework of effect systems or monads, an expression may be described as "causing the effects of reading from memory locations C". In modal logic, the same expression will be characterized as being "executable in any state of memory in which the locations C are initialized".

This switch of emphasis will allow modal systems that may express interactions between programs and environments that are much more diverse than just effects. In fact, the notion of a generic monad gives rise to a particularly simple version of modal logic, called *lax logic* [FM97], and thus monads may be seen as a special case of the modal approach. Of course, there are many other modal logics, which may potentially capture many different aspects of programs and environments. For example, Chapter 3 studies in more detail a version of modal logic suitable for application to staged computation and meta programming, where programs may be generated, compiled and even inspected at run time.

The rest of this sections describes the organization of the dissertation and the contributions of each particular chapter.

Organization and contributions

Chapter 1: Constructive modal logic

The purpose of this chapter is to establish the main concepts that we operate with in the rest of the document. We use the methodology of Martin-Löf [ML96] to clearly separate between the notions of proposition and judgments, and then develop a natural deduction for a particular version of modal logic. The modal logic in question is called Constructive S4 (CS4), and it will be a basis for all the considerations in the following chapters. In addition to the usual connectives of propositional logic, CS4 contains the modal propositional operators $\Box$ and $\Diamond$ which express *universal* and *existential* quantification over possible worlds.

The proof term assignment for the developed natural deduction defines a modal extension of the λ-calculus, and provides the computational context for the modal logic CS4. The modal λ-calculus is characterized by the new term constructors **box** and **let box** (which correspond to the inference rules for the operator $\Box$) and **dia** and **let dia** (which correspond to the inference rules for the operator $\Diamond$).

The chapter concludes with the formulation of the relevant expression substitutions, and the corresponding substitution principles in the setting of both the natural deduction and the modal λ-calculus.

The development in this chapter closely follows the work of Pfenning and Davies [PD01]. The chapter does not contain any novel contributions.

Chapter 2: Partial modal logic

This chapter develops partial modal logic CS4, as an extension of ordinary CS4 from Chapter 1. The main idea is to introduce a condition C that serves to characterize arbitrary aspect of the possible worlds that may be of relevance for the eventual application. The condition C is called *support*. The basics of the logic are developed with the support C kept abstract, so that the chapter is rather general. Eventually, C is defined as a set of *names* (to be described below), but many other definitions seem plausible.

The introduction of supports leads to the definition of modal operators $\Box_C$ and $\Diamond_C$, which are indexed by the support C. The indexed modal operators correspond to *bounded* quantification over possible worlds. For example, $\Box_C A$ will intuitively be true at the current world iff A is true at *all* possible worlds in which C holds. Dually, $\Diamond_C A$ will be true at the current world iff there *exists* a world in which both C holds and A is true.

The extensions of the logic will also influence the corresponding λ-calculus. In order to preserve the completeness, we will add new term constructors. But most importantly, the definition of supports will lead to a definition of a new and interesting operation of *modal substitution*. Unlike the ordinary substitution, which treats the substituting terms parametrically, modal substitution allows that the term may be modified before it is substituted in. It is important that a different modification may be specified for each substituting occurrence. This process of modification is called *reflection*, and may be defined in many ways, depending on the definition of supports.

This chapter also introduces *names*, which provide a particular way to specify supports. Each name is associated with some proposition A, and it serves as a placeholder for proofs of A (or for that matter, any proofs of any other property related to A that may be of interest for the application). As already mentioned, support C may then be viewed as a *set* of names. In this case, the condition expressed by C is the *conjunction* of the propositions associated with each name in C. The process of reflection is then defined as an *explicit substitution* of the names in C. The proof-term assignment obtained for the partial modal logic with names gives rise to an extension of a λ-calculus, which we call a modal ν-calculus.

The chapter concludes with the proofs of the main principles associated with ordinary, modal and explicit substitutions. All the work presented in this chapter is original.

Chapter 3: Staged computation and meta programming

In staged computation and meta programming, we are concerned in writing code that generates other code. Frequently, the generated code may be seen as source code (i.e. a syntactic entity), and the operations of interest include not only generating but also compiling and inspecting source code.

The type safety for meta programming applications has to guarantee that well-typed meta programs only generate well-typed source code. One of the most persistent challenges related to the types in meta programming has been in devising a type system that may differentiate between source code which is *closed* (i.e. it does not depend on free variables, and may therefore be compiled and executed at run time), and source code which is *open*, i.e. may depend on free variables.

It turns out that the $\Box$-fragment of the modal ν-calculus from Chapter 2 directly extends to a meta programming calculus with types for closed and open source code. The type $\Box_C A$ classifies source code of type A which may depend on free variables (i.e. names) listed in the set C. When the set C is empty, then $\Box A$ classifies closed source code. In this chapter, we also define the notion of polymorphism in supports, so that we can write programs that manipulate source code of different or even unknown support. The chapter also presents some initial development towards extending the calculus with features for pattern matching against source code.

From the technical standpoint, the contributions of the chapter involve the development of the logical relations for the $\Box$-fragment of the modal ν-calculus, as well as proofs of the appropriate progress and type preservation theorems. All the work presented in this chapter is original.

Chapter 4: Modal theory of effects

In this chapter we develop a general modal calculus in which types can distinguish between two kinds of effects: effects that are *persistent*, and effects that are *benign*. The execution of a persistent effects inflicts a change upon the run-time environment, while the *benign* effects only depend on the environment, but do not change it. A typical persistent effect is writing into a memory location, while typical benign effects are memory reads or control flow effects. The derived type system is able to differentiate between values (which are ascribed non-modal types), computations with benign effects (ascribed the indexed modal type $\Box_C A$) and computations with persistent effects (ascribed the type $\Diamond_C A$). This development is an original contribution.

The programming style enforced by this type system serializes the computations with persistent effects. The persistent effects must be totally ordered, simply because their execution changes the run-time environment, so any well-defined semantics has to fix this order. Such a requirement, however, is not imposed on benign effects.

The idea to use types to differentiate between values and (possibly effectful) computations have certainly been considered before. The most prominent representative of this line of research are monads and the monadic meta language [Mog91, Wad92, Wad95, Wad98]. The notion of a generic monadic type operator $\bigcirc$ gives rise to *lax logic* [FM97], which is a simple variant of modal logic.

It is interesting that lax logic may be embedded into the constructive modal logic S4, as discovered by Pfenning and Davies [PD01]. In this chapter, we present both the lax logic and its embedding. While we adopt the approach of [PD01] in the description of lax logic, the embedding itself is presented in a novel way. Rather than insisting on the formal syntactic particulars of the embedding, we focus on its more illustrative semantical importance, which is in identifying the concepts of truth and necessity. This identification of truth and necessity may be formally achieved by adjoining a single axiom schema $A \rightarrow \Box A$ to modal logic CS4. In this case, the modal operator $\Diamond$ becomes the monadic operator $\bigcirc$ from lax logic.

The development of the chapter proceeds by performing a similar modification to partial modal logic. When truth and necessity are identified in partial modal logic (or, equivalently, if partial modal logic is extended with the axiom schema $A \rightarrow \Box A$), we obtain a general type system for benign and persistent effects described at the beginning. This observation is also an original contribution.

The general calculus may be uniformly instantiated to treat various different effects, and we do so to obtain novel calculi for memory reads and writes and calculi for control effects like exceptions, catch-and-throw and composable continuations. As mentioned before, an important characteristics of these calculi is that benign effects need not be explicitly serialized. This is an improvement when compared to the monadic meta language, where programs must explicitly specify a total ordering on all effects. In the modal calculus, such total ordering in imposed only on persistent effects. The modal formulation of benign effects may also potentially improve the efficiency of the computations, when compared to the monadic treatment of the same effects.

It is interesting that the $\Box$-fragment of the calculus for memory implements a type-safe version of *dynamic binding*. In this calculus, computations that read from memory are ascribed a universal bounded type $\Box_C A$. The construct for dynamic binding binds values to memory locations, and thus specifies an environment in which a computation of type $\Box_C A$ may be executed. In this sense, dynamic binding logically corresponds to instantiation of the bounded universal quantifier $\Box_C$. Discovering the logic behind dynamic binding was a long-standing problem in functional programming.

Chapter 1

Constructive modal logic

1.1 Natural deduction

1.1.1 Judgments and propositions

A modality is a logical operator that qualifies assertions about the truth of propositions. For example, given a certain proposition A, we may consider if A is true or false, but may also be interested if A is *necessarily* true, or *possibly* true, *will be* true at the next moment in time, *is believed to be* true, and so on.

The assertions expressed by modalities are customarily given formal semantics using the approach of *Kripke frames* [Kri63]. A Kripke frame is a relational structure $(\mathcal{W}, R)$, consisting of a set of *possible worlds* $\mathcal{W}$, and a relation $R \subseteq \mathcal{W} \times \mathcal{W}$ of *accessibility*. Then, a modally qualified proposition expresses an assertion about truth across accessible worlds. The nature of the assertion is determined by the nature of the accessibility relation.

We illustrate the concept of Kripke frames using a particularly simple example of temporal modal logic, which is a logic for reasoning about truth in *subsequent* moments in time. The appropriate Kripke frame for this logic defines the possible worlds $\mathcal{W}$ as moments in time. The accessibility relation R is discrete and total, determining the temporal relation between worlds. We have $(w, w') \in R$ if and only if w is a moment occuring sometime before w'. Because R is discrete, for each moment w there is a w' that can be chosen as a *subsequent* moment. Then we can define a modality $\bigcirc$ as an operator on propositions expressing truth at the subsequent time moment. More precisely, we say that $\bigcirc A$ is *true at time moment* w if and only if A is *true at the subsequent time moment* w'.

Some other operators frequently considered in modal logic are the operator $\Box$ of necessity and the operator $\Diamond$ of possibility. The two operators express universal and existential quantification over accessible worlds, respectively. As an illustration, in the temporal logic described above, we say that $\Box A$ is true at some time moment w if and only if A is true at *all* time moments in the future of w. Dually, $\Diamond A$ is true at w if and only if A is true at *some* time moment in the future of w.

In this section, we review the results of Pfenning and Davies from [PD01] and consider modal logic from intuitionistic and type theoretic perspective, rather than from the perspective of Kripke frames and possible worlds. The intuitionistic approach puts special emphasis on the constructive import of propositions: A will be considered true, if and only if we can *construct and exhibit* an evidence of it. In our formulation, we follow the methodology of Martin-Löf [ML96] to clearly separate the notions of judgments and propositions. Propositions are logical objects which encode statements of interest for the formal consideration. Judgments represent properties of propositions that are subject to proof.

For example, we can judge if a certain proposition A is *well formed* or not, and we can formulate a judgment

$$A\ prop$$

defining what counts as a proof of well-formedness. If we assume that our logic contains an operator $\wedge$ for conjunction, then a conjunction of two propositions A and B is a well-formed proposition whenever both A and B are well-formed. This (rather self-evident) fact can be expressed as an *inference rule* of the judgment $A\ prop$ as follows

$$\frac{A\ prop \qquad B\ prop}{A \wedge B\ prop}$$

The inference rule is oriented in a top-down manner: the judgments above the line are *premises*, and the judgment below the line is a *conclusion* that may be inferred after the premises have been judged satisfied (i.e. witnessed by a proof). In this sense, a proof that $A \wedge B\ prop$ consists of the proofs that $A\ prop$ and $B\ prop$.

A completely separate judgment has to be used to determine when a proposition A is true, and what constitutes a proof, i.e. evidence for the truth of A. Appropriately enough, we call this judgment

$$A\ true$$

and we implicitly assume that $A\ prop$ is satisfied before we can judge if $A\ true$. In intuitionistic logic, we have an evidence of $A \wedge B$ if and only if we have an evidence of each of the two propositions. We can express the if-then direction of this fact using the *introduction* rule

$$\frac{A\ true \qquad B\ true}{A \wedge B\ true}$$

and the only-if direction is encoded using the two *elimination* rules

$$\frac{A \wedge B\ true}{A\ true} \qquad \frac{A \wedge B\ true}{B\ true}$$

The introduction rule defines when it is justified to conclude that a conjunction of two propositions is true. The rule is named "introduction" because it allows us to *introduce* the $\wedge$ operator into the proposition $A \wedge B$. The elimination rules define how to use a conjunction once it has been proved. In particular, we can always "eliminate" the $\wedge$ operator from $A \wedge B$, and obtain A in isolation from B, or vice versa.

Of course, the introduction and elimination rules for a logical operator cannot be completely arbitrary, but must satisfy certain coherence conditions which ensure that the rules match. For example, the elimination rules should not be *too strong* and allow us to infer unjustified conclusions. We can make a conclusion from the elimination rule only if we have enough evidence for the premises. This property is known as *local soundness*. It is witnessed by *local reduction* which constructs evidence for the conclusion of an elimination rule out of evidence of the premises. The local reductions witnessing the local soundness of the elimination rules for conjunction are stated in the following form.

$$\cfrac{\cfrac{A\ true \quad B\ true}{A \wedge B\ true}}{A\ true} \quad \Longrightarrow_R \quad A\ true$$

and

$$\cfrac{\cfrac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}}{B\ \mathit{true}} \quad \Longrightarrow_R \quad B\ \mathit{true}$$

The first local reduction shows that the conclusion $A\ \mathit{true}$ obtained after eliminating $A \wedge B\ \mathit{true}$ could have already been obtained as a first premise of the rule that introduced $A \wedge B\ \mathit{true}$. Therefore, the elimination rule is not too strong, because we can only use it to establish something we already had. The local reduction shows how the proof could have been derived without the detour of introducing and then eliminating the conjunction. This is why it is called "reduction"; it establishes simpler evidence for the conclusion obtained after conjunction elimination. The other local reduction is completely symmetric, except that it uses the second elimination rule for conjunction.

The elimination rules must not be *too weak* either. We should be able to use an elimination rule in such a way that its premises can be recovered. This property is known as *local completeness*. It is witnessed by *local expansion*, which applies the elimination rules in order to obtain enough knowledge to reconstruct the original judgment. It is called "expansion" because it obtains a more complex evidence for the original judgment. In case of conjunction, the local expansion takes the following form.

$$A \wedge B\ \mathit{true} \quad \Longrightarrow_E \quad \cfrac{\cfrac{A \wedge B\ \mathit{true}}{A\ \mathit{true}} \quad \cfrac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}}{A \wedge B\ \mathit{true}}$$

As shown above, the local expansion eliminates $A \wedge B\ \mathit{true}$ to obtain $A\ \mathit{true}$ and $B\ \mathit{true}$. The two are then combined to reintroduce $A \wedge B\ \mathit{true}$.

1.1.2 Hypothetical judgments and implication

A further primitive notion that we need is that of a *hypothetical judgment*, i.e. a judgment which is made under hypotheses, or assumptions. Hypothetical judgments are needed in order to formalize the concept of implication. We would like to define the implication $A \to B$ to be true if and only if $B\ \mathit{true}$ can be proved whenever $A\ \mathit{true}$ can. But in order to formally state this causal dependence between A and B, we need to define what it means to judge $B\ \mathit{true}$ under an assumption that $A\ \mathit{true}$.

The general form of a hypothetical judgment is written as

$$J_1, \ldots J_n \vdash J$$

which expresses that J can be proved under the hypotheses $J_1, \ldots, J_n$. We also refer to $J_1, \ldots, J_n$ as *antecedents* and J as the *succedent* of the hypothetical judgment.

The first specific hypothetical judgment that we consider in this section limits $J_1, \ldots, J_n, J$ to be instances of $A\ \mathit{true}$, and therefore has the form

$$A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \vdash A\ \mathit{true}$$

The collection $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$ is called a *context* of hypothesis. We use Γ and variants to range over contexts, and will usually write the hypothetical judgment in an abbreviated form

$$\Gamma \vdash A\ \mathit{true}.$$

When defining a new judgment, we need to state what counts as an evidence, or proof for it. In the particular case of the hypothetical judgment $\Gamma \vdash A\ \mathit{true}$, we need to define a notion of hypothetical proof. What does it mean to derive $A\ \mathit{true}$ under assumptions Γ? In a hypothetical proof of $A\ \mathit{true}$ under assumptions $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$, we can use the hypotheses as if we knew them. Once a derivation of $A_i\ \mathit{true}$ is given (for some A_i), we can *substitute* it for the uses of the assumption $A_i\ \mathit{true}$ in the hypothetical proof, to obtain a judgment and a proof that no longer depend on $A_i\ \mathit{true}$. In this sense, a proof of $\Gamma \vdash A\ \mathit{true}$ *prescribes* how a proof $A\ \mathit{true}$ can be constructed, once proofs of $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$ are given. The emphasis in this construction is on the operation of *substitution*. When deriving $A\ \mathit{true}$, the proofs of $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$ may only be used *as given*, without any opportunity for inspection or modification. Because of this particular property, we say that the hypothetical judgment is *parametric* in its assumptions.

The nature of the hypothetical judgment and the dependence between antecedents and succedent is usually stated in the form of the following substitution principle.

If $\Gamma \vdash A\ \mathit{true}$ and $\Gamma, A\ \mathit{true}, \Gamma' \vdash B\ \mathit{true}$, then $\Gamma, \Gamma' \vdash B\ \mathit{true}$.

The substitution principle implicitly assumes that the proof of $\Gamma \vdash A\ \mathit{true}$ is indeed substituted into the proof of $\Gamma, A\ \mathit{true}, \Gamma' \vdash B\ \mathit{true}$ to obtain a proof of $\Gamma, \Gamma' \vdash B\ \mathit{true}$. Notice that the substitution principle should not be viewed as an inference rule. Rather, it is a meta-theoretic property which we will have to prove once all the inference rules of $\Gamma \vdash A\ \mathit{true}$ are defined.

In addition to the substitution principle, we impose some further structure of the hypothetical judgment. In particular, we require the following structural properties.

1. *Exchange.* If $\Gamma_1, A_1\ \mathit{true}, \Gamma_2, A_2\ \mathit{true} \vdash B\ \mathit{true}$, then $\Gamma_1, A_2\ \mathit{true}, \Gamma_2, A_1\ \mathit{true} \vdash B\ \mathit{true}$.

   This structural property of exchange states that the ordering of hypothesis in the context Γ is irrelevant for the judgment. In other words, we may consider Γ to be a *set*, rather than a list. We immediately put exchange to use in order to abbreviate the statements about our hypothetical judgments.

2. *Weakening.* If $\Gamma \vdash B\ \mathit{true}$ then $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$.

3. *Contraction.* If $\Gamma, A\ \mathit{true}, A\ \mathit{true} \vdash B\ \mathit{true}$, then $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$.

Using the structural properties of exchange and weakening, we can further simplify the substitution principle for the truth judgment, and rephrase it as presented below. It is this form of the substitution principle that we adopt in the rest of the dissertation.

Principle (Substitution)
If $\Gamma \vdash A\ \mathit{true}$ and $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$, then $\Gamma \vdash B\ \mathit{true}$.

The hypothesis rule of the truth judgment formalizes the intuition that assumptions in a hypothetical judgment may be used as if they were known. In particular, under the assumption $A\ \mathit{true}$, we may always conclude $A\ \mathit{true}$. Following the structural property of exchange, the rule ignores the ordering of the hypothesis in the context Γ.

$$\frac{}{\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}}$$

After introducing all the machinery of hypothetical judgments and proofs, we are finally ready to define *implication* $A \to B$ as a new form of propositions, which expresses that $B\ \mathit{true}$ may be derived when $A\ \mathit{true}$ is given. We will frequently say that implication *internalizes* hypothetical truth, because it provides means to reason about hypothetical truth within the ordinary truth judgment.

As a first step in the definition of the new propositional operator, we need to extend the formation judgment A *prop* so that it can treat the new case involving the operator $\rightarrow$. The appropriate formation rule simply states that $A \rightarrow B$ is a well formed proposition, whenever both A and B are.

$$\frac{A\ \mathit{prop} \qquad B\ \mathit{prop}}{A \rightarrow B\ \mathit{prop}}$$

More interesting are the inference rules that extend the truth judgment. Following the methodology of natural deduction that we previously used in the case of conjunction, we provide an introduction and an elimination rule for implication. The introduction rule formally states that $A \rightarrow B\ \mathit{true}$ can be derived if there is a hypothetical proof of $A\ \mathit{true} \vdash B\ \mathit{true}$. The introduction rule therefore exactly serves to define the operator of implication as an internalization of hypothetical judgments.

$$\frac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}{\Gamma \vdash A \rightarrow B\ \mathit{true}}$$

The elimination rule for implication realizes the substitution principle, and provides a way to infer $B\ \mathit{true}$ when both $A \rightarrow B\ \mathit{true}$ and $A\ \mathit{true}$ can be obtained.

$$\frac{\Gamma \vdash A \rightarrow B\ \mathit{true} \qquad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}}$$

The rules are locally sound and complete, and therefore of matching strength. Local reduction is presented below, and is justified by the substitution principle.

$$\frac{\dfrac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}{\Gamma \vdash A \rightarrow B\ \mathit{true}} \qquad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}} \quad \Longrightarrow_R \quad \Gamma \vdash B\ \mathit{true}$$

Indeed, the derivation of $\Gamma \vdash B\ \mathit{true}$ may be obtained by substituting the premise $\Gamma \vdash A\ \mathit{true}$ into the premise $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$, just as claimed by the substitution principle.

The local completeness is witnessed by local expansion.

$$\Gamma \vdash A \rightarrow B\ \mathit{true} \quad \Longrightarrow_E \quad \frac{\dfrac{\Gamma, A\ \mathit{true} \vdash A \rightarrow B\ \mathit{true} \qquad \dfrac{}{\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}}}{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}}}{\Gamma \vdash A \rightarrow B\ \mathit{true}}$$

Local expansion first uses the structural property of *weakening* to modify $\Gamma \vdash A \rightarrow B\ \mathit{true}$ into $\Gamma, A\ \mathit{true} \vdash A \rightarrow B\ \mathit{true}$. Implication elimination is performed on this premise to obtain $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$, before reintroducing implication again and conclude $\Gamma \vdash A \rightarrow B\ \mathit{true}$.

Example 1 The following are example judgments that can be derived in the logic presented so far.

1. $\vdash A \rightarrow A\ \mathit{true}$

2. $\vdash A \rightarrow B \rightarrow A\ \mathit{true}$

3. $\vdash (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C\ \mathit{true}$

1. Derivation of $\vdash A \to A\ \mathit{true}$.

$$\frac{A\ \mathit{true} \vdash A\ \mathit{true}}{\vdash A \to A\ \mathit{true}}$$

2. Derivation of $\vdash A \to B \to A\ \mathit{true}$ first uses the hypothesis rule to infer $A\ \mathit{true}, B\ \mathit{true} \vdash A\ \mathit{true}$, which is then followed by two introductions.

$$\dfrac{\dfrac{A\ \mathit{true}, B\ \mathit{true} \vdash A\ \mathit{true}}{A\ \mathit{true} \vdash B \to A\ \mathit{true}}}{\vdash A \to B \to A\ \mathit{true}}$$

3. Derivation of $\vdash (A \to B \to C) \to (A \to B) \to A \to C\ \mathit{true}$. To reduce clutter, we abbreviate contexts as follows: $\Gamma_1 = \mathcal{A} \to B \to C\ \mathit{true}$, $\Gamma_2 = (\Gamma_1, A \to B\ \mathit{true})$, and $\Gamma = (\Gamma_2, A\ \mathit{true})$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A \to B \to C\ \mathit{true} \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B \to C\ \mathit{true}} \quad \dfrac{\Gamma \vdash A \to B\ \mathit{true} \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash B\ \mathit{true}}}{\Gamma \vdash C\ \mathit{true}}}{\Gamma_2 \vdash A \to C\ \mathit{true}}}{\Gamma_1 \vdash (A \to B) \to A \to C\ \mathit{true}}}{\vdash (A \to B \to C) \to (A \to B) \to A \to C\ \mathit{true}}$$

■

1.1.3 Necessity

In the previous sections we considered two versions of the judgment for truth: the hypothetical version $\Gamma \vdash A\ \mathit{true}$, and the non-hypothetical version $A\ \mathit{true}$. The hypothetical version $\Gamma \vdash A\ \mathit{true}$ extends $A\ \mathit{true}$, in the sense that the later can be recovered as $\cdot \vdash A\ \mathit{true}$ where the context Γ is chosen to be empty. The variant $\cdot \vdash A\ \mathit{true}$ is known as a *categorical judgment*, because it does not depend on any hypotheses. It can be seen as stating a universal fact, which does not rely on external arguments. Categorical judgments are witnessed by *categorical proofs*. A categorical proof is, again, a proof that does not depend on any hypotheses; a proof which is, in some sense, closed.

In this section we isolate the notions of categorical judgment and categorical proof, and consider them in and of themselves, rather than as special cases of hypothetical judgments and proofs. To this end, we introduce the judgment for *necessity*

$$A\ \mathit{nec}$$

defined by the following two clauses.

1. If $\cdot \vdash A\ \mathit{true}$, then $A\ \mathit{nec}$.
2. If $A\ \mathit{nec}$, then $\Gamma \vdash A\ \mathit{true}$.

The two clauses define that $A\ \mathit{nec}$ holds if and only if $\cdot \vdash A\ \mathit{true}$. The clause (1) establishes that if-then direction, and the clause (2) corresponds to the only-if direction. Notice that we allow non-empty Γ in the definitional clause (2) in order to avoid explicit context weakening.

The choice of the name for the necessity judgment is not accidental. As we will soon demonstrate, the consideration of categorical proofs and categorically true propositions very quickly leads to a formulation of *modal logic*. An informal but useful intuition that relates categorical judgments to modal logic is based on the following observation. Each context Γ of a hypothetical truth judgment may be seen as selecting a set of possible worlds in a Kripke-style semantics. The selected worlds are those that satisfy all the hypotheses in Γ. If the proposition A is categorically true, i.e. if $\cdot \vdash A\ true$, then A is true in a generic world about which we know nothing. In other words, A is true in *all* accessible worlds. In this sense, categorical truth corresponds to *universal quantification*, and categorically true propositions are *necessary*. On the other hand, the hypothetical judgment $\Gamma \vdash A\ true$ only provides evidence for the truth of A in the *current* world of reference. We will frequently rely on this intuition to motivate particular design choices in our logic, but we do not pursue further its formal side. The interested reader is refered to the work of Alechina at al. [AMdPR01], which provides a Kripke semantics for a natural deduction related to ours.

As evident from the definition, necessity is a judgment whose meaning is described in terms of truth. Thus, necessity in itself does not introduce anything new, unless we take a step further and extend the truth judgment so that it can depend on necessary hypotheses. Because the order of hypotheses is not important, we separate the context into two parts (separated by semi-colon for visual clarity), and consider a judgment of the following form.

$$B_1\ nec, \ldots, B_m\ nec; A_1\ true, \ldots, A_n\ true \vdash A\ true$$

We use Γ to range over sets of hypotheses of the form $A\ true$, and Δ to range over sets of hypothesis $A\ nec$. We will implicitly assume that both the contexts are subject to the structural properties of weakening, exchange and contraction.

To define what counts as a proof of the new hypothetical judgment, we need to extend the notion of categorical proof. Similar to before, a categorical proof of $\Delta; \Gamma \vdash A\ true$ is a proof obtained without any reference to *truth* hypotheses. However, a categorical proof is *allowed to depend on necessary hypotheses*. This is only natural, because categorical proofs are evidence for necessary propositions, and could therefore be substituted for necessary hypotheses. The following substitution principle formaly states the described reasoning.

Principle (Substitution for necessity)
If $\Delta; \cdot \vdash A\ true$ and $(\Delta, A\ nec); \Gamma \vdash B\ true$ then $\Delta; \Gamma \vdash B\ true$.

Note that the judgment $\Delta; \cdot \vdash A\ true$ in the substitution principle does not depend on true hypotheses. Its proof is categorical, and can therefore be *substituted* for the hypotheses $A\ nec$ to derive $B\ true$. The emphasis here is again on substitution. The proof of A may *not* be modified or inspected in any way before it is used to derive $B\ true$.

Related to the substitution principle for necessity is the rule for necessary hypotheses. The judgment $A\ nec$ is witnessed by a categorical proof of $A\ true$, and a categorical proof can always be viewed as an ordinary proof. Thus, given $A\ nec$, we are justified in deriving $A\ true$, as the following rule for necessary hypotheses states.

$$\frac{}{(\Delta, A\ nec); \Gamma \vdash A\ true}$$

After introducing the concept of necessity, the next step is to internalize it. To that end, we introduce a new unary operator on propositions $\Box$, with the expected formation rule.

$$\frac{A\ prop}{\Box A\ prop}$$

The introduction rule follows the definition of necessity: we can derive $\Box A\ \mathit{true}$ only if there is a derivation of $A\ \mathit{nec}$, i.e. only if there is a categorical derivation of $A\ \mathit{true}$.

$$\frac{\Delta; \cdot \vdash A\ \mathit{true}}{\Delta; \Gamma \vdash \Box A\ \mathit{true}}$$

The elimination rule follows the substitution principle for necessity. Given a derivation of $\Box A\ \mathit{true}$, we know by definition that $\Delta; \cdot \vdash A\ \mathit{true}$. If in addition $(\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{true}$, then by the substitution principle for necessity, we may derive $\Delta; \Gamma \vdash B\ \mathit{true}$.

$$\frac{\Delta; \Gamma \vdash \Box A\ \mathit{true} \qquad (\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{true}}{\Delta; \Gamma \vdash B\ \mathit{true}}$$

This exact reasoning justifies the local reduction and local soundness.

$$\frac{\dfrac{\Delta; \cdot \vdash A\ \mathit{true}}{\Delta; \cdot \vdash \Box A\ \mathit{true}} \quad (\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{true}}{\Delta; \Gamma \vdash B\ \mathit{true}} \quad \Longrightarrow_R \quad \Delta; \Gamma \vdash B\ \mathit{true}$$

The local completeness is established by the local expansion given below.

$$\Delta; \Gamma \vdash \Box A\ \mathit{true} \quad \Longrightarrow_E \quad \frac{\Delta; \Gamma \vdash \Box A\ \mathit{true} \quad \dfrac{(\Delta, A\ \mathit{nec}); \cdot \vdash A\ \mathit{true}}{(\Delta, A\ \mathit{nec}); \Gamma \vdash \Box A\ \mathit{true}}}{\Delta; \Gamma \vdash \Box A\ \mathit{true}}$$

Example 2 The following are valid derivations in the modal logic of necessity presented so far.

1. $\vdash \Box A \to A\ \mathit{true}$
2. $\vdash \Box A \to \Box\Box A\ \mathit{true}$
3. $\vdash \Box(A \to B) \to \Box A \to \Box B\ \mathit{true}$

1. Derivation of $\vdash \Box A \to A\ \mathit{true}$.

$$\frac{\dfrac{\cdot; \Box A\ \mathit{true} \vdash \Box A\ \mathit{true} \quad A\ \mathit{nec}; \Box A\ \mathit{true} \vdash A\ \mathit{true}}{\cdot; \Box A\ \mathit{true} \vdash A\ \mathit{true}}}{\vdash \Box A \to A\ \mathit{true}}$$

2. Derivation of $\vdash \Box A \to \Box\Box A\ \mathit{true}$.

$$\frac{\dfrac{\cdot; \Box A\ \mathit{true} \vdash \Box\Box A\ \mathit{true} \quad \dfrac{\dfrac{A\ \mathit{nec}; \cdot \vdash A\ \mathit{true}}{A\ \mathit{nec}; \cdot \vdash \Box A\ \mathit{true}}}{A\ \mathit{nec}; \Box A\ \mathit{true} \vdash \Box\Box A\ \mathit{true}}}{\cdot; \Box A\ \mathit{true} \vdash \Box\Box A\ \mathit{true}}}{\vdash \Box A \to \Box\Box A\ \mathit{true}}$$

3. Derivation of $\vdash \Box(A \to B) \to \Box A \to \Box B\ \mathit{true}$. To reduce clutter, we introduce the following abbreviations: $\Gamma_1 = \Box(A \to B)\ \mathit{true}$, $\Gamma = \Gamma_1, \Box A\ \mathit{true}$, and $\Delta_1 = (A \to B)\ \mathit{nec}$, $\Delta = \Delta_1, A\ \mathit{nec}$.

$$\dfrac{\dfrac{\dfrac{\cdot;\Gamma \vdash \Box(A \to B)\ \mathit{true} \qquad \dfrac{\Delta_1;\Gamma \vdash \Box A\ \mathit{true} \qquad \dfrac{\dfrac{\Delta;\cdot \vdash A \to B\ \mathit{true} \quad \Delta;\cdot \vdash A\ \mathit{true}}{\Delta;\cdot \vdash B\ \mathit{true}}}{\Delta;\Gamma \vdash \Box B\ \mathit{true}}}{\Delta_1;\Gamma \vdash \Box B\ \mathit{true}}}{\cdot;\Gamma \vdash \Box B\ \mathit{true}}}{\cdot;\Gamma_1 \vdash \Box A \to \Box B\ \mathit{true}}}{\vdash \Box(A \to B) \to \Box A \to \Box B\ \mathit{true}}$$

■

1.1.4 Possibility

In modal logic, a proposition is necessarily true if it is true in *all* the accessible worlds. A dual concept is that of *possible truth*. We say that A is *possible* if there *exists* an accessible world in which A is true. The formulation of possible truth in classical modal logic is usually in terms of necessity, simply because in classical logic existential quantification may be expressed in terms of universal quantification. But since we are interested in a *constructive* variant of modal logic, this approach is not available — possibility should be defined in and of itself. Furthermore, we would like to analyze possibility without actually referring to particular worlds within the formal system, and without describing the totality of worlds and the accessibility relation on it.

As discovered by Pfenning and Davies [PD01], this kind of a formulation can be achieved if one adopts a judgmental approach and considers how the knowledge that A is possibly true can be used to derive new facts. If A is possibly true, than there exists a world about which we know nothing, except that A is true at that world. Therefore, if we assume that A is true (but nothing else), and then conclude that B is possible, then B must be possible. Notice that starting from the possibility of A, we can only make conclusions about the possibility of B, but not about the truth of B. To initially establish that A is possible, we simply need to show that A is true. To formalize this reasoning, we introduce a new judgment $A\ \mathit{poss}$ to witness the possibility of A, and immediately consider its hypothetical variant

$$\Delta; \Gamma \vdash A\ \mathit{poss}$$

where Δ and Γ abbreviate necessary and true assumptions, respectively. The possibility judgment is defined by the following two clauses.

1. If $\Delta; \Gamma \vdash A\ \mathit{true}$, then $\Delta; \Gamma \vdash A\ \mathit{poss}$.

2. If $\Delta; \Gamma \vdash A\ \mathit{poss}$ and $\Delta; A\ \mathit{true} \vdash B\ \mathit{poss}$, then $\Delta; \Gamma \vdash B\ \mathit{poss}$.

Note that the definitional clause (2) makes the necessity assumptions Δ available for deriving $B\ \mathit{poss}$, but removes the truth assumptions Γ. This is because the context Δ stands for propositions that are true in all possible worlds, while Γ stands for propositions that are true only in the current world. Therefore, if A is possible in some world, then we may assume of that world that it validates

Δ, but not Γ. The definitional clause (2) takes form of a substitution principle, and establishes the hypothetical nature of the judgment for possibility with respect to truth hypotheses. On the other hand, the hypothetical character of possibility with respect to necessity hypotheses is described by a version of the substitution principle for necessity.

If $\Delta; \cdot \vdash A\ \mathit{true}$ and $(\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{poss}$, then $\Delta; \Gamma \vdash B\ \mathit{poss}$.

Next we internalize possibility as a propositional operator $\Diamond$, with the obvious formation rule.

$$\frac{A\ \mathit{prop}}{\Diamond A\ \mathit{prop}}$$

The introduction rule for $\Diamond$ simply encodes the fact that $\Diamond$ internalizes possibility into the truth judgment. The elimination rule for $\Diamond$ follows the definitional clause (2), except that instead of the assumption $\Delta; \Gamma \vdash A\ \mathit{poss}$, it uses the internalized variant $\Delta; \Gamma \vdash \Diamond A\ \mathit{true}$.

$$\frac{\Delta; \Gamma \vdash A\ \mathit{poss}}{\Delta; \Gamma \vdash \Diamond A\ \mathit{true}} \qquad \frac{\Delta; \Gamma \vdash \Diamond A\ \mathit{true} \qquad \Delta; A\ \mathit{true} \vdash B\ \mathit{poss}}{\Delta; \Gamma \vdash B\ \mathit{poss}}$$

We also need an inference rule in order to realize the definitional clause (1). This rule takes the form of a judgmental coercion from possibility into truth.

$$\frac{\Delta; \Gamma \vdash A\ \mathit{true}}{\Delta; \Gamma \vdash A\ \mathit{poss}}$$

It is easy to see that the presented inference rules are locally sound and complete. Local soundness is witnessed by the local reduction below.

$$\frac{\dfrac{\Delta; \Gamma \vdash A\ \mathit{poss}}{\Delta; \Gamma \vdash \Diamond A\ \mathit{true}} \quad \Delta; A\ \mathit{true} \vdash B\ \mathit{poss}}{\Delta; \Gamma \vdash B\ \mathit{poss}} \quad \Longrightarrow_R \quad \Delta; \Gamma \vdash B\ \mathit{poss}$$

This local reduction is justified on the grounds of the definitional clause (2). Indeed, given the premises $\Delta; \Gamma \vdash A\ \mathit{poss}$ and $\Delta; A\ \mathit{true} \vdash B\ \mathit{poss}$, the clause (2) leads to the reduct $\Delta; \Gamma \vdash B\ \mathit{poss}$.

Local completeness is witnessed by the local expansion, which itself relies on the judgmental coercion from truth to possibility in order to derive $\Delta; A\ \mathit{true} \vdash A\ \mathit{poss}$.

$$\Delta; \Gamma \vdash \Diamond A\ \mathit{true} \quad \Longrightarrow_E \quad \frac{\dfrac{\Delta; \Gamma \vdash \Diamond A\ \mathit{true} \quad \dfrac{\Delta; A\ \mathit{true} \vdash A\ \mathit{true}}{\Delta; A\ \mathit{true} \vdash A\ \mathit{poss}}}{\Delta; \Gamma \vdash A\ \mathit{poss}}}{\Delta; \Gamma \vdash \Diamond A\ \mathit{true}}$$

We need a yet further rule to realize the substitution principle for necessary hypotheses within the judgment for possibility.

$$\frac{\Delta; \Gamma \vdash \Box A\ \mathit{true} \qquad (\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{poss}}{\Delta; \Gamma \vdash B\ \mathit{poss}}$$

As explained in [PD01], without this rule, the logic will not posses the strict subformula property. For example, a proof of the judgment $\cdot; \Box(A \rightarrow B)\ \mathit{true}, \Diamond A\ \mathit{true} \vdash B\ \mathit{poss}$, will first have to make a detour and establish a more complicated fact $\cdot; \Box(A \to B)\ \mathit{true}, \Diamond A\ \mathit{true} \vdash \Diamond B\ \mathit{true}$, before eliminating $\Diamond B\ \mathit{true}$ to obtain $B\ \mathit{poss}$. The new rule is sound, as witnessed by the following local reduction, justified on the grounds of the substitution principle for necessary hypotheses.

$$\dfrac{\dfrac{\Delta; \cdot \vdash A\ \mathit{true}}{\Delta; \Gamma \vdash \Box A\ \mathit{true}} \quad (\Delta, A\ \mathit{nec}); \Gamma \vdash B\ \mathit{poss}}{\Delta; \Gamma \vdash B\ \mathit{poss}} \quad \Longrightarrow_R \quad \Delta; \Gamma \vdash B\ \mathit{poss}$$

Example 3 The following are valid derivations in the presented modal logic.

1. $\vdash A \to \Diamond A\ \mathit{true}$
2. $\vdash \Diamond\Diamond A \to \Diamond A\ \mathit{true}$
3. $\vdash \Box(A \to B) \to \Diamond A \to \Diamond B\ \mathit{true}$

1. Derivation of $\vdash A \to \Diamond A\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\cdot; A\ \mathit{true} \vdash A\ \mathit{true}}{\cdot; A\ \mathit{true} \vdash A\ \mathit{poss}}}{\cdot; A\ \mathit{true} \vdash \Diamond A\ \mathit{true}}}{\vdash A \to \Diamond A\ \mathit{true}}$$

2. Derivation of $\vdash \Diamond\Diamond A \to \Diamond A\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\cdot; \Diamond\Diamond A\ \mathit{true} \vdash \Diamond\Diamond A\ \mathit{true} \quad \dfrac{\cdot; \Diamond A\ \mathit{true} \vdash \Diamond A\ \mathit{true} \quad \dfrac{\cdot; A\ \mathit{true} \vdash A\ \mathit{true}}{\cdot; A\ \mathit{true} \vdash A\ \mathit{poss}}}{\cdot; \Diamond A\ \mathit{true} \vdash A\ \mathit{poss}}}{\cdot; \Diamond\Diamond A\ \mathit{true} \vdash A\ \mathit{poss}}}{\cdot; \Diamond\Diamond A\ \mathit{true} \vdash \Diamond A\ \mathit{true}}}{\vdash \Diamond\Diamond A \to \Diamond A\ \mathit{true}}$$

3. Derivation of $\vdash \Box(A \to B) \to \Diamond A \to \Diamond B\ \mathit{true}$. To reduce clutter, we explicitly name the contexts $\Gamma_1 = \Box(A \to B)\ \mathit{true}$, $\Gamma_2 = \Gamma_1, \Diamond A\ \mathit{true}$, $\Gamma = A\ \mathit{true}$, and $\Delta = (A \to B)\ \mathit{nec}$.

$$\dfrac{\dfrac{\cdot; \Gamma_2 \vdash \Box(A \to B)\ \mathit{true} \quad \dfrac{\dfrac{\Delta; \Gamma_2 \vdash \Diamond A\ \mathit{true} \quad \dfrac{\dfrac{\Delta; \Gamma \vdash A \to B\ \mathit{true} \quad \Delta; \Gamma \vdash A\ \mathit{true}}{\Delta; \Gamma \vdash B\ \mathit{true}}}{\Delta; \Gamma \vdash B\ \mathit{poss}}}{\Delta; \Gamma_2 \vdash B\ \mathit{poss}}}{\Delta; \Gamma_2 \vdash \Diamond B\ \mathit{true}}}{\dfrac{\cdot; \Gamma_2 \vdash \Diamond B\ \mathit{true}}{\cdot; \Gamma_1 \vdash \Diamond A \to \Diamond B\ \mathit{true}}}}{\vdash \Box(A \to B) \to \Diamond A \to \Diamond B\ \mathit{true}}$$

■

Examples 2 and 3 between themselves list six propositions whose truth is derivable in our logic. It is of particular interests here to emphasize the connection between two pairs of dual propositions

1. $\Box A \to A$ and $A \to \Diamond A$
2. $\Box A \to \Box\Box A$ and $\Diamond\Diamond A \to \Diamond A$

In classical modal logic these pairs correspond to particular properties of the accessibility relation in the possible world semantics. The pair (1), for example, requires that the accessibility relation between worlds is *reflexive*. The pair (2) requires that the accessibility relation is *transitive*. The classical modal logic satisfying these two requirements is known under the name of S4. In analogy with this notational convention, we call the logic presented here Constructive S4, or simply CS4[1]. A formulation of many other intuitionistic modal logics (including a yet another intuitionistic version of S4) can be found in the PhD dissertation of Alex Simpson [Sim94].

1.1.5 Summary of the system

We now summarize the formal system of modal logic with implication, necessity and possibility, as described in the previous sections.

$$\begin{array}{lrcl} \textit{Propositions} & A, B & ::= & P \mid A \to B \mid \Box A \mid \Diamond A \\ \textit{True hypothesis} & \Gamma & ::= & \cdot \mid \Gamma, A\ \textit{true} \\ \textit{Necessary hypothesis} & \Delta & ::= & \cdot \mid \Delta, A\ \textit{nec} \end{array}$$

The logic consists of three basic judgments *A true*, *A nec* and *A poss*, which are used as hypotheses in two different hypothetical judgments $\Delta; \Gamma \vdash A\ \textit{true}$ and $\Delta; \Gamma \vdash A\ \textit{poss}$. The rules of the hypothetical judgments are listed below.

$$\frac{}{\Delta; (\Gamma, A\ \textit{true}) \vdash A\ \textit{true}}$$

$$\frac{\Delta; (\Gamma, A\ \textit{true}) \vdash B\ \textit{true}}{\Delta; \Gamma \vdash A \to B\ \textit{true}} \qquad \frac{\Delta; \Gamma \vdash A \to B\ \textit{true} \quad \Delta; \Gamma \vdash A\ \textit{true}}{\Delta; \Gamma \vdash B\ \textit{true}}$$

$$\frac{}{(\Delta, A\ \textit{nec}); \Gamma \vdash A\ \textit{true}}$$

$$\frac{\Delta; \cdot \vdash A\ \textit{true}}{\Delta; \Gamma \vdash \Box A\ \textit{true}} \qquad \frac{\Delta; \Gamma \vdash \Box A\ \textit{true} \quad (\Delta, A\ \textit{nec}); \Gamma \vdash B\ \textit{true}}{\Delta; \Gamma \vdash B\ \textit{true}}$$

[1] The name Constructive S4 has already been proposed in [AMdPR01] for a logic obtained when the six propositions from Examples 2 and 3 are added to the axioms of the intuitionistic propositional calculus. The inference rules of Constructive S4 include modus ponens and the rule of necessitation. We take the liberty to use the same name for our logic, because we expect that it is equal to the logic proposed in [AMdPR01]. This conjecture, however, has not been proved.

$$\frac{\Delta;\Gamma \vdash A\ \mathit{true}}{\Delta;\Gamma \vdash A\ \mathit{poss}}$$

$$\frac{\Delta;\Gamma \vdash A\ \mathit{poss}}{\Delta;\Gamma \vdash \Diamond A\ \mathit{true}} \qquad \frac{\Delta;\Gamma \vdash \Diamond A\ \mathit{true} \quad \Delta; A\ \mathit{true} \vdash B\ \mathit{poss}}{\Delta;\Gamma \vdash B\ \mathit{poss}}$$

$$\frac{\Delta;\Gamma \vdash \Box A\ \mathit{true} \quad (\Delta, A\ \mathit{nec});\Gamma \vdash B\ \mathit{poss}}{\Delta;\Gamma \vdash B\ \mathit{poss}}$$

The inference rules indeed respect the definitional properties of the hypothetical judgments, as the following theorem shows.

Theorem 1 (Substitution principles)

1. *If* $\Delta;\Gamma \vdash A\ \mathit{true}$ *then*
   (a) *if* $\Delta;(\Gamma, A\ \mathit{true}) \vdash B\ \mathit{true}$ *then* $\Delta;\Gamma \vdash B\ \mathit{true}$
   (b) *if* $\Delta;(\Gamma, A\ \mathit{true}) \vdash B\ \mathit{poss}$ *then* $\Delta;\Gamma \vdash B\ \mathit{poss}$
2. *If* $\Delta;\cdot \vdash A\ \mathit{true}$, *then*
   (a) *if* $(\Delta, A\ \mathit{nec});\Gamma \vdash B\ \mathit{true}$, *then* $\Delta;\Gamma \vdash B\ \mathit{true}$
   (b) *if* $(\Delta, A\ \mathit{nec});\Gamma \vdash B\ \mathit{poss}$, *then* $\Delta;\Gamma \vdash B\ \mathit{poss}$
3. *If* $\Delta;\Gamma \vdash A\ \mathit{poss}$ *and* $\Delta; A\ \mathit{true} \vdash B\ \mathit{poss}$, *then* $\Delta;\Gamma \vdash B\ \mathit{poss}$

Proof: Statements (1) and (2) are proved by straightforward induction over the structure of the first given derivation. Statement (3) is proved by induction over the second derivation. ■

1.2 Modal λ-calculus

1.2.1 Judgments and proof terms

Following the type-theoretic methodology of Martin-Löf [ML96], in this section we annotate the judgments of our natural deduction with *proof terms*. A proof term serves as a witness for its corresponding judgment, in the sense that a derivation of the judgment may be recovered by inspection of the proof term. If a judgment is *annotated* with a proof term, then each judgment contains in itself an instruction on how to discover its derivation. It is not necessary to look outside of the judgment to establish an evidence for it.

In this case, instead of *A true* and *A poss*, we will have judgments $e : A$ and $f \div A$. The meaning of the judgment $e : A$ is that "e is a proof term witnessing that *A true*". The meaning of the judgment $f \div A$ is that "f is a proof term witnessing that *A poss*. The elements of the syntactic category e are called *expressions*, and the elements of the syntactic category f are called *phrases*.

As an illustration, consider the rules for conjunction from Section 1.1.1, which we here decorate with proof terms.

$$\frac{e_1 : A \qquad e_2 : B}{\langle e_1, e_2 \rangle : B}$$

$$\frac{e : A \wedge B}{\mathbf{fst}\ e : A} \qquad \frac{e : A \wedge B}{\mathbf{snd}\ e : B}$$

The proof-annotated rules uncover the computational content of the logic, as proofs can be treated as *programs*, and propositions can be treated as *types*. For example, the introduction rule for conjuction makes it explicit that the proof of $A \wedge B$ can be *constructed* using $e_1 : A$ and $e_2 : B$ as a *pair* $\langle e_1, e_2 \rangle : A \wedge B$. The elimination forms **fst** e and **snd** e destruct a pair by taking its first or second component.

Local reduction and local expansions can now be stated using proof terms for conjuction.

$$\begin{array}{rcl} \mathbf{fst}\ \langle e_1, e_2 \rangle & \Longrightarrow_R & e_1 \\ \mathbf{snd}\ \langle e_1, e_2 \rangle & \Longrightarrow_R & e_2 \end{array}$$

$$e : A \wedge B \quad \Longrightarrow_E \quad \langle \mathbf{fst}\ e, \mathbf{snd}\ e \rangle$$

As customary for type theory, the proof-annotated version of local reduction is what carries the computational meaning of the logical construct, because it explains how a program can reduced, i.e. evaluated. In the case of conjuction, for example, local reductions formally specify what it means to select the first or the second element of a pair. If the pair has the form $\langle e_1, e_2 \rangle$ then, in order to compute its first element we simply need to take the expression e_1, and to compute the second element, we need to take e_2. On the other hand, local expansion implements the principle of extensionality. In the case of conjuction, it states that every expression $e{:}A \wedge B$ is guaranteed to be equal (in an appropriate sense of equality which we do not define here) to the pair $\langle \mathbf{fst}\ e, \mathbf{snd}\ e \rangle$.

To obtain the proof-annotated versions of the hypothetical judgments, we first label the assumptions from the contexts Γ and Δ with variables. We write $x{:}A$ for "x is a proof of A *true*", and $u{::}A$ for "u is a proof of A *nec*". The usual assumptions of variables contexts hold here as well: variables declared in Δ and Γ are considered different and we tacitly employ α-renaming to guarantee this invariant. We will call variables from Γ *ordinary* or *value* variables, while the variables from Δ will be *modal* or *expression* variables. The decorated hypothesis rule now has the form

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

and the corresponding substitution principle formalizes how the hypothetical judgments depend on the value variables.

Principle (Value substitution)
If $\Delta; \Gamma \vdash e_1 : A$ then the following holds:

1. if $\Delta; (\Gamma, x{:}A) \vdash e_2 : B$, then $\Delta; \Gamma \vdash [e_1/x]e_2 : B$.
2. if $\Delta; (\Gamma, x{:}A) \vdash f_2 \div B$, then $\Delta; \Gamma \vdash [e_1/x]f_2 \div B$.

In this principle, we denote by $[e_1/x]e_2$ and $[e_1/x]f_2$ the result of capture-avoiding substitution of e_1 for x in the expression e_2 and phrase f_2, respectively. Because the substitution principle now has access to proof terms, it can explicitly state that the judgments are parametric with respect to variables. The expression $e_1 : A$ can only be *substituted* for x in the hypothetical proofs, but cannot be used in any other way. This reliance on substitution was only implicitly assumed in the previous formulations of the principle, but once proof terms are provided, it can be stated explicitly.

The rules for implication introduction and elimination are annotated using λ-abstraction and function application, respectively.

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

As usual, the local soundness and completeness are witnessed by local reduction and expansion on the proof terms, which in this case are the ordinary β-reduction and η-expansion of the λ-calculus.

$$\begin{array}{lll} (\lambda x{:}A.\ e_1)\ e_2 & \Longrightarrow_R & [e_2/x]e_1 \\ \\ e : A \to B & \Longrightarrow_E & \lambda x{:}A.\ (e\ x) \qquad \text{where } x \text{ not free in } e \end{array}$$

Example 4 The following are well-typed expression in the modal λ-calculus.

1. $\Delta; \Gamma \vdash \lambda x.\ x : A \to A$

2. $\Delta; \Gamma \vdash \lambda x.\ \lambda y.\ x : A \to B \to A$

3. $\Delta; \Gamma \vdash \lambda f.\ \lambda g.\ \lambda x.\ (f\ x)\ (g\ x) : (A \to B \to C) \to (A \to B) \to A \to C$

■

The hypothesis rule for modal variables is annotated as follows

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

and the corresponding substitution principle is given below.

Principle (Modal substitution)
If $\Delta; \cdot \vdash e_1 : A$, then the following holds.

1. if $(\Delta, u{::}A); \Gamma \vdash e_2 : B$, then $\Delta; \Gamma \vdash [\![e_1/u]\!]e_2 : B$

2. if $(\Delta, u{::}A); \Gamma \vdash f_2 \div B$, then $\Delta; \Gamma \vdash [\![e_1/u]\!]f_2 \div B$

In this principle, the operations $[\![e_1/u]\!]e_2$ and $[\![e_1/u]\!]f_2$ are capture-avoiding substitutions of e_1 for the *modal* variable u in e_2 and f_2, respectively. We use a different notation because the operation substitutes a different kind of variables. The separate notation will come handy in the future sections, where we redefine the modal substitution so that it differs from ordinary substitution.

The proof-annotated forms of the introduction and elimination rules for □ are as follows.

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

and the local soundness and completeness are witnessed by the local reduction and expansion

$$\textbf{let box}\ u = \textbf{box}\ e_1\ \textbf{in}\ e_2 \quad \Longrightarrow_R \quad [\![e_1/u]\!]e_2$$

$$e : \Box A \quad \Longrightarrow_E \quad \textbf{let box}\ u = e\ \textbf{in box}\ u$$

Example 5 The following are well-typed expressions in the modal λ-calculus.

1. $\Delta; \Gamma \vdash \lambda x.\ \textbf{let box}\ u = x\ \textbf{in}\ u : \Box A \to A$
2. $\Delta; \Gamma \vdash \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box box}\ u : \Box A \to \Box\Box A$
3. $\Delta; \Gamma \vdash \lambda x.\ \lambda y.\ \textbf{let box}\ u = x\ \textbf{in let box}\ v = y\ \textbf{in box}\ u\ v$
   $: \Box(A \to B) \to \Box A \to \Box B$

■

The inference rules for possibility are easily annotated as well. The proof terms that we use in this case belong to the syntactic category of *phrases*, and we start by rewriting the definitional clauses for possibility (Section 1.1.4) to take phrases into account.

1. If $\Delta; \Gamma \vdash e : A$ then $\Delta; \Gamma \vdash e \div A$.
2. If $\Delta; \Gamma \vdash f_1 \div A$ and $\Delta; x{:}A \vdash f_2 \div B$, then $\Delta; \Gamma \vdash \langle\!\langle f_1/x \rangle\!\rangle f_2 \div B$.

The definitional clause (1) makes it evident that each expression $e : A$ may be considered as a phrase witnessing $e \div A$. The definitional clause (2) takes a form of a *phrase substitution principle.* It uses a new operation of *phrase substitution* $\langle\!\langle f_1/x \rangle\!\rangle f_2$ which we define below after introducing the other phrase constructors.

Just as in Section 1.1.4, the formulation of the proof-annotated possibility judgment, uses an explicit inference rule to realize the definitional clause (1).

$$\frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash e \div A}$$

The introduction and elimination rules are decorated using the new phrase constructors **dia** and **let dia** as follows.

$$\frac{\Delta; \Gamma \vdash f \div A}{\Delta; \Gamma \vdash \textbf{dia}\ f : \Diamond A} \qquad \frac{\Delta; \Gamma \vdash e : \Diamond A \qquad \Delta; x{:}A \vdash f \div B}{\Delta; \Gamma \vdash \textbf{let dia}\ x = e\ \textbf{in}\ f \div B}$$

Notice that the typing rule for **let dia** erases the context Γ, and introduces a new variable $x{:}A$, which is considered bound by the **let dia** constructor.

There is also an additional rule for eliminating □ into the possibility judgment.

$$\frac{\Delta; \Gamma \vdash e : \Box A \qquad (\Delta, u{::}A); \Gamma \vdash f \div B}{\Delta; \Gamma \vdash \textbf{let box}\ u = e\ \textbf{in}\ f \div B}$$

$$\begin{aligned} \textbf{let dia}\ x = \textbf{dia}\ f_1\ \textbf{in}\ f_2 \quad &\Longrightarrow_R \quad \langle\!\langle f_1/x \rangle\!\rangle f_2 \\ \textbf{let box}\ u = \textbf{box}\ e_1\ \textbf{in}\ f_2 \quad &\Longrightarrow_R \quad [\![e_1/u]\!] f_2 \end{aligned}$$

$$e : \Diamond A \quad \Longrightarrow_E \quad \textbf{dia}\ (\textbf{let dia}\ x = e\ \textbf{in}\ x)$$

The new substitution operation $\langle\!\langle f_1/x \rangle\!\rangle f$ is defined in a slightly unusal way, by induction on the structure of f_1, rather than by induction on the structure of f.

$$\begin{array}{rcl} \langle\!\langle e_1/x \rangle\!\rangle f & = & [e_1/x]f \\ \langle\!\langle \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle f & = & \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle f \\ \langle\!\langle \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle f & = & \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle f \end{array}$$

Example 6 The following are well-typed terms in the modal λ-calculus.

1. $\Delta; \Gamma \vdash \lambda x.\ \mathbf{dia}\ x : A \to \Diamond A$
2. $\Delta; \Gamma \vdash \lambda x.\ \mathbf{dia}\ (\mathbf{let\ dia}\ y = x\ \mathbf{in\ let\ dia}\ z = y\ \mathbf{in}\ z) : \Diamond\Diamond A \to \Diamond A$
3. $\Delta; \Gamma \vdash \lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ dia}\ (\mathbf{let\ dia}\ z = y\ \mathbf{in}\ u\ z)$
   $: \Box(A \to B) \to \Diamond A \to \Diamond B$

■

1.2.2 Summary of the system

This section sumarizes the main aspects of the definition of the modal λ-calculus.

$$\begin{array}{lrcl} \textit{Types} & A, B & ::= & P \mid A \to B \mid \Box A \mid \Diamond A \\ \textit{Expressions} & e & ::= & x \mid \lambda x{:}A.\ e \mid e_1\ e_2 \\ & & & \mid u \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \\ & & & \mid \mathbf{dia}\ f \\ \textit{Phrases} & f & ::= & e \mid \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f \\ & & & \mid \mathbf{let\ box}\ u = e\ \mathbf{in}\ f \\ \textit{Ordinary contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\ \textit{Modal contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}A \end{array}$$

The calculus contains two typing judgments:

$$\Delta; \Gamma \vdash e : A \qquad \text{and} \qquad \Delta; \Gamma \vdash f \div A$$

The first judgment states that the expression e has type A relative to the modal context Δ and ordinary context Γ. Alternatively, e is a proof of A *true*, under necessary hypotheses Δ and true hypotheses Γ. The second judgment states that the phrase f has type A relative to the modal context Δ and ordinary context Γ. The alternative reading of this judgment is that f is a proof of A *poss* under necessary hypotheses Δ and true hypotheses Γ. The following are the inference rules of the two judgments.

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

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

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

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

$$\frac{\Delta;\Gamma \vdash e : A}{\Delta;\Gamma \vdash e \div A}$$

$$\frac{\Delta;\Gamma \vdash f \div A}{\Delta;\Gamma \vdash \mathbf{dia}\ f : \Diamond A} \qquad \frac{\Delta;\Gamma \vdash e : \Diamond A \quad \Delta; x{:}A \vdash f \div B}{\Delta;\Gamma \vdash \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f \div B}$$

$$\frac{\Delta;\Gamma \vdash e : \Box A \quad (\Delta, u{::}A);\Gamma \vdash f \div B}{\Delta;\Gamma \vdash \mathbf{let\ box}\ u = e\ \mathbf{in}\ f \div B}$$

There are three different forms of capture-avoiding substitution in the calculus:

1. *Ordinary substitution.* $[e_1/x]e$ and $[e_1/x]f$ which replace the value variable x by the expression e_1
2. *Modal substitution.* $[\![e_1/u]\!]e$ and $[\![e_1/u]\!]f$ which replace the modal variable u by the expression e_1
3. *Phrase substitution.* $\langle\!\langle f_1/x\rangle\!\rangle f$ which replaces an ordinary variable x by a phrase f_1.

The ordinary and modal substitutions are defined in a standard way, but the phrase substitution is a bit unusal. It is defined by induction on the structure of f_1, rather than f, using the following three clauses.

$$\begin{array}{rcl}
\langle\!\langle e_1/x\rangle\!\rangle f & = & [e_1/x]f \\
\langle\!\langle \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ f_2/x\rangle\!\rangle f & = & \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ \langle\!\langle f_2/x\rangle\!\rangle f \\
\langle\!\langle \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2/x\rangle\!\rangle f & = & \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ \langle\!\langle f_2/x\rangle\!\rangle f
\end{array}$$

The following theorem proves that the presented formulation respect the substitution principles stated before as definitional properties of the judgments.

Theorem 2 (Substitution principles)

1. *If* $\Delta;\Gamma \vdash e_1 : A$ *then*
   - (a) *if* $\Delta;(\Gamma, x{:}A) \vdash e_2 : B$ *then* $\Delta;\Gamma \vdash [e_1/x]e_2 : B$
   - (b) *if* $\Delta;(\Gamma, x{:}A) \vdash f_2 \div B$ *then* $\Delta;\Gamma \vdash [e_1/x]f_2 \div B$
2. *If* $\Delta;\cdot \vdash e_1 : A$*, then*
   - (a) *if* $(\Delta, u{::}A);\Gamma \vdash e_2 : B$*, then* $\Delta;\Gamma \vdash [e_1/u]e_2 : B$
   - (b) *if* $(\Delta, u{::}A);\Gamma \vdash f_2 \div B$*, then* $\Delta;\Gamma \vdash [e_1/u]f_2 \div B$
3. *If* $\Delta;\Gamma \vdash f_1 \div A$ *and* $\Delta; x{:}A \vdash f_2 \div B$*, then* $\Delta;\Gamma \vdash \langle\!\langle f_1/x\rangle\!\rangle f_2 \div B$

Proof: By straightforward induction on the structure of the typing derivations [PD01]. ■

1.3 Notes

Related work on the proof theory of intuitionistic modal logics

As already mentioned, our presentation of constructive S4 from the previous section was based on the work by Pfenning and Davies. But other approaches to the natural deduction have also been proposed. For example, in the work of Alechina at al. [AMdPR01], Biermana and de Paiva [BdP00], and Benton, Bierman and de Paiva [BBdP98], the modalities are formulated in the following way.

$$\frac{\Gamma \vdash e_1 : \Box A_1 \quad \cdots \quad \Gamma \vdash e_n : \Box A_n \qquad x_1 : \Box A_1, \ldots, x_n : \Box A_n \vdash e : B}{\Gamma \vdash \textbf{box}\ e\ \textbf{with}\ \bar{e}\ \textbf{for}\ \bar{x} : \Box B} \qquad \frac{\Gamma \vdash e : \Box A}{\Gamma \vdash \textbf{unbox}\ e : A}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \textbf{dia}\ e : \Diamond A}$$

$$\frac{\Gamma \vdash e_1 : \Box A_1 \quad \cdots \quad \Gamma \vdash e_n : \Box A_n \qquad \Gamma \vdash e : \Diamond B \qquad x_1 : \Box A_1, \ldots, x_n : \Box A_n, y : B \vdash f : \Diamond C}{\Gamma \vdash \textbf{let dia}\ y = e\ \textbf{in}\ f\ \textbf{with}\ \bar{e}\ \textbf{for}\ \bar{x} : \Diamond C}$$

This formulation is similar to the approach by Prawitz in [Pra65]. Notice how the □-introduction and ◇-elimination rules require explicit substitution. This is avoided in our presentation in Section 1 by separating ordinary variables from modal variables.

In fact, in the subsequent sections (Section 2 and Section 3) we will introduce *Partial* CS4, which extends the ordinary CS4 with explicit substitutions. The use of explicit substitutions there, however, will be directly opposite to the CS4 from this note. In Partial CS4, it will be the □-elimination and ◇-introduction rules that use explicit substitutions. This kind of approach will provide a lot of additional expressiveness and flexibility when compared to ordinary CS4.

Another approach to the natural deduction of constructive modal logic in general, and versions of modal S4 in particular, is exemplified by the work of Alex Simpson [Sim94]. The truth judgment used in this kind of approaches has the form $w : A$, denoting that the proposition A is true at the world w. The inference rules explicitly manipulate the accessibility relation R for the modal logic in question. We show below the rules for modalities, in the form of derivation trees, as formulated in [Sim94].

$$\frac{\begin{array}{c}[wRw'] \\ \vdots \\ w' : A\end{array}}{w : \Box A} \qquad \frac{w' : \Box A \quad w'Rw}{w : A}$$

$$\frac{w' : A \quad wRw'}{w : \Diamond A} \qquad \frac{w' : \Diamond A \qquad \begin{array}{c}[w'' : A][w'Rw''] \\ \vdots \\ w : B\end{array}}{w : B}$$

It is interesting that the version of modal logic formulated by Simpson is slightly different from the Constructive S4 introduced in Section 1. In particular, Simpsons' formulation, which is called

Intuitionistic S4 (or IS4 for short), admits the following theorems.

1. $\neg\Diamond\bot$
2. $\Diamond(A \vee B) \to (\Diamond A \vee \Diamond B)$
3. $(\Diamond A \to \Box B) \to \Box(A \to B)$

These propositions are not derivable in an appropriate extensions of CS4 with disjunction and $\bot$.

Simpson's dissertation also axiomatizes many other intuitionistic modal logics, and is a good source of hystorical references on this subject.

Related work on the Kripke semantics of Constructive S4

A Kripke model of CS4 is presented by Alechina at al. in [AMdPR01]. The model consists of a set of worlds W and two accessibility relations, one for the intuitionistic implication $\sqsubseteq$, and one for the modalities $\to$. More formally:

Definition 3
A Kripke model of CS4 is a structure $M = (W, \sqsubseteq, \to, \models)$, where W is a non-empty set, $\sqsubseteq$ and $\to$ are reflexive and transitive binary relations on W, and $\models$ a relation between elements of $w \in W$ and propositions A, such that:

- *$\sqsubseteq$ is monotone with respect to atomic propositions, i.e. if $w \sqsubseteq w'$ and P is an atomic proposition, then $w \models P$ implies $w' \models P$*
- *$\sqsubseteq$ and $\to$ are coherent in the following sense:*

  if $w \to v$ and $v \sqsubseteq v'$, there exists w' such that $w \sqsubseteq w'$ and $w' \to v'$

- *the relation $\models$ has the following properties*
  - $w \models \top$
  - $w \models A \wedge B$ *iff* $w \models A$ *and* $w \models B$
  - $w \models A \vee B$ *iff* $w \models A$ *or* $w \models B$
  - $w \models A \to B$ *iff for all* $w' \sqsupseteq w$, $w' \models A$ *implies* $w' \models B$
  - $w \models \Box A$ *iff for all* $w' \sqsupseteq w$ *and* $u' \leftarrow w'$, $u' \models A$
  - $w \models \Diamond A$ *iff for all* $w' \sqsupseteq w$ *there exists* $u' \leftarrow w'$ *such that* $u' \models A$

  The definition does not require that $w \not\models \bot$. Rather, inconsistent worlds are permitted, as long as the following requirements are met:

  - *if* $w \models \bot$ *and* $w \sqsubseteq w'$ *then* $w' \models \bot$
  - *if* $w \models \bot$ *then for every atomic proposition* P, $w \models P$

In his dissertation [Sim94], Simpson describes Kripke semantics for IS4, but not for CS4. The differences between the two semantics include:

1. The semantics for IS4 does not allow inconsistent worlds. The inconsistent worlds are the feature that eliminates the theorem $\neg\Diamond\bot$ in the CS4 semantics.

2. In IS4, $w \models \Diamond A$ iff there exists $w' \sqsupseteq w$ such that $w' \models A$. This definition permits the theorem $\Diamond(A \vee B) \to (\Diamond A \vee \Diamond B)$.

3. In IS4, a further coherence condition is imposed between the two accessibility relations. In particular, the IS4 semantics requires that

   if $w' \sqsupseteq w$ and $w \to v$, then there exists v' such that $w' \to v'$ and $v' \sqsupseteq v$

   The presence of this condition in IS4 permits the theorem $(\Diamond A \to \Box B) \to \Box(A \to B)$.

Related work on the categorical semantics of Constructive S4

Categorical semantics for CS4 has been considered by several authors, most notably by Kobayashi [Kob97], Bierman and de Paiva [BdP00] and Alechina at al. [AMdPR01]. As established in these papers, a categorical model for CS4 consists of a cartesian closed category with co-products $\mathbb{C}$, together with a *monoidal comonad* $\Box$ and a $\Box$-*strong monad* $\Diamond$.

We present here only the standard definitions of a monad and a comonad, but for simplicity, we omit the details related to *monoidal functors* and $\Box$-*strength*. The ommitted details describe various important commuting conditions that the functors $\Box$ and $\Diamond$ have to satisfy. We refer to the work cited above for a thorough treatment.

Definition 4

1. *A monad* $\Diamond$ *is an endofunctor on* $\mathbb{C}$ *with two natural transformations* $\eta : A \to \Diamond A$ *and* $\mu : \Diamond\Diamond A \to \Diamond A$*, satisfying the following equations:*

   (a) $\mu_A \circ \eta_{\Diamond A} = \mu_A \circ \Diamond(\eta_A) = id_{\Diamond A}$

   (b) $\mu_A \circ \mu_{\Diamond A} = \mu_A \circ \Diamond(\mu_A)$

2. *A comonad* $\Box$ *is an endofunctor on* $\mathbb{C}$ *with two natural transformations* $\varepsilon : \Box A \to A$ *and* $\delta : \Box A \to \Box\Box A$ *satisfying the following equations:*

   (a) $\varepsilon_{\Box A} \circ \delta_A = \Box(\varepsilon_A) \circ \delta_A = id_{\Box A}$

   (b) $\delta_{\Box A} \circ \delta_A = \Box(\delta_A) \circ \delta_A$

Chapter 2

Partial modal logic

2.1 Natural deduction

2.1.1 Partial judgments and supports

In this section, we develop the notion of *partial* truth judgments. The idea is to capture that a derivation or a witness of some fact may be obtained, but only if a certain condition is satisfied. The syntactic form of the partial truth judgment is

$$A\ \mathit{true}\,[C]$$

where A is a proposition, and C is a *supporting condition*, or *support*, for short. The semantics of this judgment is to witness that a proof of $A\ \mathit{true}$ can be obtained if the condition C is fulfilled. To emphasize this contrast between the partial judgment $A\ \mathit{true}\,[C]$, and the ordinary judgment $A\ \mathit{true}$ defined in Chapter 1, we will call the later judgment *total*. Partial truth judgments resemble somewhat the idea behind total hypothetical judgments. In a hypothetical judgment

$$A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \vdash A\ \mathit{true}$$

the condition on $A\ \mathit{true}$ consists of a set of hypotheses $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$, and a derivation of $A\ \mathit{true}$ can be obtained by means of *substitution* from the derivations of $A_1\ \mathit{true}$, ..., $A_n\ \mathit{true}$. Because these derivations must be substituted without any inspection or modification, the judgment $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true} \vdash A\ \mathit{true}$ is said to be *parametric* in its hypotheses.

Partial judgments, however, are intended to be much more general. For example, given a derivation of $A\ \mathit{true}\,[C]$ and a witness that the condition C is satisfied, it will be possible to reconstruct a derivation of $A\ \mathit{true}$, but it is not required that the witness for C is used only via substitution. In fact, any particular application may specify a different way to obtain $A\ \mathit{true}$ from a witness of C and a derivation of $A\ \mathit{true}\,[C]$. In this section, we remain uncommitted and treat this dependency in the abstract. That will lead to properties of partial judgments that persist across a broader range of applications.

The process of transforming the proof of $A\ \mathit{true}\,[C]$, when a witness for C is provided, is called *reflection*, and will typically be justified by the meta-theoretic properties of the truth judgment and its derivations. In this sense, a support C may be seen as a condition in the meta-logical reasoning about derivability of $A\ \mathit{true}$. Correspondingly, reflection allows that a conclusion obtained in the meta logic be coerced into the truth judgment, when the condition C holds.

Reflection will have interesting consequences for the computational content of partial truth, when propositions are seen as types, and proofs as programs. For example, a proof of $A\ \mathit{true}\,[C]$

may be considered as a program that produces a value of type A, but only if executed in a run-time environment that satisfies the condition C. In this case, reflection may be defined as evaluation, or for that matter, any other kind of type-preserving program transformation.

In the remainder of this section, we embark on the formulation and analysis of partial truth, which will eventually motivate a development of a whole modal logic of partial judgments, with very diverse applications in functional programming. Because supports are syntactic equivalents of meta-logical propositions, any definition of partial truth must start by formally explaining the correspondence between a given support C and the proposition that C represents. For that purpose, we will use the judgments

$$C\ \mathit{supp} \qquad \text{and} \qquad C\ \mathit{holds}$$

which will need to be defined for any particular application, but which we keep abstract for the time being. The judgment $C\ \mathit{supp}$ determines if a support C is well-formed, and the judgment $C\ \mathit{holds}$ determines if a condition represented by C is satisfied. It is implicitly assumed that $C\ \mathit{holds}$ is itself well-formed only if C is a valid support, i.e. only if $C\ \mathit{supp}$.

In order to formally capture the causal dependency between supports, we need to impose some further algebraic structure. In particular, we require that the set of all supports is *partially ordered* by the reflexive, anti-symmetric and transitive relation $\sqsubseteq$, and that it has a *minimal element* 0. The idea is that $C \sqsubseteq D$ if and only if the condition associated with D *implies in the meta logic* the condition associated with C. In this case, the minimal support 0 simply corresponds to the condition that is always, trivially, true. To formalize this intuition, the support judgments will contain the derivation rules

$$\frac{}{0\ \mathit{supp}} \qquad \text{and} \qquad \frac{}{0\ \mathit{holds}}$$

which establish that 0 is a well-formed support, and that 0 corresponds to a true condition, respectively. We also require the following *support weakening* principle.

Principle (Support weakening)
If $C \sqsubseteq D$, than any witness of $D\ \mathit{holds}$ is a witness of $C\ \mathit{holds}$ as well.

Having introduced the support judgments and ordering, we can now provide a formal definition for the partial truth. Henceforth, we write

$$A\ \mathit{true}\,[C]$$

if and only if $C\ \mathit{holds}$ implies $A\ \mathit{true}$. We assume here that the partial truth judgment is well formed, i.e. that $A\ \mathit{prop}$ and $C\ \mathit{supp}$. Notice that each particular application will have to specify concretely the dependency between the derivations of $C\ \mathit{holds}$ and $A\ \mathit{true}\,[C]$. However, having in mind that the support 0 always holds, we impose the requirement that $A\ \mathit{true}\,[0]$ if and only if $A\ \mathit{true}$. This will allow us to regard the total truth judgment as a special case of its partial counterpart.

We also consider a partial version of the support judgment $C\ \mathit{holds}$, and write

$$C\ \mathit{holds}\,[D]$$

if and only if $D\ \mathit{holds}$ implies $C\ \mathit{holds}$. In order for this judgment to respect the support ordering, we require the following as one of its derivation rules.

$$\frac{C \sqsubseteq D}{C\ \mathit{holds}\,[D]}$$

Again, we insist that C *holds* $[0]$ if and only if C *holds*, and treat C *holds* as a special case of C *holds* $[D]$, when D is the 0 support.

The two partial judgments are further required to respect the partial ordering $\sqsubseteq$, in the sense of the following support weakening principle. The support weakening principle stated previously is subsumed as a special case (obtained when the support D' is taken to be 0).

Principle (Support weakening)
Let C and D be well-formed supports with $C \sqsubseteq D$. Then the following holds:

1. if A *true* $[C]$, then A *true* $[D]$
2. if C_1 *holds* $[C]$, then C_1 *holds* $[D]$
3. if D *holds* $[D_1]$, then C *holds* $[D_1]$

Finally, in order to relate partial truth with the partial support judgment, we impose the following requirement phrased as a *reflection principle*.

Principle (Reflection)
If C *holds* $[D]$, then the following holds:

1. if A *true* $[C]$, then A *true* $[D]$
2. if C_1 *holds* $[C]$, then C_1 *holds* $[D]$

Notice that if D is taken to be 0, then the reflection principle makes a connection between the partial truth and support judgments and their total counterparts.

2.1.2 Hypothetical partial judgments

The next step in the development of the logic of partial truth is to extend the non-hypothetical reasoning associated with supports and reflection, and parametrize the judgments with respect to a context of hypotheses

$$A_1 \ \mathit{true}, \ldots, A_n \ \mathit{true}.$$

As customary, we use Γ to range over contexts, and generalize the judgments to the following form

$$\Gamma \vdash C \ \mathit{holds}\,[D] \qquad \text{and} \qquad \Gamma \vdash A \ \mathit{true}\,[D].$$

Of course, the usual coherence conditions apply to this generalization. In particular, if Γ is the empty context, the new judgments reduce to the non-hypothetical partial judgments from the previous section. Analogously, if D is the minimal support 0, we require that the partial judgment $\Gamma \vdash A$ *true* $[0]$ be equivalent to the total judgment $\Gamma \vdash A$ *true*. In a similar fashion, we will abbreviate $\Gamma \vdash C$ *holds* $[0]$ simply as $\Gamma \vdash C$ *holds*. To simplify matters, the definition of the partial judgments will immediately assume that Γ is a set, so that the judgment will satisfy the structural rule of exchange.

Henceforth, we define the judgment

$$\Gamma \vdash C \ \mathit{holds}\,[D]$$

to be satisfied only if a derivation of C *holds* $[D]$ can be obtained given the derivations of A_1 *true* $[D]$, $\ldots$, A_n *true* $[D]$. It is important that the derivations of A_i *true* $[D]$ must be used in a parametric way, i.e. by means of substitutions. These proofs may not be modified in any way, and in particular, they are not subject to reflection. The rules of the judgment must extend accordingly, to account for the context Γ. For example, the following is a rule of the hypothetical support judgment which relates causally dependent contexts.

$$\frac{C \sqsubseteq D}{\Gamma \vdash C\ \mathit{holds}\,[D]}$$

The partial hypothetical truth judgment is defined in the similar fashion. We say that

$$\Gamma \vdash A\ \mathit{true}\,[D]$$

only if a derivation of $A\ \mathit{true}\,[D]$ may be obtained from derivations of $A_1\ \mathit{true}\,[D], \dots, A_n\ \mathit{true}\,[D]$, by means of substitution. Notice how the scope of the support D in the above definitions extends across the whole judgment. The support modifies the hypotheses $A_1\ \mathit{true}, \dots, A_n\ \mathit{true}$, as well as the conclusions $C\ \mathit{holds}$ and $A\ \mathit{true}$.[1]

As a coherence condition, we impose a support weakening principle for hypothetical partial judgments analogous to the support weakening principle from the previous section.

Principle (Support weakening)
Let C and D be well-formed supports with $C \sqsubseteq D$. Then the following holds:

1. if $\Gamma \vdash A\ \mathit{true}\,[C]$, then $\Gamma \vdash A\ \mathit{true}\,[D]$
2. if $\Gamma \vdash C_1\ \mathit{holds}\,[C]$, then $\Gamma \vdash C_1\ \mathit{holds}\,[D]$
3. if $\Gamma \vdash D\ \mathit{holds}\,[D_1]$, then $\Gamma \vdash C\ \mathit{holds}\,[D_1]$

The extensions of the reflection principle is also straightforward, but with one *essential restriction.*

Principle (Reflection)
If $\Gamma \vdash C\ \mathit{holds}\,[D]$, then the following holds:

1. if $\cdot \vdash A\ \mathit{true}\,[C]$, then $\Gamma \vdash A\ \mathit{true}\,[D]$
2. if $\cdot \vdash C_1\ \mathit{holds}\,[C]$, then $\Gamma \vdash C_1\ \mathit{holds}\,[D]$

It is of crucial importance to observe that the above reflection principle involve premises that are *categorical*, i.e. do not depend on any hypotheses. In the case of supports, we reflect a proof of $\cdot \vdash C_1\ \mathit{holds}\,[C]$, and in the case of truth, we reflected a proof of $\cdot \vdash A\ \mathit{true}\,[C]$, but neither of these judgments depends on Γ. Indeed, reflecting a hypothetical proof would violate its hypothetical nature, because the operations of substitution and reflection *need not commute.* Any sound way to combine hypothetical reasoning embodied by substitution, with the non-hypothetical reasoning embodied by reflection, must impose that reflection is only used on *categorical proofs.*

The hypothetical nature of the partial judgments is axiomatized by means of the hypothesis rule

$$\frac{}{\Gamma, A\ \mathit{true} \vdash A\ \mathit{true}\,[D]}$$

[1] In the terminology of modal logic, we can say that the support D is a condition on the current world. Because the hypotheses $A_1\ \mathit{true}, \dots, A_n\ \mathit{true}$ are associated with the current world, their derivations are allowed to be partial in D.

The corresponding substitution principle simply axiomatizes the definitional properties.

Principle (Substitution)
If $\Gamma \vdash A\ \mathit{true}\,[C]$, then the following holds:

1. if $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}\,[C]$, then $\Gamma \vdash B\ \mathit{true}\,[C]$
2. if $\Gamma, A\ \mathit{true} \vdash D\ \mathit{holds}\,[C]$, then $\Gamma \vdash D\ \mathit{holds}\,[C]$

The partial judgments also require rules to witness that proofs can be derived by reflection. We state the appropriate rules here, but repeat that each specific application may define its own notions of supports and reflection. For each of these applications, we will have to prove that reflection is sound, i.e. that the reflected and the derived proof are witnessing one and the same judgment.

$$\frac{\Gamma \vdash C\ \mathit{holds}\,[D] \qquad \cdot \vdash A\ \mathit{true}\,[C]}{\Gamma \vdash A\ \mathit{true}\,[D]} \qquad \frac{\Gamma \vdash C\ \mathit{holds}\,[D] \qquad \cdot \vdash C_1\ \mathit{holds}\,[C]}{\Gamma \vdash C_1\ \mathit{holds}\,[D]}$$

Just as in the case of total judgments, we can internalize the hypothetical dependence between an antecedent and a conclusion by means of the new propositional constructor of implication $A \to B$. We say that $\Gamma \vdash A \to B\ \mathit{true}\,[C]$ if and only if $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}\,[C]$ implies $\Gamma \vdash B\ \mathit{true}\,[C]$. The new operator is axiomatized by standard introduction and elimination rules.

$$\frac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}\,[C]}{\Gamma \vdash A \to B\ \mathit{true}\,[C]} \qquad \frac{\Gamma \vdash A \to B\ \mathit{true}\,[C] \qquad \Gamma \vdash A\ \mathit{true}\,[C]}{\Gamma \vdash B\ \mathit{true}\,[C]}$$

The local reduction and expansion are similar as in the case of total judgments.

$$\frac{\dfrac{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}\,[C]}{\Gamma \vdash A \to B\ \mathit{true}\,[C]} \qquad \Gamma \vdash A\ \mathit{true}\,[C]}{\Gamma \vdash B\ \mathit{true}\,[C]} \quad \Longrightarrow_R \quad \Gamma \vdash B\ \mathit{true}\,[C]$$

$$\Gamma \vdash A \to B\ \mathit{true}\,[C] \quad \Longrightarrow_E \quad \dfrac{\dfrac{\dfrac{\Gamma \vdash A \to B\ \mathit{true}\,[C]}{\Gamma, A\ \mathit{true} \vdash A \to B\ \mathit{true}\,[C]} \quad \Gamma, A\ \mathit{true} \vdash A\ \mathit{true}\,[C]}{\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}\,[C]}}{\Gamma \vdash A \to B\ \mathit{true}\,[C]}$$

2.1.3 Relativized necessity

As illustrated by the previous sections, dealing with partial judgments and reflection puts a special emphasis on proofs that are *categorical*, i.e. do not depend on any hypotheses. It therefore seems particularly fruitful for the theory of partial judgments, if we could separate the notions of categorical and hypothetical *partial* truth. Such a development will have many important consequences. For one, we could clearly specify that reflection may only be performed over categorical proofs, but not over hypothetical ones. But most importantly, categorical partial truth may be internalized. As described in Section 2.1.1, supports and partial proofs are intended to capture aspects of the meta theory of the truth judgment. If we internalize categorical partial truth, that would provide a way to reason, within the logic itself, about the meta theoretic properties represented by the supports.

Motivated by the need for this distinction, we employ here the theory of modal logic and modal λ-calculus from Section 1.2. The idea is to introduce a separate judgment

$$A\ nec\,[C]$$

of *partial*, or *relativized necessity*, to witness the categorical partial truth $\cdot \vdash A\ true\,[C]$.

The intuition behind necessity in modal logic can be given using the notion of possible worlds (Section 1.1.3). We imagine the existence of a set of worlds, interconnected in some way, so that some worlds are accessible from the others. Any given proposition may be true at a certain world, but need not be true elsewhere. In the hypothetical judgment $\Gamma \vdash A\ true$, the set of antecedents describes the propositions that are known to be true at the current world, and the conclusion A is deemed true at the same world. Therefore, if $A\ nec$, then $\cdot \vdash A\ true$, establishing the truth of A in a generic world that we know nothing about. In other words, if $A\ nec$, then A is true in *all* accessible worlds — necessity is universal quantification over accessible worlds.

The intuition behind the relativized necessity is similar, except that now $A\ nec\,[C]$ is a witness that A is true in *all* accessible worlds in which $C\ holds$. Relativized necessity is *bounded universal quantification* over accessible worlds. The reflection principles can then be viewed as *specialization* of bounded universal quantification. Indeed, if we have a proof that is valid in all worlds where $C\ holds$, by reflection we can modify and specialize it to correspond to the current world.

Just as in Section 1.1.3, the interesting development begins once we introduce hypotheses of relativized necessity, and extend the judgments $\Gamma \vdash C\ holds\,[D]$ and $\Gamma \vdash A\ true\,[C]$ into

$$\Delta; \Gamma \vdash C\ holds\,[D] \quad \text{and} \quad \Delta; \Gamma \vdash A\ true\,[D]$$

where Δ is the *set* of hypotheses $B_1\ nec\,[C_1], \ldots, B_m\ nec\,[C_m]$, and Γ is the *set* of hypotheses $A_1\ true, \ldots, A_n\ true$. Of course, we treat the necessity and truth hypotheses in different ways. Recall from Section 1.1, that the truth hypotheses in the hypothetical judgments are used only in a parametric way, by means of substitutions. We adopt a similar requirement here. Given derivations of $A_1\ true\,[D]$, $\ldots$, $A_n\ true\,[D]$, they may only be *substituted* to obtain derivations of $C\ holds\,[D]$ and $A\ true\,[D]$, respectively. Such a restriction is not imposed on necessity hypotheses. Derivations of $B_1\ nec\,[C_1]$, $\ldots$, $B_m\ nec\,[C_m]$ in fact witness categorical judgments $\cdot \vdash B_1\ true\,[C_1]$, $\ldots$, $\cdot \vdash B_m\ true\,[C_m]$, and may therefore be *reflected* to obtain derivations of $\Gamma \vdash C\ holds\,[D]$ and $\Gamma \vdash A\ true\,[D]$.

Because relativized necessity is defined via the notion of partial truth, we do not require a separate judgment for hypothetical relativized necessity $\Delta \vdash A\ nec\,[C]$. It can already be expressed as $\Delta; \cdot \vdash A\ true\,[C]$.

The support weakening principle for the new judgment is a straightforward extension of the principle from the previous section.

Principle (Support weakening)
Let C and D be well-formed supports with $C \sqsubseteq D$. Then the following holds:

1. if $\Delta; \Gamma \vdash A\ true\,[C]$, then $\Delta; \Gamma \vdash A\ true\,[D]$

2. if $\Delta; \Gamma \vdash C_1\ holds\,[C]$, then $\Delta; \Gamma \vdash C_1\ holds\,[D]$

3. if $\Delta; \Gamma \vdash D\ holds\,[D_1]$, then $\Delta; \Gamma \vdash C\ holds\,[D_1]$

The extensions of the reflection principle still allows reflection to be perform only over derivations that are obtained in a *categorical* way. In the judgments $\Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$ and $\Delta; \Gamma \vdash A\ \mathit{true}\,[D]$, a derivation is categorical if it does not use the ordinary truth hypotheses from Γ. However, a categorical derivation may use hypotheses from Δ, because the hypotheses from Δ themselves stand for other categorical derivations. This leads to the following reflection principle.

Principle (Reflection)
If $\Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$, then the following holds:

1. if $\Delta; \cdot \vdash A\ \mathit{true}\,[C]$, then $\Delta; \Gamma \vdash A\ \mathit{true}\,[D]$

2. if $\Delta; \cdot \vdash C_1\ \mathit{holds}\,[C]$, then $\Delta; \Gamma \vdash C_1\ \mathit{holds}\,[D]$

In the axiomatization of the judgment $\Delta; \Gamma \vdash A\ \mathit{true}\,[C]$, the hypothetical nature of the judgment with respect to relativized necessity is made explicit by the hypothesis rule below.

$$\frac{(\Delta, A\ \mathit{nec}[C]); \Gamma \vdash C\ \mathit{holds}\,[D]}{(\Delta, A\ \mathit{nec}[C]); \Gamma \vdash A\ \mathit{true}\,[D]}$$

The rule is justified on the following grounds: a proof of $A\ \mathit{nec}\,[C]$ is a proof of the categorical judgment $\cdot \vdash A\ \mathit{true}\,[C]$, and hence may be reflected into a proof of $A\ \mathit{true}\,[D]$, given the evidence of $C\ \mathit{holds}\,[D]$. The corresponding substitution principle follows the definition of the hypothetical judgment.

Principle (Substitution for relativized necessity)
If $\Delta; \cdot \vdash A\ \mathit{true}\,[C]$, then the following holds:

1. if $(\Delta, A\ \mathit{nec}[C]); \Gamma \vdash B\,[D]$, then $\Delta; \Gamma \vdash B\,[D]$

2. if $(\Delta, A\ \mathit{nec}[C]); \Gamma \vdash D'\ \mathit{holds}\,[D]$, then $\Delta; \Gamma \vdash D'\ \mathit{holds}\,[D]$

We refer to this principle as a substitution principle, even though, strictly speaking, there is no requirement that the derivation of $\Delta; \cdot \vdash A\ \mathit{true}\,[C]$ must, in fact, be used unmodified. The reason for this terminology is that, while categorical proofs may be modified by reflection, the reflection is really the only operation that may be used for this purpose. Therefore, we may still consider the judgments parametric in necessity hypotheses, except that the concept of a parametricity is now extended to admit a *limited and well-specified* way to alter derivations. [2]

Finally, we internalize the judgment of relativized necessity into the truth judgment, by introducing a new operator on propositions $\Box$. Unlike in Section 1.1.3, this time we have a whole family $\Box_C$ operators, in order to express bounded universal quantification over accessible worlds. When the support C is 0, we will simply write $\Box A$ instead of $\Box_0 A$. The formation rule for the $\Box_C$ operator is as follows:

$$\frac{A\ \mathit{prop} \qquad C\ \mathit{supp}}{\Box_C A\ \mathit{prop}}$$

[2] The following analogy may be illustrative. The full-fledged parametricity of truth hypotheses requires that the corresponding proofs be used as *black boxes.* The proofs can be substituted into desired positions, but they must remain unmodified. On the other hand, proofs of necessity hypotheses are black boxes whose functionality may be controlled by a well-specified interface C, but by no other means.

with the introduction and elimination rules similar as before, but this time indexed by supports.

$$\dfrac{\Delta;\cdot\vdash A\ \mathit{true}\,[C]}{\Delta;\Gamma\vdash \Box_C A\ \mathit{true}\,[D]}\qquad\qquad \dfrac{\Delta;\Gamma\vdash \Box_C A\ \mathit{true}\,[D]\qquad (\Delta, A\ \mathit{nec}[C]);\Gamma\vdash B\ \mathit{true}\,[D]}{\Delta;\Gamma\vdash B\ \mathit{true}\,[D]}$$

While the elimination rule above is justified simply on the grounds of the substitution principle for necessary hypothesis, it is the introduction rule that is interesting, as it embodies the definition of the relativized necessity. Indeed, $\Box_C A$ is true if and only if $A\ \mathit{true}\,[C]$ can be proved categorically. This motivates the erasure of the context Γ from the premise of the rule. In contrast, notice that the support C persists in the judgment. Unlike Γ which represents hypotheses that are local to the current world, the support condition C has global nature. On the other hand, while the conclusion $\Box_C A$ is obtained in a total way, we allow weakening with an arbitrary support D in order to conform with the support weakening principle.

Local soundness is justified on the grounds of the substitution principle for relativized necessity.

$$\dfrac{\dfrac{\Delta;\cdot\vdash A\ \mathit{true}\,[C]}{\Delta;\cdot\vdash \Box_C A\ \mathit{true}\,[D]}\quad (\Delta, A\ \mathit{nec}[C]);\Gamma\vdash B\ \mathit{true}\,[D]}{\Delta;\Gamma\vdash B\ \mathit{true}\,[D]}\quad\Longrightarrow_R\quad \Delta;\Gamma\vdash B\ \mathit{true}\,[D]$$

Local completeness is witnessed by the local expansion similar to Section 1.1.3.

$$\Delta;\Gamma\vdash \Box_C A\ \mathit{true}\,[D]\quad\Longrightarrow_E\quad \dfrac{\Delta;\Gamma\vdash \Box_C A\ \mathit{true}\,[D]\quad \dfrac{\dfrac{\dfrac{C\sqsubseteq C}{(\Delta, A\ \mathit{nec}[C]);\cdot\vdash C\ \mathit{holds}\,[C]}}{(\Delta, A\ \mathit{nec}[C]);\cdot\vdash A\ \mathit{true}\,[C]}}{(\Delta, A\ \mathit{nec}[C]);\Gamma\vdash \Box_C A\ \mathit{true}\,[D]}}{\Delta;\Gamma\vdash \Box_C A\ \mathit{true}\,[D]}$$

Note that the local expansion employs the following rule of the support judgment

$$\dfrac{C\sqsubseteq D}{\Delta;\Gamma\vdash C\ \mathit{holds}\,[D]}$$

to derive that $(\Delta, A\ \mathit{nec}[C]);\cdot\vdash C\ \mathit{holds}\,[C]$.

Example 7 Let C and D be well-formed supports such that $C\sqsubseteq D$. Then the following derivation (which we denote by $\mathcal{D}^A_{C,D}$) is a valid derivation of the judgment $(\Delta, A\ \mathit{nec}[C]);\Gamma\vdash A\ \mathit{true}\,[D]$.

$$\dfrac{\dfrac{C\sqsubseteq D}{(\Delta, A\ \mathit{nec}[C]);\Gamma\vdash C\ \mathit{holds}\,[D]}}{(\Delta, A\ \mathit{nec}[C]);\Gamma\vdash A\ \mathit{true}\,[D]}$$

We next use this derivation to establish that $\Delta;\Gamma\vdash \Box_C A\rightarrow\Box_D A\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{}{\Delta;(\Gamma,\Box_C A\ \mathit{true})\vdash \Box_C A\ \mathit{true}}\quad \dfrac{\begin{array}{c}\mathcal{D}^A_{C,D}\\ (\Delta, A\ \mathit{nec}[C]);\cdot\vdash A\ \mathit{true}[D]\end{array}}{(\Delta, A\ \mathit{nec}[C]);\Gamma\vdash \Box_D A\ \mathit{true}}}{\Delta;(\Gamma,\Box_C A\ \mathit{true})\vdash \Box_D A\ \mathit{true}}}{\Delta;\Gamma\vdash \Box_C A\rightarrow\Box_D A\ \mathit{true}}$$

We also establish the support-decorated versions of the customary axioms of constructive modal logic S4 (Section 1.1.3):

1. $\Delta;\Gamma \vdash \Box_C A \to A\ \mathit{true}\,[D]$
2. $\Delta;\Gamma \vdash \Box_C A \to \Box\Box_C A\ \mathit{true}$
3. $\Delta;\Gamma \vdash \Box_C(A \to B) \to \Box_C A \to \Box_C B\ \mathit{true}$

1. Derivation of $\Delta;\Gamma \vdash \Box_C A \to A\ \mathit{true}\,[D]$.

$$\dfrac{\dfrac{\Delta;(\Gamma,\Box_C A\ \mathit{true}) \vdash \Box_C A\ \mathit{true}\,[D] \quad \begin{array}{c}\mathcal{D}^A_{C,D}\\ (\Delta, A\ \mathit{nec}[C]);\Gamma \vdash A\ \mathit{true}\,[D]\end{array}}{\Delta;(\Gamma,\Box_C A\ \mathit{true}) \vdash A\ \mathit{true}\,[D]}}{\Delta;\Gamma \vdash \Box_C A \to A\ \mathit{true}\,[D]}$$

2. Derivation of $\Delta;\Gamma \vdash \Box_C A \to \Box\Box_C A\ \mathit{true}$.

$$\dfrac{\dfrac{\Delta;(\Gamma,\Box_C A\ \mathit{true}) \vdash \Box_C A\ \mathit{true} \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}^A_{C,C}\\ (\Delta, A\ \mathit{nec}[C]);\cdot \vdash A\ \mathit{true}\,[C]\end{array}}{(\Delta, A\ \mathit{nec}[C]);\cdot \vdash \Box_C A\ \mathit{true}}}{(\Delta, A\ \mathit{nec}[C]);\Gamma \vdash \Box\Box_C A\ \mathit{true}}}{\Delta;(\Gamma,\Box_C A\ \mathit{true}) \vdash \Box\Box_C A\ \mathit{true}}}{\Delta;\Gamma \vdash \Box_C A \to \Box\Box_C A\ \mathit{true}}$$

3. Derivation of $\Delta;\Gamma \vdash \Box_C(A \to B) \to \Box_C A \to \Box_C B\ \mathit{true}$. To reduce clutter, we define $\Gamma_1 = \Gamma, \Box_C(A \to B)\ \mathit{true}$, $\Delta_1 = \Delta, (A \to B)\ \mathit{nec}[C]$, $\Gamma_2 = \Gamma_1, \Box_C A\ \mathit{true}$, and $\Delta_2 = \Delta_1, A\ \mathit{nec}[C]$. We proceed by splitting the derivation of $\Delta;\Gamma \vdash \Box_C(A \to B) \to \Box_C A \to \Box_C B\ \mathit{true}$ into two parts. First, we obtain a derivation $\mathcal{D}'$ to establish that $\Delta_2;\Gamma_2 \vdash \Box_C B\ \mathit{true}$.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}^{A\to B}_{C,C}\\ \Delta_2;\cdot \vdash A \to B\ \mathit{true}\,[C]\end{array} \quad \begin{array}{c}\mathcal{D}^A_{C,C}\\ \Delta_2;\cdot \vdash A\ \mathit{true}\,[C]\end{array}}{\Delta_2;\cdot \vdash B\ \mathit{true}\,[C]}}{\Delta_2;\Gamma_2 \vdash \Box_C B\ \mathit{true}}$$

Then $\mathcal{D}'$ can be used to obtain a derivation of $\Delta;\Gamma \vdash \Box_C(A \to B) \to \Box_C A \to \Box_C B\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\Delta;\Gamma_2 \vdash \Box_C(A \to B)\ \mathit{true} \quad \dfrac{\Delta_1;\Gamma_2 \vdash \Box_C A\ \mathit{true} \quad \begin{array}{c}\mathcal{D}'\\ \Delta_2;\Gamma_2 \vdash \Box_C B\ \mathit{true}\end{array}}{\Delta_1;\Gamma_2 \vdash \Box_C B\ \mathit{true}}}{\Delta;\Gamma_2 \vdash \Box_C B\ \mathit{true}}}{\Delta;\Gamma_1 \vdash \Box_C A \to \Box_C B\ \mathit{true}}}{\Delta;\Gamma \vdash \Box_C(A \to B) \to \Box_C A \to \Box_C B\ \mathit{true}}$$

■

2.1.4 Simultaneous possibility

The dual concepts to bounded universal quantification and relativized necessity, are of course, bounded existential quantification, and the related notion of *simultaneous possibility.* Where relativized necessity expresses that a proposition A is true in *all worlds* in which C *holds*, simultaneous possibility expresses that there *exists a world* in which C *holds and also* A *true.* In order to formalize the notion of simultaneous possibility, we introduce a new judgment $\langle C, A\rangle$ *poss*, and immediately generalize it to its partial and hypothetical variant

$$\Delta; \Gamma \vdash \langle C, A\rangle \ \mathit{poss}\,[D]$$

When C is the 0 support, we omit it from the notation and abbreviate simply as $\Delta; \Gamma \vdash A \ \mathit{poss}\,[D]$. The intuition behind this judgment is to establish the derivability of both $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ and $\Delta; \Gamma \vdash A \ \mathit{true}\,[D]$, but where the second derivation may be obtained by means of reflection using the first derivation.

Being intuitively specified in terms of C *holds* and A *true*, the new judgment is required to satisfy similar weakening, reflection and substitution principles.

Principle (Support weakening)
If $\Delta; \Gamma \vdash \langle C_1, A\rangle \ \mathit{poss}\,[C]$ and $C \sqsubseteq D$, then $\Delta; \Gamma \vdash \langle C_1, A\rangle \ \mathit{poss}\,[D]$.

Principle (Reflection)
If $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ and $\Delta; \cdot \vdash \langle C_1, A\rangle \ \mathit{poss}\,[C]$, then $\Delta; \Gamma \vdash \langle C_1, A\rangle \ \mathit{poss}\,[D]$.

Principle (Substitution for truth)
If $\Delta; \Gamma \vdash A \ \mathit{true}\,[C]$ and $\Delta; (\Gamma, A \ \mathit{true}) \vdash \langle D, B\rangle \ \mathit{poss}\,[C]$, then $\Delta; \Gamma \vdash \langle D, B\rangle \ \mathit{poss}\,[C]$.

Principle (Substitution for relativized necessity)
If $\Delta; \cdot \vdash A \ \mathit{true}\,[C]$ and $(\Delta, A \ \mathit{nec}[C]); \Gamma \vdash \langle C_1, B\rangle \ \mathit{poss}\,[D]$, then $\Delta; \Gamma \vdash \langle C_1, B\rangle \ \mathit{poss}\,[D]$.

There are four ways simultaneous possibility can be established, giving raise to four basic definitional principles.

1. If $\Delta; \Gamma \vdash A \ \mathit{true}\,[C]$, then $\Delta; \Gamma \vdash A \ \mathit{poss}\,[C]$.
2. If $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ and $\Delta; \cdot \vdash A \ \mathit{true}\,[C]$, then $\Delta; \Gamma \vdash \langle C, A\rangle \ \mathit{poss}\,[D]$.
3. If $\Delta; \Gamma \vdash \langle C_1, A\rangle \ \mathit{poss}\,[D]$ and $\Delta; A \ \mathit{true} \vdash B \ \mathit{true}\,[C_1]$, then $\Delta; \Gamma \vdash \langle C_1, B\rangle \ \mathit{poss}\,[D]$.
4. If $\Delta; \Gamma \vdash \langle C_1, A\rangle \ \mathit{poss}\,[D]$ and $\Delta; A \ \mathit{true} \vdash \langle C_2, B\rangle \ \mathit{poss}\,[C_1]$, then $\Delta; \Gamma \vdash \langle C_2, B\rangle \ \mathit{poss}\,[D]$.

Principle (1) is justified by the fact that $\Delta; \Gamma \vdash 0 \ \mathit{holds}\,[C]$ always trivially holds. Taken together with the assumed $\Delta; \Gamma \vdash A \ \mathit{true}\,[C]$, this ensures that the two judgments simultaneously hold in the *current world*, and are therefore simultaneously possible.

To justify principle (2), observe that given $C \ \mathit{holds}\,[D]$ and $A \ \mathit{true}\,[C]$, we can obtain $A \ \mathit{true}\,[D]$ by reflection. The derivations are in the *current world*, and are therefore simultaneously true. The required reflection, however, can only be performed if $A\ true\,[C]$ is derived in a categorical way. Hence the restriction that the judgment $\Delta; \cdot \vdash A\ true\,[C]$ uses no truth hypotheses.

Principle (3) is justified by the following observation: if C_1 *holds* and A *true* are simultaneously possible, then there exists a world about which we know nothing, except that C_1 *holds* and A *true* can be derived in it. If we can use these two facts, but nothing else, to conclude that B *true* in the very same world, then certainly C_1 *holds* and B *true* are simultaneously true in this world, and are therefore simultaneously possible. If the possibility of C_1 *holds* and A *true* is partial in D, so would be the concluded possibility of C_1 *holds* and B *true*.

The reasoning behind the principle (4) is similar. If C_1 *holds* and A *true* are simultaneously possible in some world, and we can use these two facts, but nothing else, to conclude the simultaneous possibility of C_2 *holds* and B *true*, then the later two are certainly possible. If the possibility of C_1 *holds* and A *true* is partial in D, so is the concluded possibility of C_2 *holds* and B *true*.

In order to internalize simultaneous possibility of C *holds* and A *true*, we introduce the indexed family of operators $\Diamond_C A$ for bounded existential quantification over possible worlds. When the support C is 0, we will simply write $\Diamond A$ instead of $\Diamond_0 A$. The appropriate formation rule is

$$\frac{A\ prop \qquad C\ supp}{\Diamond_C A\ prop}$$

and the introduction rule defines the operator as an internalization of simultaneous possibility.

$$\frac{\Delta; \Gamma \vdash \langle C, A\rangle\ poss\,[D]}{\Delta; \Gamma \vdash \Diamond_C A\ true\,[D]}$$

The axiomatization of the possibility judgment itself reflects the definitional principles outlined previously. For example, the principles (1) and (2) are directly translated into the following derivation rules.

$$\frac{\Delta; \Gamma \vdash A\ true\,[C]}{\Delta; \Gamma \vdash A\ poss\,[C]} \qquad \frac{\Delta; \Gamma \vdash C\ holds\,[D] \qquad \Delta; \cdot \vdash A\ true\,[C]}{\Delta; \Gamma \vdash \langle C, A\rangle\ poss\,[D]}$$

There are two elimination rules for $\Diamond_C$, arising from the definitional principles (3) and (4). However, instead of the hypothesis $\langle C_1, A\rangle$ *poss*, these rules use the internalized version $\Diamond_{C_1} A$ *true*.

$$\frac{\Delta; \Gamma \vdash \Diamond_{C_1} A\ true\,[D] \qquad \Delta; A\ true \vdash B\ true\,[C_1]}{\Delta; \Gamma \vdash \langle C_1, B\rangle\ poss\,[D]}$$

$$\frac{\Delta; \Gamma \vdash \Diamond_{C_1} A\ true\,[D] \qquad \Delta; A\ true \vdash \langle C_2, B\rangle\ poss\,[C_1]}{\Delta; \Gamma \vdash \langle C_2, B\rangle\ poss\,[D]}$$

Local soundness is established by two local reduction, which are justified by the definitional principles (3) and (4). Local completeness and local expansion are also simple to verify.

$$\frac{\dfrac{\Delta; \Gamma \vdash \langle C_1, A\rangle\ poss\,[D]}{\Delta; \Gamma \vdash \Diamond_{C_1} A\ true\,[D]} \qquad \Delta; A\ true \vdash B\ true\,[C_1]}{\Delta; \Gamma \vdash \langle C_1, B\rangle\ poss\,[D]} \quad \Longrightarrow_R \quad \Delta; \Gamma \vdash \langle C_1, B\rangle\ poss\,[D]$$

$$\dfrac{\dfrac{\Delta;\Gamma\vdash\langle C_1,A\rangle\ \mathit{poss}\,[D]}{\Delta;\Gamma\vdash\Diamond_{C_1}A\ \mathit{true}\,[D]} \qquad \Delta;A\ \mathit{true}\vdash\langle C_2,B\rangle\ \mathit{poss}\,[C_1]}{\Delta;\Gamma\vdash\langle C_2,B\rangle\ \mathit{poss}\,[D]} \quad \Longrightarrow_R \quad \Delta;\Gamma\vdash\langle C_2,B\rangle\ \mathit{poss}\,[D]$$

$$\Delta;\Gamma\vdash\Diamond_C A\ \mathit{true}\,[D] \quad \Longrightarrow_E \quad \dfrac{\dfrac{\Delta;\Gamma\vdash\Diamond_C A\ \mathit{true}\,[D] \quad \Delta;A\ \mathit{true}\vdash A\ \mathit{true}\,[C]}{\Delta;\Gamma\vdash\langle C,A\rangle\ \mathit{poss}\,[D]}}{\Delta;\Gamma\vdash\Diamond_C A\ \mathit{true}\,[D]}$$

Finally, similar to Section 1.1.4, we also have the additional rule for eliminating $\Box_C$ in the new possibility judgment

$$\dfrac{\Delta;\Gamma\vdash\Box_C A\ \mathit{true}\,[D] \qquad (\Delta,A\ \mathit{nec}[C]);\Gamma\vdash\langle C_2,B\rangle\ \mathit{poss}\,[D]}{\Delta;\Gamma\vdash\langle C_2,B\rangle\ \mathit{poss}\,[D]}$$

Example 8 Let C, C_1 and D be well-formed supports such that $C\sqsubseteq D$. Then the following are support-decorated versions of the customary axioms of constructive modal logic S4 (Section 1.1.4):

1. $\Delta;\Gamma\vdash A\to\Diamond A\ \mathit{true}$
2. $\Delta;\Gamma\vdash\Diamond_{C_1}\Diamond_C A\to\Diamond_C A\ \mathit{true}$
3. $\Delta;\Gamma\vdash\Box_C(A\to B)\to\Diamond_D A\to\Diamond_D B\ \mathit{true}$

The derivations proceed as follows.

1. $$\dfrac{\dfrac{\dfrac{\Delta;(\Gamma,A\ \mathit{true})\vdash A\ \mathit{true}}{\Delta;(\Gamma,A\ \mathit{true})\vdash A\ \mathit{poss}}}{\Delta;(\Gamma,A\ \mathit{true})\vdash\Diamond A\ \mathit{true}}}{\Delta;\Gamma\vdash A\to\Diamond A\ \mathit{true}}$$

2. To reduce clutter, we define $\Gamma_1=\Gamma,\Diamond_{C_1}\Diamond_C A\ \mathit{true}$. Then we have the following inference.

$$\dfrac{\dfrac{\dfrac{\Delta;\Gamma_1\vdash\Diamond_{C_1}\Diamond_C A\ \mathit{true} \qquad \dfrac{\Delta;\Diamond_C A\ \mathit{true}\vdash\Diamond_C A\ \mathit{true}\,[C_1] \quad \Delta;A\ \mathit{true}\vdash A\ \mathit{true}\,[C]}{\Delta;\Diamond_C A\ \mathit{true}\vdash\langle C,A\rangle\ \mathit{poss}\,[C_1]}}{\Delta;\Gamma_1\vdash\langle C,A\rangle\ \mathit{poss}}}{\Delta;\Gamma_1\vdash\Diamond_C A\ \mathit{true}}}{\Delta;\Gamma\vdash\Diamond_{C_1}\Diamond_C A\to\Diamond_C A\ \mathit{true}}$$

3. In this case, we introduce the following abbreviations: $\Gamma_1=\Gamma,\Box_C(A\to B)\ \mathit{true}$, $\Gamma_2=\Gamma_1,\Diamond_D A\ \mathit{true}$, and $\Delta_1=\Delta,(A\to B)\ \mathit{nec}[C]$. We also recall the inference $\mathcal{D}^{A\to B}_{C,D}$ of the

judgment $\Delta_1;\Gamma \vdash A \to B\ \mathit{true}\,[D]$ derived in Example 7. The proof is now split into two parts. We first obtain the derivation $\mathcal{D}'$ of the judgment $\Delta_1;\Gamma_2 \vdash \langle B, D\rangle\ \mathit{poss}$.

$$\dfrac{\Delta_1;\Gamma_2 \vdash \Diamond_D A\ \mathit{true} \qquad \dfrac{\overset{\mathcal{D}^{A\to B}_{C,D}}{\Delta_1; A\ \mathit{true} \vdash A \to B\ \mathit{true}\,[D]} \qquad \Delta_1; A\ \mathit{true} \vdash A\ \mathit{true}\,[D]}{\Delta_1; A\ \mathit{true} \vdash B\ \mathit{true}\,[D]}}{\Delta_1;\Gamma_2 \vdash \langle B, D\rangle\ \mathit{poss}}$$

The derivation $\mathcal{D}'$ can now be used to infer the required $\Delta;\Gamma \vdash \Box_C(A \to B) \to \Diamond_D A \to \Diamond_D B\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\Delta;\Gamma_2 \vdash \Box_C(A \to B)\ \mathit{true} \qquad \dfrac{\overset{\mathcal{D}'}{\Delta_1;\Gamma_2 \vdash \langle B, D\rangle\ \mathit{poss}}}{\Delta_1;\Gamma_2 \vdash \Diamond_D B\ \mathit{true}}}{\Delta;\Gamma_2 \vdash \Diamond_D B\ \mathit{true}}}{\Delta;\Gamma_1 \vdash \Diamond_D A \to \Diamond_D B\ \mathit{true}}}{\Delta;\Gamma \vdash \Box_C(A \to B) \to \Diamond_D A \to \Diamond_D B\ \mathit{true}}$$

■

2.1.5 Names

One possible way to specify the notion of support for the modal logic of partial judgments from Section 2.1.1 is by using *names*. Names are elements of a countable universe $\mathcal{N}$, and will be used as labels witnessing a certain fact about the derivability of truth judgments. Every name from $\mathcal{N}$ is associated with some proposition, and for each proposition itself there is a countable number of names associated with it. When the name X is associated with the proposition A, we will write that as[3]

$$\mathit{typeof}(X) = A.$$

The semantics of this relation between X and A may be defined in various ways. For example, a particularly simple definition – and this is the semantics of names that we consider in this chapter – is to associate X with the *existence* of a derivation of $A\ \mathit{true}$.

Having introduced names, we define the notion of support as a *finite set* of names. If the support C consists of names $X_1, \ldots, X_n$, then the condition represented by C is the *conjunction* of the properties represented by each of the names. For example, if $X_1, \ldots, X_n$ are associated with propositions $A_1, \ldots, A_n$, respectively, then the whole support C stands for the meta theoretic statement that the judgments $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$ are all derivable. In such a case, the partial judgment

$$A\ \mathit{true}\,[X_1, \ldots, X_n]$$

simply expresses the fact that A is true, given the derivability of $A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}$. Obviously, in this formulation names very much resemble ordinary variables in hypothetical judgments from Chapter 1, but there are several notable *proof-theoretic* distinctions between the two. First of all, ordinary variables in the hypothetical judgments have local nature. In terminology of modal

[3] Should we insist that this relation is well-founded? Seems unnecessary, though once Σ is introduced, well-foundedness is enforced.

logic, variables represent proofs that are meaningful only in the current world. On the other hand, names are global and their identity persists across the worlds. This will be significant in the later developments of the calculus. For example, in Section 3.3 we will consider polymorphism in supports, and universally quantify over arbitrary finite sets of names. A similar quantification over parts of local contexts would result in impredicativity of the calculus. However, the most significant distinction between names and variables involves the process of reflection. The only way a hypothetical proof depending on A_1 *true*, ... , A_n *true* may be used is by substituting the proofs of A_1 *true*, ... , A_n *true* when these proofs are available. In contrast, a categorical proof that is partial in $X_1, \ldots, X_n$ may be modified in an arbitrary way, as specified by reflection.

We further remark on a particular aspect illustrating the versatility of names, which we will not explore in this chapter. While a *variable* of type A in a hypothetical judgment *must* stand for a derivability of the judgment A *true*, such a requirement is not enforced upon names. It is possible that a name X with $\mathit{typeof}(X) = A$ may stand for the derivations of A *true* that satisfy particular properties or invariants. For example, X may represent that A *true* is provable using only introduction, or using only elimination rules. Or, for that matter, X may even stand for the fact that $\not\vdash A$ *true*. Combined with modalities and reflection, this provides a way to encode arbitrary aspects of the meta reasoning about derivability, for a number of different meta logics.

Having defined the universe of supports as the set $\mathcal{P}_{\mathit{fin}}(\mathcal{N})$, we also need to establish a partial ordering on it. For the purposes of this section, if C and D are two supports, we will consider

$$C \sqsubseteq D \qquad \text{if and only if} \qquad C \subseteq D$$

Then the empty support set, which we denote by $(\cdot)$ is the minimal support in this ordering. In addition, we will have the following formation rules for supports.

$$\frac{}{\cdot \ \mathit{supp}} \qquad \frac{C \ \mathit{supp}}{C, X \ \mathit{supp}}$$

The axiomatization of the judgments $\Delta; \Gamma \vdash A \ \mathit{true}\,[C]$ and $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ now proceeds in a mutually recursive way. The most important rule is

$$\frac{\mathit{typeof}(X) = A}{\Delta; \Gamma \vdash A \ \mathit{true}\,[C, X]}\,(*)$$

specifying that if the name X witnesses the derivability of A *true*, then we can certainly conclude that A is true partially in X. Notice that we allow weakening with an arbitrary support set C in the conclusion, in order to give rise to the support weakening principle. On the other hand, $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ is axiomatized using the following two rules:

$$\frac{C \subseteq D}{\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]} \qquad \frac{\Delta; \Gamma \vdash A \ \mathit{true}\,[D] \quad \Delta; \Gamma \vdash (C \setminus X) \ \mathit{holds}\,[D] \quad \mathit{typeof}(X) = A}{\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]}$$

where we denote by $C \setminus X$ the set-difference between C and $\{X\}$. The first of the above rules serves to establish the basic causal dependence between supports – if D represents a stronger condition than C, then trivially C *holds* $[D]$. The second rule formalizes that the support C actually represents the *conjunction* of the conditions associated with the names in C. Indeed, if C consists of names $X_1, \ldots, X_n$, where $\mathit{typeof}(X_i) = A_i$, then $\Delta; \Gamma \vdash C \ \mathit{holds}\,[D]$ if and only if $\Delta; \Gamma \vdash A_i \ \mathit{true}\,[D]$ for every $i = 1, \ldots, n$.

2.1.6 Name-space management

A notable feature of the formulation of partial judgments from the previous section is the global nature of names. Names are given once and for all, and are shared by all the worlds. For computational purposes, however, it is beneficial to introduce the notion of local names. Local names can dynamically be generated during the derivation; each generated name is fresh, i.e. different from all the names generated so far. Also, each local name will have a scope within which it can be used, and outside of which it is inaccessible.

In order to deal with the freshness of local names, we make the judgments hypothetical in a yet another context – the context generated names. This context will associate each generated name with its type. For example, the new truth judgment will now have the form

$$\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$$

where Σ consists of $X_1{:}A_1, \dots, X_n{:}A_n$, associating the names X_1, ... , X_n with propositions A_1, ... , A_n, respectively. We denote by $\mathrm{dom}(\Sigma)$ the set of names $\{X_1, \dots, X_n\}$. Notice that Σ is a dependently typed context, because each proposition may itself depend on names. Henceforth, we impose on Σ the typical requirements of dependent contexts. In particular, we assume that the names $X_1, \dots, X_n$ are all different, and that each X_i may be used only to the right of its declaration. For example, the name X_1 may appear in the propositions $A_2, \dots, A_n$, as well as in Δ, Γ, A and C, but not in A_1. The name X_2 may not appear in A_1 and A_2, but may appear elsewhere, and so on. Furthermore, we insist that a name can be used in this judgment only if it is actually declared in the name context Σ. Thus, we rephrase the rule $(*)$ of the truth judgment from the previous section, which now has the form:

$$\frac{X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C, X]}$$

While we insist that the judgment $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ is *well-formed* only if all its names are declared in Σ, we allow a bit more leeway in defining what counts as a proof of $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$. In particular, the intended meaning of $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ is that there exists a name context Σ' extending Σ and a proof for $\Delta; \Gamma \vdash A\ \mathit{true}\,[C]$, such that the names contained in this proof are declared in Σ' (even though Δ, Γ, A, and C must still use only the names from Σ, in order to be well-formed). In this sense, a proof of the judgment $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ will be a *pair* consisting of both Σ' and a proof of $\Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ satisfying the above requirement.[4]

Notice that the outlined semantics of name contexts to serve as lists of currently generated names does allow the following structural properties. Here we use J as an abbreviation for the $\Delta; \Gamma \vdash A\ \mathit{true}\,[C]$, and $\Sigma \vdash J$ as an abbreviation for $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$.

1. *Strengthening.* If $(\Sigma, X{:}A) \vdash J$, and X is not used in J, then $\Sigma \vdash J$.

   Indeed, if X is not used in J, then $\Sigma \vdash J$ is well-formed. Furthermore any context $\Sigma' \supseteq (\Sigma, X{:}A)$ is also $\Sigma' \supseteq \Sigma$, and thus a proof of $(\Sigma, X{:}A) \vdash J$ is also a proof of $\Sigma \vdash J$.

2. *Renaming.* If $(\Sigma, X{:}A, \Sigma') \vdash J$ and the name Y is not used in Σ, Σ', A, or J, then $(\Sigma, Y{:}A, [Y/X]\Sigma') \vdash ([Y/X]J)$.

3. *Weakening.* If $\Sigma \vdash J$, and X is not used in J, then $(\Sigma, X{:}A) \vdash J$.

[4] I.e. the proof may use the names from Σ'.

This principle is justified on the grounds of the previous principle for renaming. Indeed, if $\Sigma \vdash J$, then there exists a name context $\Sigma' \supseteq \Sigma$ and a proof of J using Σ'. If Σ' does not declare X, then $\Sigma', X{:}A$ is a well-formed name context and the proof of J uses $\Sigma', X{:}A$. If Σ' declared X then we can rename that occurrence of X in both Σ' and the supplied proof of J.

4. *Exchange.* Permutation of name contexts is allowed if it does not violate the dependencies between names and the propositions associated with them. In other words, if $(\Sigma, X{:}A, \Sigma', \Sigma'') \vdash J$, and X is not used in Σ', then $(\Sigma, \Sigma', X{:}A, \Sigma'') \vdash J$. In the rest of this document, we will abuse the notation and treat name contexts as if they were sets. In particular, we consider Σ' and Σ to be equal if they only differ by a dependency-preserving reordering. Similarly, we write $\Sigma' \supseteq \Sigma$, if Σ' extends Σ (with possible name reordering).

Notice however that contraction is not something we require of a name context. We want to preserve the distinction between names: if the judgment B *true* is derived by *reflection* using two different names $X{:}A$ and $Y{:}A$, there is no requirement that the same derivation is produced if X and Y are simultaneously renamed into some new name $Z{:}A$. In accord with the renaming principle, both X and Y may individually be renamed into Z, but not at the same time.

The judgments $\Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$ and $\Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]$ are extended with Σ in a similar way. For example, the rules for introduction and elimination of implication in the truth judgment now have the form

$$\frac{\Sigma; \Delta; (\Gamma, A\ \mathit{true}) \vdash B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A \to B\ \mathit{true}\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash A \to B\ \mathit{true}\,[C] \qquad \Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}$$

The elimination rule deserves further discussion. From the premises, we know that there exist name contexts $\Sigma_1 \supseteq \Sigma$, $\Sigma_2 \supseteq \Sigma$ such that the proof of $\Delta; \Gamma \vdash A \to B\,[C]$ uses only Σ_1, and the proof of $\Delta; \Gamma \vdash A\,[C]$ uses only Σ_2. By the substitution principle for truth, we may then produce a derivation of $\Delta; \Gamma \vdash B\ \mathit{true}\,[C]$, which uses the names from Σ_1 and Σ_2. This derivation, together with the context Σ_3 obtained as a union of Σ_1 and Σ_2, is a witness of $\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$. [5]

We also need to account for Σ in the judgments for formation of supports and propositions, and extend then into $\Sigma \vdash C\ \mathit{supp}$ and $\Sigma \vdash A\ \mathit{prop}$. The relevant rules of the new judgments are listed below.

$$\frac{}{\Sigma \vdash \cdot\ \mathit{supp}} \qquad \frac{\Sigma \vdash C\ \mathit{supp} \qquad X \in \mathrm{dom}(\Sigma)}{\Sigma \vdash C, X\ \mathit{supp}}$$

$$\frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Box_C A\ \mathit{prop}} \qquad \frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Diamond_C A\ \mathit{prop}}$$

As customary, we will implicitly assume that the proposition and supports in our judgments for truth, necessity and possibility are always well-formed according to the above rules.

The next step in the axiomatization of the judgment $\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$, is to internalize the dependence of the conclusion B *true* on names from Σ. With that goal, we introduce a new

[5] We need not rename here, as we assume that each name is associated with a type right from the beginning. Thus, Σ_1 and Σ_2 will coherently ascribe types to names.

constructor on propositions $A \not\twoheadrightarrow B$, with the following formation rule.

$$\frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash B\ \mathit{prop}}{\Sigma \vdash A \not\twoheadrightarrow B\ \mathit{prop}}$$

The judgment $A \not\twoheadrightarrow B\ \mathit{true}$ should be provable if and only if $B\ \mathit{true}$ can be proved using an *arbitrary fresh* name of type A. In other words, we have the following introduction rule.

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]}$$

In this rule we assume that X is fresh, i.e. X does not appear in Σ, Δ, Γ, A, B, or C. Notice that the exact identity of the name X is irrelevant, as long as X is one of the *unused* names with $\mathit{typeof}(X) = A$. Indeed, by the renaming principle for names, any chosen fresh name would have produced the same derivation. Furthermore, because X does not appear in Σ, Δ, Γ, A, B, or C, it *remains local to the proof* of $\Delta; \Gamma \vdash B\ \mathit{true}\,[C]$.

If we can prove $\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]$, then there exists a proof of $\Delta; \Gamma \vdash B\ \mathit{true}\,[C]$ that uses names from some context $\Sigma' \supseteq (\Sigma, X{:}A)$, where X is fresh. But then $\Sigma' \supseteq \Sigma$, and therefore the same derivation proves $\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$ as well. This reasoning gives rise to the following elimination rule for $A \not\twoheadrightarrow B$.

$$\frac{\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}$$

The local reduction for the new type operator is justified by the strengthening principle, because of the assumption that X does not appear in Σ, Δ, Γ, B, C.

$$\dfrac{\dfrac{(\Sigma, X{:}A); \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]}}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]} \quad \Longrightarrow_R \quad \Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$$

The local expansion is justified by the weakening principle

$$\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C] \quad \Longrightarrow_E \quad \dfrac{\dfrac{\dfrac{\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}}{(\Sigma, X{:}A); \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}}{\Sigma; \Delta; \Gamma \vdash A \not\twoheadrightarrow B\ \mathit{true}\,[C]}$$

We conclude this section with a summary of the system with names, as presented thus far. We postpone proving its properties until Section 2.2 where we introduce a proof-term calculus for the judgments. Proof terms will give us a way to describe explicitly the process of reflection, and will provide a concrete notation for developing our meta theory.

Names	X, Y	$\in$	$\mathcal{N}$
Supports	C, D	$::=$	$\cdot \mid C, X$
Propositions	A, B	$::=$	$P \mid A \to B \mid A \not\twoheadrightarrow B \mid \Box_C A \mid \Diamond_C A$
True hypothesis	Γ	$::=$	$\cdot \mid \Gamma, A\ \mathit{true}$
Necessary hypothesis	Δ	$::=$	$\cdot \mid \Delta, A\ \mathit{nec}[C]$
Name context	Σ	$::=$	$\cdot \mid \Sigma, X{:}A$

The system consists of the following six judgments: $\Sigma \vdash C\ \mathit{supp}$, $\Sigma \vdash A\ \mathit{prop}$, $\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$, $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ and $\Sigma; \Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]$, whose rules we list below.

Definition of $\Sigma \vdash C\ \mathit{supp}$.

$$\frac{}{\Sigma \vdash \cdot\ \mathit{supp}} \qquad \frac{\Sigma \vdash C\ \mathit{supp} \qquad X \in \mathrm{dom}(\Sigma)}{\Sigma \vdash C, X\ \mathit{supp}}$$

Definition of $\Sigma \vdash A\ \mathit{prop}$.

$$\frac{}{\Sigma \vdash P\ \mathit{prop}} \qquad \frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash B\ \mathit{prop}}{\Sigma \vdash A \rightarrow B\ \mathit{prop}} \qquad \frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash B\ \mathit{prop}}{\Sigma \vdash A \not\twoheadrightarrow B\ \mathit{prop}}$$

$$\frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Box_C A\ \mathit{prop}} \qquad \frac{\Sigma \vdash A\ \mathit{prop} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Diamond_C A\ \mathit{prop}}$$

Definition of $\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$.

$$\frac{C \subseteq D}{\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[D] \quad \Sigma; \Delta; \Gamma \vdash (C \setminus X)\ \mathit{holds}\,[D] \quad X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D] \qquad \Sigma; \Delta; \cdot \vdash C_1\ \mathit{holds}\,[C]}{\Sigma; \Delta; \Gamma \vdash C_1\ \mathit{holds}\,[D]}$$

Definition of $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$.

$$\frac{X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C, X]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D] \qquad \Sigma; \Delta; \cdot \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[D]}$$

$$\frac{}{\Sigma; \Delta; (\Gamma, A\ \mathit{true}) \vdash A\ \mathit{true}\,[C]}$$

$$\frac{\Sigma; \Delta; (\Gamma, A\ \mathit{true}) \vdash B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A \rightarrow B\ \mathit{true}\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash A \rightarrow B\ \mathit{true}\,[C] \qquad \Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}$$

$$\frac{\Sigma; (\Delta, A\ \mathit{nec}[C]); \Gamma \vdash C\ \mathit{holds}\,[D]}{\Sigma; (\Delta, A\ \mathit{nec}[C]); \Gamma \vdash A\ \mathit{true}\,[D]}$$

$$\frac{\Sigma; \Delta; \cdot \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash \Box_C A\ \mathit{true}\,[D]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash \Box_C A\ \mathit{true}\,[D] \qquad \Sigma; (\Delta, A\ \mathit{nec}[C]); \Gamma \vdash B\ \mathit{true}\,[D]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[D]}$$

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A \nrightarrow B\ \mathit{true}\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash A \nrightarrow B\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]}$$

Definition of $\Sigma; \Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]$.

$$\frac{\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash A\ \mathit{poss}\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash C\ \mathit{holds}\,[D] \qquad \Sigma; \Delta; \cdot \vdash A\ \mathit{true}\,[C]}{\Sigma; \Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]}{\Sigma; \Delta; \Gamma \vdash \Diamond_C A\ \mathit{true}\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash \Diamond_{C_1} A\ \mathit{true}\,[D] \qquad \Sigma; \Delta; A\ \mathit{true} \vdash B\ \mathit{true}\,[C_1]}{\Sigma; \Delta; \Gamma \vdash \langle C_1, B\rangle\ \mathit{poss}\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash \Diamond_{C_1} A\ \mathit{true}\,[D] \qquad \Sigma; \Delta; A\ \mathit{true} \vdash \langle C_2, B\rangle\ \mathit{poss}\,[C_1]}{\Sigma; \Delta; \Gamma \vdash \langle C_2, B\rangle\ \mathit{poss}\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash \Box_C A\ \mathit{true}\,[D] \qquad \Sigma; (\Delta, A\ \mathit{nec}[C]); \Gamma \vdash \langle C_2, B\rangle\ \mathit{poss}\,[D]}{\Sigma; \Delta; \Gamma \vdash \langle C_2, B\rangle\ \mathit{poss}\,[D]}$$

2.2 Modal ν-calculus

2.2.1 Partial judgments and proof terms

In this section, we develop a proof-term system for the modal logic of partial judgments, which we call the *modal ν-calculus*. The presentation will closely follow the development and methodology of the modal λ-calculus from Section 1.2. Each of the judgments $\Delta; \Gamma \vdash C\ \mathit{holds}\,[D]$, $\Delta; \Gamma \vdash A\ \mathit{true}\,[C]$, and $\Delta; \Gamma \vdash \langle C, A\rangle\ \mathit{poss}\,[D]$ defined in the previous sections, is now decorated with proof terms, and has the form $\Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, $\Delta; \Gamma \vdash e : A\,[C]$, and $\Delta; \Gamma \vdash f \div_C A\,[D]$, respectively. As can be noticed, we now have three separate syntactic categories that serve to encode proofs of our judgments.

1. *Expressions* are ranged over by e, and serve as proofs for partial truth and partial necessity.

2. *Phrases* are ranged over by f, and serve to witness simultaneous possibility.

3. *Explicit substitutions* are ranged over by Θ, and serve as proof objects for the support judgment $C\ \mathit{holds}\,[D]$. Correspondingly, they will be used to witness derivation of proofs by reflection.

The assumptions from contexts Δ and Γ are now labeled with variables. We write $x{:}A$ and $u{::}A[C]$ to denote that x stands for a proof of $A\ \mathit{true}$ and that u stands for a proof of $A\ \mathit{nec}[C]$, respectively. Just as in Section 1.2, we will refer to variables x as *ordinary* or *value* variables, and to variables u as *modal* or *expression* variables. The usual assumptions of variable contexts apply here as well: variables declared in Δ and Γ are considered different, and we tacitly employ α-renaming to guarantee this invariant.

We start with the formulation of the λ-calculus fragment of the system. The development is fairly standard. The decorated version of the hypothesis rule of the truth judgment has the form

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

The associated substitution principle is also customary. Because the judgments $\Delta; \Gamma \vdash f \div_D A\,[C]$ and $\Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C]$ are defined in a mutually recursive fashion with the truth judgment, we list here the substitution principles for value variables for all the three judgments.

Principle (Value substitution)
Let $\Delta; \Gamma \vdash e_1 : A\,[C]$. Then the following holds:

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

The rules for implication introduction and elimination are annotated using λ-abstraction and application, respectively, and the local soundness and completeness are witnessed by local reduction and expansion on proof terms.

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

$$\begin{array}{lll}(\lambda x{:}A.\ e_1)\ e_2 & \Longrightarrow_R & [e_2/x]e_1 \\ \\ e : A \to B\,[C] & \Longrightarrow_E & \lambda x{:}A.\ (e\ x) \qquad \text{where } x \text{ not free in } e\end{array}$$

Of course, the most important development in this section concerns names, partiality and the treatment of reflection. In order to define the notion of proof for the judgment of partial truth, we allow names into the syntactic category of expression. Thus, for example, using names to derive partial truth is now formalized by the following rule.

$$\frac{\mathit{typeof}(X) = A}{\Delta; \Gamma \vdash X : A\,[C, X]}$$

The justification for this rule is as follows. If X is associated with the proposition A, then it stands for a proof of A *true*. Thus, we may use X itself as a proof of A *true*, which is partial in X. Notice that we allow weakening with an arbitrary support C, in order to provide for the support weakening principle.

Principle (Support weakening)
Let $C \subseteq D$ be two supports. Then the following holds.

1. if $\Delta; \Gamma \vdash e : A\,[C]$, then $\Delta; \Gamma \vdash e : A\,[D]$

2. if $\Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C]$, then $\Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$
3. if $\Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C_1]$, then $\Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [C_1]$
4. if $\Delta; \Gamma \vdash f \div_{C_1} A\,[C]$, then $\Delta; \Gamma \vdash f \div_{C_1} A\,[D]$

Associated with the notion of partial proofs is the reflection principle as a way to remove or replace the support of a given derivation. In Section 2.1, we used the judgment C *holds* $[D]$ to formalize when a support C may be replaced by the support D in any given derivation of partial truth. A proof-annotated version of this judgment has the form $\Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, where Θ belongs to the syntactic category of *explicit substitution*.

Definition 5 (Explicit substitution, its domain and range)
An explicit substitution is a finite set of pairs $X \rightarrow e$, where X is a name, and e is an expression, so that the name X is paired up with the expression e. Given an explicit substitution Θ, its domain and range are the following sets.

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

and

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

The set $\mathrm{fv}(\Theta)$ *of free variables of Θ is the set of free variables of expressions in* $\mathrm{range}(\Theta)$. *The set* $\mathrm{fn}(\Theta)$ *of free names of Θ is the set of names in the domain and range of Θ. The empty substitution is denoted as $\langle\ \rangle$.*

Having defined explicit substitutions, we may now use them to axiomatize the judgment $\Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Notice how the rules below closely resemble the rules of the judgment $\Delta; \Gamma \vdash C$ *holds* $[D]$ from Section 2.1.2.

$$\frac{C \subseteq D}{\Delta; \Gamma \vdash \langle\ \rangle : [C] \Rightarrow [D]} \qquad \frac{\Delta; \Gamma \vdash e : A\,[D] \quad \Delta; \Gamma \vdash \langle\Theta\rangle : [C \setminus X] \Rightarrow [D] \quad \mathit{typeof}(X) = A}{\Delta; \Gamma \vdash \langle X \rightarrow e, \Theta\rangle : [C] \Rightarrow [D]}$$

Every explicit substitution Θ determines a function $[\![\Theta]\!]$ from names to expressions, defined as follows.

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

This function can also be uniquely extended to a new function $\{\Theta\}$ that acts over arbitrary expressions and phrases. We will define this function explicitly in Section 2.2.3, once we introduce all the expression constructors of the ν-calculus. Here we just present several typical rules.

$$\begin{array}{llcll} \{\Theta\} & X & = & [\![\Theta]\!](X) & \\ \{\Theta\} & x & = & x & \\ \{\Theta\} & \lambda x{:}A.\ e & = & \lambda x{:}A.\ \{\Theta\}e & x \notin \mathrm{fv}(\Theta) \\ \{\Theta\} & e_1\ e_2 & = & \{\Theta\}e_1\ \{\Theta\}e_2 & \end{array}$$

Given two explicit substitutions Θ and Θ', we can define the operation of *substitution composition* $\Theta \circ \Theta'$, so that $\{\Theta \circ \Theta'\}$ is a composition of functions $\{\Theta\}$ and $\{\Theta'\}$. We also postpone the definition of this operation until Section 2.2.3.

The operation $\{\Theta\}$ is the crucial part of the ν-calculus, because it describes how expressions are *reflected*, i.e. transformed from proofs of *categorical* partial judgments into proofs of total judgments. For example, if e is an expression such that $\vdash e : A\,[C]$, and $\langle\Theta\rangle : [C] \Rightarrow [\,]$, then *reflection* of e under Θ is defined as $\{\Theta\}e$, and it will be the case that $\{\Theta\}e : A$. The typing properties of reflected categorical proofs are established by the following *explicit substitution* principle, which is the equivalent of the reflection principles in the logic of partial judgments.

Principle (Explicit substitution)
Let $\Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Then the following holds:

1. if $\Delta; \cdot \vdash e : A\,[C]$, then $\Delta; \Gamma \vdash \{\Theta\}e : A\,[D]$
2. if $\Delta; \cdot \vdash \langle\Theta_1\rangle : [C_1] \Rightarrow [C]$, then $\Delta; \Gamma \vdash \langle\Theta \circ \Theta_1\rangle : [C_1] \Rightarrow [D]$
3. if $\Delta; \cdot \vdash f \div_{C_1} A\,[C]$, then $\Delta; \Gamma \vdash \{\Theta\}f \div_{C_1} A\,[D]$

Because modal variables stand for proof expressions that are subject to reflection, the hypothesis rule for modal variables must specify the explicit substitutions that will guide the reflection. The annotated version of this rule has the following form.

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

As can be noticed, each use of modal variable u is now paired up with an explicit substitution Θ (and when Θ is the empty substitution, we will abbreviate $\langle\Theta\rangle u$ simply as u). The above rule realizes a form of elimination for the bounded universal quantification that is embodied by relativized necessitation. Indeed, if $u{::}A[C]$ stands for a proof that A *true* in *any* world in which C *holds*, and we have an explicit substitution Θ proving that C *holds* $[D]$ in the current world, then A *true* $[D]$ must hold in the current world. The proof of the later, however, is obtained by reflection.

This intuition gives rise to the new operation of modal substitution $[\![e/u]\!]e'$, which substitutes the categorical proof e for u in e'. However, e may first be *reflected*, i.e. modified in accordance with the explicit substitutions that are paired up with the occurrences of u in e'. The new operation is defined by induction on the structure of e'. Again, we postpone the complete definition for Section 2.2.3, where we introduce all of our language constructs. Here we present the two most important cases, which illustrate the gist of the operation of modal substitution.

$$\begin{array}{lcll}
[\![e/u]\!]\langle\Theta\rangle u & = & \{[\![e/u]\!]\Theta\}e & \\
[\![e/u]\!]\langle\Theta\rangle v & = & \langle[\![e/u]\!]\Theta\rangle v & u \neq v
\end{array}$$

It is essential to observe in these equations that substituting e for u in the term $\langle\Theta\rangle u$ actually applies $\{[\![e/u]\!]\Theta\}$ to e. This explicit substitution exactly carries out the process of *reflection* mentioned above – the *categorical* expression e is reflected before it is substituted for u. Reflection of categorical expressions is what differentiates modal substitution from the ordinary value substitution. Ordinary value substitution treats the substituted expressions parametrically, and is not allowed to modify them in any way.

Principle (Modal substitution)
Let $\Delta; \cdot \vdash e : A\,[C]$. Then the following holds:

1. if $(\Delta, u{::}A[C]); \Gamma \vdash e_2 : B\,[D]$, then $\Delta; \Gamma \vdash [\![e_1/u]\!]e_2 : B\,[D]$

2. if $(\Delta, u{::}A[C]); \Gamma \vdash \langle\Theta\rangle : [D'] \Rightarrow [D]$, then $\Delta; \Gamma \vdash \langle[\![e_1/u]\!]\Theta\rangle : [D'] \Rightarrow [D]$

3. if $(\Delta, u{::}A[C]); \Gamma \vdash f \div_{C_1} B\,[D]$, then $\Delta; \Gamma \vdash [\![e_1/u]\!]f \div_{C_1} B\,[D]$

The introduction and elimination rules for relativized modal necessity operator use the **box** and **let box** proof term constructors, just like in the modal λ-calculus.

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

However, in the ν-calculus, the local reduction is realized by means of the new operation of modal substitution $[\![e_1/u]\!]e_2$.

$$\textbf{let box}\ u = e_1\ \textbf{in}\ e_2 \quad \Longrightarrow_R \quad [\![e_1/u]\!]e_2 : B\,[D]$$

The local expansion still has the same form as in Section 1.1.3.

$$e : \Box_C A\,[D] \quad \Longrightarrow_E \quad \textbf{let box}\ u = e\ \textbf{in box}\ u$$

Example 9 Let X be a name of type A. Then the term T defined as

$$\textbf{let box}\ u = (\textbf{box}\ X)\ \textbf{in box}\ (\lambda y{:}A.\ \langle X \to y\rangle u)$$

is well-typed, of type $\Box(A \to A)$. The β-reduction of T is computed as

$$\begin{aligned} & [\![X/u]\!](\textbf{box}\ (\lambda y{:}A.\ \langle X \to y\rangle u)) \\ = \ & \textbf{box}\ (\lambda y{:}A.\ \{X \to y\}X) \\ = \ & \textbf{box}\ (\lambda y{:}A.\ y) \end{aligned}$$

■

Example 10 Let C and D be well-formed supports such that $C \subseteq D$. Then the following are valid typings in the modal ν-calculus.

1. $(\Delta, u{::}A[C]); \Gamma \vdash u : A\,[D]$

2. $\Delta; \Gamma \vdash \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ u : \Box_C A \to \Box_D A$

3. $\Delta; \Gamma \vdash \lambda x.\ \textbf{let box}\ u = x\ \textbf{in}\ u : \Box_C A \to A\,[D]$

4. $\Delta; \Gamma \vdash \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box box}\ u : \Box_C A \to \Box\Box_C A$

5. $\Delta; \Gamma \vdash \lambda x.\ \lambda y.\ \textbf{let box}\ u = x\ \textbf{in let box}\ v = y\ \textbf{in box}\ u\ v$
   $: \Box_C(A \to B) \to \Box_C A \to \Box_C B$

■

The proof annotation of the judgment for simultaneous possibility starts with the following two rules.

$$\frac{\Delta;\Gamma \vdash e : A\,[C]}{\Delta;\Gamma \vdash e \div A\,[C]} \qquad \frac{\Delta;\Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D] \qquad \Delta;\cdot \vdash e : A\,[C]}{\Delta;\Gamma \vdash [\Theta, e] \div_C A\,[D]}$$

The first rule follows the definitional property (1) of simultaneous possibility from Section 2.1.4. If a proposition A is true in the current world, then A is possible (simultaneously with the empty support). If the witness for the truth of A is the expression e, then e witness the possibility of A as well.

The second rule above is justified by the definitional property (2) of simultaneous possibility. The rule prescribes the pair $[\Theta, e]$ as a witness for simultaneous truth of $\Delta;\Gamma \vdash C\ \mathit{holds}\,[D]$ and $\Delta;\Gamma \vdash A\ \mathit{true}\,[D]$. In this pair, Θ is a proof of $\Delta;\Gamma \vdash C\ \mathit{holds}\,[D]$, and e is a proof for $\Delta;\cdot \vdash A\ \mathit{true}\,[C]$. By reflection, these two can obtain a derivation of $\Delta;\Gamma \vdash A\ \mathit{true}\,[D]$. Notice that e has to be typed with an empty context Γ, in order to enable reflection.

The introduction rule for $\Diamond_C$ uses the phrase constructors **dia** to internalize the judgment for simultaneous modal possibility, just like in the modal λ-calculus from Section 1.1.4.

$$\frac{\Delta;\Gamma \vdash f \div_C A\,[D]}{\Delta;\Gamma \vdash \textbf{dia}\ f : \Diamond_C A\,[D]}$$

The elimination rules for simultaneous possibility follow he inference rules from Section 2.1.4. We have two different **let** forms, which serve as proof terms corresponding to two different definitional properties. For definitional property (3), we use **let cdia** $x = e_1$ **in** e_2, where e_2 is an expression; for the definitional property (4), we use **let dia** $x = e_1$ **in** f where f is a phrase. As customary in the judgments for possibility, we also have a term constructor **let box** $u = e$ **in** f, that serves to eliminate relativized necessity in the judgment for simultaneous possibility.

$$\frac{\Delta;\Gamma \vdash e_1 : \Diamond_{C_1} A\,[D] \qquad \Delta; x{:}A \vdash e_2 : B\,[C_1]}{\Delta;\Gamma \vdash \textbf{let cdia}\ x = e_1\ \textbf{in}\ e_2 \div_{C_1} B\,[D]}$$

$$\frac{\Delta;\Gamma \vdash e : \Diamond_{C_1} A\,[D] \qquad \Delta; x{:}A \vdash f \div_{C_2} B\,[C_1]}{\Delta;\Gamma \vdash \textbf{let dia}\ x = e\ \textbf{in}\ f \div_{C_2} B\,[D]}$$

$$\frac{\Delta;\Gamma \vdash e : \Box_C A\,[D] \qquad (\Delta, u{::}A[C]);\Gamma \vdash f \div_{C_1} B\,[D]}{\Delta;\Gamma \vdash \textbf{let box}\ u = e\ \textbf{in}\ f \div_{C_1} B\,[D]}$$

The local reductions and expansions are

$$\begin{array}{rcl} \textbf{let cdia}\ x = \textbf{dia}\ f_1\ \textbf{in}\ e & \Longrightarrow_R & \langle\!\langle f_1/x \rangle\!\rangle e \\ \textbf{let dia}\ x = \textbf{dia}\ f_1\ \textbf{in}\ f & \Longrightarrow_R & \langle\!\langle f_1/x \rangle\!\rangle f \end{array}$$

$$e : \Diamond_C\,[D] \quad \Longrightarrow_E \quad \textbf{dia}\ (\textbf{let cdia}\ x = e\ \textbf{in}\ x)$$

Example 11 Let C, C_1, D be well-formed supports such that $C \subseteq D$. Then the following are valid typings in the modal ν-calculus.

1. $\Delta; \Gamma \vdash \lambda x.\ \mathbf{dia}\ x : A \rightarrow \Diamond A$

2. $\Delta; \Gamma \vdash \lambda x.\ \mathbf{dia}\ (\mathbf{let\ dia}\ y = x\ \mathbf{in\ let\ cdia}\ z = y\ \mathbf{in}\ z) : \Diamond_{C_1} \Diamond_C A \rightarrow \Diamond_C A$

3. $\Delta; \Gamma \vdash \lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ dia}\ (\mathbf{let\ cdia}\ z = y\ \mathbf{in}\ u\ z)$
$: \Box_C(A \rightarrow B) \rightarrow \Diamond_D A \rightarrow \Diamond_D B$

■

The two operations $\langle\!\langle f_1/x \rangle\!\rangle e$ and $\langle\!\langle f_1/x \rangle\!\rangle f$ are defined by induction on the structure of f_1 as follows.

$$\begin{array}{rcl}
\langle\!\langle e_1/x \rangle\!\rangle e & = & [e_1/x]e \\
\langle\!\langle [\Theta, e_1]/x \rangle\!\rangle e & = & [\Theta, ([e_1/x]e)] \\
\langle\!\langle \mathbf{let\ cdia}\ y = e_1\ \mathbf{in}\ e_2/x \rangle\!\rangle e & = & \mathbf{let\ cdia}\ y = e_1\ \mathbf{in}\ [e_2/x]e \\
\langle\!\langle \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle e & = & \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle e \\
\langle\!\langle \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle e & = & \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle e
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle e_1/x \rangle\!\rangle f & = & [e_1/x]f \\
\langle\!\langle [\Theta, e_1]/x \rangle\!\rangle f & = & \{\Theta\}([e_1/x]f) \\
\langle\!\langle \mathbf{let\ cdia}\ y = e_1\ \mathbf{in}\ e_2/x \rangle\!\rangle f & = & \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ [e_2/x]f \\
\langle\!\langle \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle f & = & \mathbf{let\ dia}\ y = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle f \\
\langle\!\langle \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2/x \rangle\!\rangle f & = & \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ \langle\!\langle f_2/x \rangle\!\rangle f
\end{array}$$

We emphasize in the above definition the most characteristic case, which defines the value of $\langle\!\langle \mathbf{let\ cdia}\ y = e_1\ \mathbf{in}\ e_2/x \rangle\!\rangle f$ to be $\mathbf{let\ dia}\ x = e_1\ \mathbf{in}\ [e_2/x]f$. Notice how the elimination form was changed from **let cdia** in the argument of the substitution, to **let dia** in the result.

Application of $\{\Theta\}$ to a phrase f realizes the reflection principle for phrases. It is defined by induction on the structure of f.

$$\begin{array}{rcl}
\{\Theta\}e & = & \{\Theta\}e \\
\{\Theta\}[\Theta_1, e] & = & [\Theta \circ \Theta_1, e] \\
\{\Theta\}(\mathbf{let\ cdia}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let\ cdia}\ x = \{\Theta\}e_1\ \mathbf{in}\ e_2 \\
\{\Theta\}(\mathbf{let\ dia}\ x = e_1\ \mathbf{in}\ f_2) & = & \mathbf{let\ dia}\ x = \{\Theta\}e_1\ \mathbf{in}\ f_2 \\
\{\Theta\}(\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2) & = & \mathbf{let\ box}\ u = \{\Theta\}e_1\ \mathbf{in}\ \{\Theta\}f_2
\end{array}$$

Notice here that we only apply $\{\Theta\}$ in the body of **let box**, but not in the bodies of the other let forms. This fact closely corresponds to the presented typing rules for simultaneous possibility and is therefore important for the soundness of the calculus. Indeed, when compared to the other let forms, the rule for **let box** is the only one using the same support D in both of the premises. Because the explicit substitution Θ may change the support of a phrase it is applied to, we must apply Θ to both the branch and the body of the **let box** in order to preserve the equality of their supports.

Example 12 Let X and Y be names of type A, and let e, f be expressions such that $e : A$, $f : A \to A$. Consider the phrase F defined as

$$\textbf{let dia } y = \textbf{dia } [X \to e, X] \textbf{ in } [X \to f\ (X), Y \to y, Y].$$

The phrase F is well-typed, with $F \div_{X,Y} A$. The β-reduction of F is computed as

$$\begin{array}{rl} & \{X \to e\}([X/y][X \to f\ (X), Y \to y, Y]) \\ = & \{X \to e\}[X \to f\ (X), Y \to X, Y] \\ = & [(X \to e) \circ (X \to f\ (X), Y \to X), Y] \\ = & [X \to f\ (e), Y \to e, Y] \end{array}$$

■

Principle (Phrase substitution)
If $\Delta; \Gamma \vdash f_1 \div_{C_1} A\, [D]$, then the following holds:

1. if $\Delta; x{:}A \vdash e : B\, [C_1]$, then $\Delta; \Gamma \vdash \langle\!\langle f_1/x \rangle\!\rangle e \div_{C_1} B\, [D]$.
2. if $\Delta; x{:}A \vdash f \div_{C_2} B\, [C_1]$, then $\Delta; \Gamma \vdash \langle\!\langle f_1/x \rangle\!\rangle f \div_{C_2} B\, [D]$.

2.2.2 Name-space management

In Section 2.1.6, we decorated the judgments with the additional name context Σ, in order to establish a discipline for dynamic introduction of names into derivation. For example, the partial truth judgment $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\, [C]$ was defined to hold if and only if: (1) the names appearing in Δ, Γ, A and C are all listed with their types in Σ, and (2) there exists a name context $\Sigma' \supseteq \Sigma$, and a proof of $\Delta; \Gamma \vdash A\ \mathit{true}\, [C]$ which uses only the names from Σ'.

As a consequence of this semantics, it follows that a proof for the judgment $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\, [C]$ should in fact consist of a pair $\langle \Sigma', e \rangle$ where $\Sigma' \supseteq \Sigma$, and e is a proof of of the judgment $\Delta; \Gamma \vdash A\ \mathit{true}\, [C]$, under the restriction that e only uses names in Σ'. The proof-annotated version of this judgment has the form

$$\Sigma; \Delta; \Gamma \vdash \langle \Sigma', e \rangle : A\, [C]$$

and it holds if and only if $\Sigma' \supseteq \Sigma$, e is an expression such that $\mathrm{fn}(e) \subseteq \mathrm{dom}(\Sigma')$ and $\Delta; \Gamma \vdash e : A\, [C]$, and Δ, Γ, A and C are well-formed with respect to Σ. In the sense of this definition, it may be said that Σ' declares the names that are *local* to the expression e.

The definition of the annotated judgment obviously motivates the following versions of the structural properties from the previous section.

1. *Strengthening.* If $(\Sigma, X{:}A); \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\, [C]$, then $\Sigma; \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\, [C]$.
2. *Renaming.* If $(\Sigma, X{:}A, \Sigma_1); \Delta; \Gamma \vdash \langle \Sigma', e \rangle : B\, [C]$, and the name Y is fresh, i.e. it does not appear anywhere in the above judgment, then
$$(\Sigma, Y{:}A, [Y/X]\Sigma_1); [Y/X]\Delta; [Y/X]\Gamma \vdash \langle [Y/X]\Sigma', [Y/X]e \rangle : ([Y/X]B)\, [[Y/X]C]$$
3. *Weakening.* If $\Sigma; \Delta; \Gamma \vdash \langle \Sigma', e \rangle : B\, [C]$, and $X \notin \mathrm{dom}(\Sigma')$, then $(\Sigma, X{:}A); \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\, [C]$.

Since the names appearing in the judgment are now declared in the name context, we rephrase the rules to take this into account. In particular, instead of having the rule

$$\frac{\mathit{typeof}(X) = A}{\Delta; \Gamma \vdash X : A\,[C, X]}$$

we can now introduce the following formulation.

$$\frac{X{:}A \in \Sigma}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', X \rangle : A\,[C, X]}$$

The introduction and the elimination rules for the type $A \not\twoheadrightarrow B$ are annotated using the proof terms $\nu X{:}A.\ e$ and **choose** e, as follows.

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\,[C]}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \nu X{:}A.\ e \rangle : A \not\twoheadrightarrow B\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', e \rangle : A \not\twoheadrightarrow B\,[C]}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \mathbf{choose}\ e \rangle : B\,[C]}$$

In the introduction rule it is assumed that X is a *fresh* name, that is, $X \notin \mathrm{dom}(\Sigma')$.[6] The exact identity of the name X in this rule is not important. As ensured by the renaming principle, any unused name X such that as $\mathit{typeof}(X) = A$ may be chosen. This observation justifies the proof term $\nu X{:}A.\ e$ which actually *binds* the name X and allows α-renaming X into other unused names. [7]

The local soundness of the new rules is established by the following local reduction, which we represent in a form of a derivation tree.

$$\frac{\dfrac{(\Sigma, X{:}A); \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\,[C]}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \nu X{:}A.\ e \rangle : A \not\twoheadrightarrow B\,[C]}}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \mathbf{choose}\ \nu X{:}A.\ e \rangle : B\,[C]} \quad \Longrightarrow_R \quad \Sigma; \Delta; \Gamma \vdash \langle (\Sigma', X{:}A), e \rangle : B\,[C]$$

or in a more compact form, using the proof terms:

$$\langle \Sigma', \mathbf{choose}\ \nu X{:}A.\ e \rangle \quad \Longrightarrow_R \quad \langle (\Sigma', X{:}A), e \rangle \qquad X \notin \mathrm{dom}(\Sigma')$$

The local reduction is justified by the strengthening principle. Indeed, if $\langle (\Sigma', X{:}A), e \rangle$ is a witness for $(\Sigma, X{:}A); \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$, then $\Delta; \Gamma \vdash e : B\,[C]$, and X does not appear in Δ, Γ, B or C. By definition, this is sufficient to ensure that $\langle (\Sigma', X{:}A), e \rangle$ is a witness for $\Sigma; \Delta; \Gamma \vdash B\ \mathit{true}\,[C]$ as well.

The local completeness of the rules is established by local elimination as follows.

$$\Sigma; \Delta; \Gamma \vdash \langle \Sigma', e \rangle : A \not\twoheadrightarrow B\,[C] \quad \Longrightarrow_E \quad \frac{\dfrac{\dfrac{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', e \rangle : A \not\twoheadrightarrow B\,[C]}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \mathbf{choose}\ e \rangle : B\,[C]}}{(\Sigma, Y{:}A); \Delta; \Gamma \vdash \langle (\Sigma', Y{:}A), \mathbf{choose}\ e \rangle : B\,[C]}}{\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \nu Y{:}A.\ \mathbf{choose}\ e \rangle : A \not\twoheadrightarrow B\,[C]}$$

[6] And in fact X will not be introduced again by any other ν-construct.

[7] Do I need a comment on application? If $\langle \Sigma_1, e_1 \rangle : A \to B\,[C]$, and $\langle \Sigma_2, e_2 \rangle : A\,[C]$, we assume that Σ_1 and Σ_2 are disjoint, before we join them into $\langle (\Sigma_1, \Sigma_2), (e_1\ e_2) \rangle : B\,[C]$? This is justified by the assumption that names are ascribed types globally, and the context Σ only repeats this global typing. In other words, two different name contexts cannot ascribe two different types to one and the same name.

Notice that with this approach names are not completely fresh, as one and the same name may be chosen by both e_1 and e_2 above. Fully fresh names require explicit serialization of the name generation process.

or in a short form:

$$\langle \Sigma', e \rangle \quad \Longrightarrow_E \quad \langle \Sigma', \nu X{:}A.\ \mathbf{choose}\ e \rangle$$

The expanded derivation is justified by the weakening principle, which allows us to conclude $(\Sigma, Y{:}A); \Delta; \Gamma \vdash \langle (\Sigma', Y{:}A), \mathbf{choose}\ e \rangle : B\,[C]$ out of $\Sigma; \Delta; \Gamma \vdash \langle \Sigma', \mathbf{choose}\ e \rangle : B\,[C]$, under the assumption that Y is fresh, i.e. $Y \notin \mathrm{dom}(\Sigma')$.

Observe that the names appearing in the expression e such that $\Delta; \Gamma \vdash e : A\,[C]$ can always be recovered by simply inspecting e. Strictly speaking, therefore, it is not really necessary that the rules of our judgments explicitly carry the second name context Σ'. We can always keep Σ' implicit, and only rely on Σ to declare which names can be used in a well-formed judgment. Thus, we introduce an abbreviated notation for the case when Σ' and Σ are equal. In this case, instead of

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

we will simply write

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

The introduction and elimination rules for $A \twoheadrightarrow B$ now have the following form.

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

It is important, however, to remember that these two rules are just an abbreviation for the old judgment. The name context Σ', while made implicit, remains explicit in the local reduction, and will therefore have a computational import. Once we ascribe operational semantics to the ν-calculus, Σ' will serve as a run-time context that lists the currently generated names. It will be used to determine which names are fresh and can therefore be introduced next time a fresh name is needed.

On a related note, the local reduction associated with the type constructor $\twoheadrightarrow$ will itself have a computational meaning – that of introducing a fresh name into the computation. In the usual formulation of calculi for fresh name generation [PS93, PG00, Ode94] , this operation is not related to a β-reduction, but is formulated by a separate language construct. In this respect, our formulation is closer to the λ-calculus, where computational content is always reserved for β-reduction.

Just as it is customary in λ-calculus to introduce an abbreviation **let val** $x = e_1$ **in** e_2 for the expression $(\lambda x.\ e_2)\ (e_1)$, we can introduce a new abbreviation here as well. We will define a new expression constructor **let name** $X{:}A$ **in** e to stand for

$$(\mathbf{let\ name}\ X{:}A\ \mathbf{in}\ e) \quad = \quad \mathbf{choose}\ (\nu X{:}A.\ e)$$

The typing rule for **let name** is appropriately derived as

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

A similar constructor is introduced in the syntactic category of phrases, with the following typing rule.

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash f \div_C B\,[D]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ name}\ X{:}A\ \mathbf{in}\ f \div_C B\,[D]}$$

In both of these rules, it is assumed that X is a fresh name, i.e. that $X \notin \mathrm{dom}(\Sigma)$.

2.2.3 Summary and structural properties

Syntax

The syntax of the modal ν-calculus is summarized in the table below. Here we assume that $\mathcal{N}$ is a countable universe of names, so that each name is associated with a certain well-formed type, and there is a countably infinite set of names for each type.[8]

$$\begin{array}{lrcl}
\textit{Names} & X, Y & \in & \mathcal{N} \\
\textit{Supports} & C, D & ::= & \cdot \mid C, X \\
\textit{Types} & A, B & ::= & P \mid A \rightarrow B \mid A \not\twoheadrightarrow B \mid \Box_C A \mid \Diamond_C A \\
\textit{Explicit substitutions} & \Theta & ::= & \cdot \mid X \rightarrow e, \Theta \\
\textit{Expressions} & e & ::= & X \mid x \mid \langle\Theta\rangle u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \\
 & & & \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 \\
 & & & \mid \nu X{:}A.\ e \mid \mathbf{choose}\ e \\
 & & & \mid \mathbf{dia}\ f \\
\textit{Phrases} & f & ::= & e \mid [\Theta, e] \mid \mathbf{let\ cdia}\ x = e_1\ \mathbf{in}\ e_2 \\
 & & & \mid \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f \mid \mathbf{let\ box}\ u = e\ \mathbf{in}\ f \\
\textit{Ordinary contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\textit{Modal contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}A[C] \\
\textit{Name context} & \Sigma & ::= & \cdot \mid \Sigma, X{:}A
\end{array}$$

The name contexts Σ are dependent contexts, because types may depend on names. It is assumed that a name declared in Σ may be used in the types appearing to the right of its declaration, but not to the left[9].

Type system

Just like in Section 2.1.6, here too we have six typing judgments: $\Sigma \vdash C\ \mathit{supp}$, $\Sigma \vdash A\ \mathit{type}$, $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$, $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, and $\Sigma; \Delta; \Gamma \vdash f \div_C A\,[D]$, whose rules we list below.

Definition of $\Sigma \vdash C\ \mathit{supp}$.

$$\frac{}{\Sigma \vdash \cdot\ \mathit{supp}} \qquad \frac{\Sigma \vdash C\ \mathit{supp} \qquad X \in \mathrm{dom}(\Sigma)}{\Sigma \vdash C, X\ \mathit{supp}}$$

Definition of $\Sigma \vdash A\ \mathit{type}$.

$$\frac{}{\Sigma \vdash P\ \mathit{type}} \qquad \frac{\Sigma \vdash A\ \mathit{type} \qquad \Sigma \vdash B\ \mathit{type}}{\Sigma \vdash A \rightarrow B\ \mathit{type}} \qquad \frac{\Sigma \vdash A\ \mathit{type} \qquad \Sigma \vdash B\ \mathit{type}}{\Sigma \vdash A \not\twoheadrightarrow B\ \mathit{type}}$$

$$\frac{\Sigma \vdash A\ \mathit{type} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Box_C A\ \mathit{type}} \qquad \frac{\Sigma \vdash A\ \mathit{type} \qquad \Sigma \vdash C\ \mathit{supp}}{\Sigma \vdash \Diamond_C A\ \mathit{type}}$$

Definition of $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$.

[8] Furthermore, we assume that the relation between names and types is well-founded.

[9] This in itself ensures that the relation between names and types is well-founded. Is there a better way to phrase this?

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

Definition of $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$.

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

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

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

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

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash e : B\,[C]}{\Sigma; \Delta; \Gamma \vdash \nu X{:}A.\ e : A \not\twoheadrightarrow B\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash e : A \not\twoheadrightarrow B\,[C]}{\Sigma; \Delta; \Gamma \vdash \mathbf{choose}\ e : B\,[C]}$$

Definition of $\Sigma; \Delta; \Gamma \vdash f \div_C A\,[D]$.

$$\frac{\Sigma; \Delta; \Gamma \vdash e : A\,[C]}{\Sigma; \Delta; \Gamma \vdash e \div\ A\,[C]} \qquad \frac{\Sigma; \Delta; \Gamma \vdash \langle \Theta \rangle : [C] \Rightarrow [D] \qquad \Sigma; \Delta; \cdot \vdash e : A\,[C]}{\Sigma; \Delta; \Gamma \vdash [\Theta, e] \div_C A\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash f \div_C A\,[D]}{\Sigma; \Delta; \Gamma \vdash \mathbf{dia}\ f : \Diamond_C A\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash e_1 : \Diamond_{C_1} A\,[D] \qquad \Sigma; \Delta; x{:}A \vdash e_2 : B\,[C_1]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ cdia}\ x = e_1\ \mathbf{in}\ e_2 \div_{C_1} B\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash e : \Diamond_{C_1} A\,[D] \qquad \Sigma; \Delta; x{:}A \vdash f \div_{C_2} B\,[C_1]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ dia}\ x = e\ \mathbf{in}\ f \div_{C_2} B\,[D]}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash e : \Box_{C_1} A\,[D] \qquad \Sigma; (\Delta, u{::}A[C_1]); \Gamma \vdash f \div_{C_2} B\,[D]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ box}\ u = e\ \mathbf{in}\ f \div_{C_2} B\,[D]}$$

Structural properties

An explicit substitution is a finite set of pairs $X \to e$, where X is a name, and e is an expression. Each explicit substitution Θ defines a function $[\![\Theta]\!]$ from names to expressions.

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

A substitution can be uniquely extended to a function over arbitrary expressions and phrases, in the following way.

Definition 6 (Substitution application)
Given a substitution Θ, the operations $\{\Theta\}e$ and $\{\Theta\}f$ for applying *Θ over the expression e or a phrase f, are defined by induction on the structure of e and f as given below. Substitution application is capture-avoiding.*

$$\begin{array}{lllll}
\{\Theta\} & X & = & [\![\Theta]\!](X) & \\
\{\Theta\} & x & = & x & \\
\{\Theta\} & (\langle\Theta_1\rangle u) & = & \langle\Theta \circ \Theta_1\rangle u & \\
\{\Theta\} & (\lambda x{:}A.\ e_1) & = & \lambda x{:}A.\ \{\Theta\}e_1 & x \notin \mathrm{fv}(\Theta) \\
\{\Theta\} & (e_1\ e_2) & = & \{\Theta\}e_1\ \{\Theta\}e_2 & \\
\{\Theta\} & (\mathbf{box}\ e_1) & = & \mathbf{box}\ e_1 & \\
\{\Theta\} & (\mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let\ box}\ u = \{\Theta\}e_1\ \mathbf{in}\ \{\Theta\}e_2 & u \notin \mathrm{fv}(\Theta) \\
\{\Theta\} & (\nu X{:}A.\ e_1) & = & \nu X{:}A.\ \{\Theta\}e_1 & X \notin \mathrm{fn}(\Theta) \\
\{\Theta\} & (\mathbf{choose}\ e_1) & = & \mathbf{choose}\ \{\Theta\}e_1 & \\
\{\Theta\} & (\mathbf{dia}\ f_1) & = & \mathbf{dia}\ \{\Theta\}f_1 & \\
 & & & & \\
\{\Theta\} & e_1 & = & \{\Theta\}e_1 & \\
\{\Theta\} & [\Theta_1, e_1] & = & [\Theta \circ \Theta_1, e_1] & \\
\{\Theta\} & \mathbf{let\ cdia}\ x = e_1\ \mathbf{in}\ e_2 & = & \mathbf{let\ cdia}\ x = \{\Theta\}e_1\ \mathbf{in}\ e_2 & x \notin \mathrm{fv}(\Theta) \\
\{\Theta\} & \mathbf{let\ dia}\ x = e_1\ \mathbf{in}\ f_2 & = & \mathbf{let\ dia}\ x = \{\Theta\}e_1\ \mathbf{in}\ f_2 & x \notin \mathrm{fv}(\Theta) \\
\{\Theta\} & \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ f_2 & = & \mathbf{let\ box}\ u = \{\Theta\}e_1\ \mathbf{in}\ \{\Theta\}f_2 & u \notin \mathrm{fv}(\Theta)
\end{array}$$

The most important aspect of the above definition is that substitution application does not recursively descend under **box**. This property is of utmost importance for the soundness of our calculus as it preserves the distinction between the categorical and hypothetical proofs. It is also justified, as applying explicit substitution Θ to the expression e is intended to replace the names which are in the support of e, and names appearing under **box** do not contribute to the support.

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

Definition 7 (Composition of substitutions)
Given two substitutions Θ_1 and Θ_2, their composition $\Theta_1 \circ \Theta_2$ is the set

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

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

$$\begin{array}{lll}
\Psi_1 & = & \{X \to [\![\Theta_1]\!]\,(X) \mid X \in \mathrm{dom}(\Theta_1) \setminus \mathrm{dom}(\Theta_2)\} \\
\Psi_2 & = & \{X \to \{\Theta_1\}([\![\Theta_2]\!]\,(X)) \mid X \in \mathrm{dom}(\Theta_2)\}
\end{array}$$

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

Lemma 8
Let $\Theta_1, \Theta_2, \Theta_3$ be explicit substitutions. Then the following holds:

1. $\{\Theta_1\}(\{\Theta_2\}e) = \{\Theta_1 \circ \Theta_2\}e$, *for every expression* e
2. $\{\Theta_1\}(\{\Theta_2\}f) = \{\Theta_1 \circ \Theta_2\}f$, *for every phrase* f
3. $\Theta_1 \circ (\Theta_2 \circ \Theta_3) = (\Theta_1 \circ \Theta_2) \circ \Theta_3$, *for every explicit substitution* Θ_3.

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

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

case $f = [\Theta', e]$. Then $\{\Theta_1\}(\{\Theta_2\}f) = \{\Theta_1\}[\Theta_2 \circ \Theta', e] = [\Theta_1 \circ (\Theta_2 \circ \Theta'), e] = \{\Theta_1 \circ \Theta_2\}f$.

case $\Theta_3 = (X \mapsto e, \Theta')$. Let Z be an arbitrary name.

If $Z = X$, then $\{\Theta_1\}([\![\Theta_2 \circ \Theta_3]\!](Z)) = \{\Theta_1\}(\{\Theta_2\}e)$. By first induction hypothesis, this is equal to $\{\Theta_1 \circ \Theta_2\}e = \{\Theta_1 \circ \Theta_2\}([\![\Theta_3]\!](Z))$.

If $Z \neq X$, then $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta_3]\!](Z) = \{\Theta_1\}\,[\![\Theta_2 \circ \Theta']\!](Z)$, and also $\{\Theta_1 \circ \Theta_2\}\,[\![\Theta_3]\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta']\!](Z)$. By second induction hypothesis, $\Theta_1 \circ (\Theta_2 \circ \Theta') = (\Theta_1 \circ \Theta_2) \circ \Theta'$, and therefore $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta']\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta']\!](Z)$. Therefore, $\{\Theta_1\}\,[\![\Theta_2 \circ \Theta_3]\!](Z) = \{\Theta_1 \circ \Theta_2\}\,[\![\Theta_3]\!](Z)$, thus concluding the proof.

■

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

Theorem 9 (Structural properties)
The following are the structural properties of the judgment $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$. Similar properties hold for $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$ and $\Sigma; \Delta; \Gamma \vdash f \div_C A\,[D]$, but we omit these for simplicity.

1. Context weakening *Let $\Sigma \subseteq \Sigma'$, $\Delta \subseteq \Delta'$ and $\Gamma \subseteq \Gamma'$. If $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, then $\Sigma'; \Delta'; \Gamma' \vdash e : A\,[C]$.*
2. Contraction on variables
   (a) *if $\Sigma; \Delta; (\Gamma, x{:}A, y{:}A) \vdash e : A\,[C]$, then $\Sigma; \Delta; (\Gamma, w{:}A) \vdash [w/x, w/y]e : A\,[C]$*
   (b) *if $\Sigma; (\Delta, u{::}A[C_1], v{::}A[C_1]); \Gamma \vdash e : A\,[C]$, then $\Sigma; (\Delta, w{::}A[C_1]); \Gamma \vdash [w/u, w/v]e : A\,[C]$.*
3. Renaming *If $(\Sigma, X{:}A, \Sigma_1); \Delta; \Gamma \vdash e : B\,[C]$, and the name $Y{:}A$ is fresh, then*

$$(\Sigma, Y{:}A, [Y/X]\Sigma_1); [Y/X]\Delta; [Y/X]\Gamma \vdash [Y/X]e : ([Y/X]B)\,[[Y/X]C]$$

Proof: By straightforward induction on the structure of the derivations. ■

Theorem 10 (Support weakening)
Support weakening is covariant on the right-hand side and contravariant on the left-hand side of the judgments. More formally, let $C \subseteq D \subseteq \mathrm{dom}(\Sigma)$ be well-formed supports. Then the following holds:

1. *if $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash e : A\,[D]$*
2. *if $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C]$, then $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$*
3. *if $\Sigma; \Delta; \Gamma \vdash f \div_{C_1} A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash f \div_{C_1} A\,[D]$*
4. *if $\Sigma; (\Delta, u{::}A[D]); \Gamma \vdash e : B\,[C_1]$, then $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash e : B\,[C_1]$*
5. *if $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C_1]$, then $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [C_1]$*
6. *if $\Sigma; (\Delta, u{::}A[D]); \Gamma \vdash f \div_{C_1} B\,[C_2]$, then $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash f \div_{C_1} B\,[C_2]$*

Proof: The first three statements are proved by simultaneous induction on the structure of their derivations. The last three statements are also proved by simultaneous induction on the structure of their respective derivations, but are independent of the first three. ■

Theorem 11 (Explicit substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Then the following holds:

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

Proof: By simultaneous induction on the structure of the derivations. Proving the first and the third induction hypothesis is easy. For the second induction hypothesis, let $\Psi = \Theta \circ \Theta'$. We split Ψ into two disjoint sets:

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

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

(a) if $X \notin \mathrm{dom}(\Psi)$ and $X \in C_1$, then $X \in D$

(b) if $X \to e \in \Psi$, then $\Sigma; \Delta; \Gamma \vdash e : A\,[D]$

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

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

This concludes the proof, since $e = \{\Theta\}e'$. ■

The following theorem is a version of the substitution principle for truth, decorated with explicit proof terms in the judgments.

Theorem 12 (Value substitution principle)
Let $\Sigma; \Delta; \Gamma \vdash e_1 : A\,[C]$. Then the following holds:

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

Proof: Simultaneous induction on the two derivations. ■

Definition 13 (Modal substitution)
The capture-avoiding substitution of $\llbracket e/u \rrbracket$ of the expression e for the modal variable *u is defined recursively as presented below. Note that in the first clause of the definition, substituting e for u in $\langle\Theta\rangle u$ is defined to actually carry of the explicit substitution.*

$$
\begin{array}{llcll}
\llbracket e/u \rrbracket & \langle\Theta\rangle u & = & \{\llbracket e/u \rrbracket\Theta\}e & \\
\llbracket e/u \rrbracket & \langle\Theta\rangle v & = & \langle\llbracket e/u \rrbracket\Theta\rangle v & u \neq v \\
\llbracket e/u \rrbracket & x & = & x & \\
\llbracket e/u \rrbracket & X & = & X & \\
\llbracket e/u \rrbracket & \lambda x{:}A.\ e_1 & = & \lambda x{:}A.\ \llbracket e/u \rrbracket e_1 & x \notin \mathrm{fv}(e) \\
\llbracket e/u \rrbracket & e_1\ e_2 & = & \llbracket e/u \rrbracket e_1\ \llbracket e/u \rrbracket e_2 & \\
\llbracket e/u \rrbracket & \mathbf{box}\ e_1 & = & \mathbf{box}\ \llbracket e/u \rrbracket e_1 & \\
\llbracket e/u \rrbracket & \mathbf{let\ box}\ v = e_1\ \mathbf{in}\ e_2 & = & \mathbf{let\ box}\ v = \llbracket e/u \rrbracket e_1\ \mathbf{in}\ \llbracket e/u \rrbracket e_2 & v \notin \mathrm{fv}(e) \\
\llbracket e/u \rrbracket & \nu X{:}A.\ e_1 & = & \nu X{:}A.\ \llbracket e/u \rrbracket e_1 & X \notin \mathrm{fn}(e) \\
\llbracket e/u \rrbracket & \mathbf{choose}\ e_1 & = & \mathbf{choose}\ (\llbracket e/u \rrbracket e_1) & \\
\llbracket e/u \rrbracket & \mathbf{dia}\ f & = & \mathbf{dia}\ (\llbracket e/u \rrbracket f) & \\
& & & & \\
\llbracket e/u \rrbracket & (\cdot) & = & (\cdot) & \\
\llbracket e/u \rrbracket & (X \to e_1, \Theta) & = & (X \to \llbracket e/u \rrbracket e_1, \llbracket e/u \rrbracket\Theta) & \\
& & & & \\
\llbracket e/u \rrbracket & e_1 & = & \llbracket e/u \rrbracket e_1 & \\
\llbracket e/u \rrbracket & [\Theta, e_1] & = & [\llbracket e/u \rrbracket\Theta, \llbracket e/u \rrbracket e_1] & \\
\llbracket e/u \rrbracket & \mathbf{let\ cdia}\ x = e_1\ \mathbf{in}\ e_2 & = & \mathbf{let\ cdia}\ x = \llbracket e/u \rrbracket e_1\ \mathbf{in}\ \llbracket e/u \rrbracket e_2 & x \notin \mathrm{fv}(e) \\
\llbracket e/u \rrbracket & \mathbf{let\ dia}\ x = e_1\ \mathbf{in}\ f_2 & = & \mathbf{let\ dia}\ x = \llbracket e/u \rrbracket e_1\ \mathbf{in}\ \llbracket e/u \rrbracket f_2 & x \notin \mathrm{fv}(e) \\
\llbracket e/u \rrbracket & \mathbf{let\ box}\ v = e_1\ \mathbf{in}\ f_2 & = & \mathbf{let\ box}\ v = \llbracket e/u \rrbracket e_1\ \mathbf{in}\ \llbracket e/u \rrbracket f_2 & v \notin \mathrm{fv}(e)
\end{array}
$$

The following theorem is a version of the substitution principle for relativized necessity.

Theorem 14 (Modal substitution principle)
Let $\Sigma; \Delta; \cdot \vdash e_1 : A\,[C]$. Then the following holds:

1. *if $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash e_2 : B\,[D]$, then $\Sigma; \Delta; \Gamma \vdash \llbracket e_1/u \rrbracket e_2 : B\,[D]$*
2. *if $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$, then $\Sigma; \Delta; \Gamma \vdash \langle\llbracket e_1/u \rrbracket\Theta\rangle : [C_1] \Rightarrow [D]$*

3. *if* $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash f \div_{C_1} B\,[D]$, *then* $\Sigma; \Delta; \Gamma \vdash [\![e_1/u]\!] f \div_{C_1} B\,[D]$

Proof: By simultaneous induction on the two derivations. ■

Definition 15 (Phrase substitution)
The operations $\langle\!\langle f_1/x\rangle\!\rangle e$ *and* $\langle\!\langle f_1/x\rangle\!\rangle f$ *of substituting the phrase* f_1 *into an expression* e *or another phrase* f_2 *are defined by induction on the structure of* f *as follows.*

$$\begin{array}{rcl}
\langle\!\langle e_1/x\rangle\!\rangle e & = & [e_1/x]e \\
\langle\!\langle [\Theta, e_1]/x\rangle\!\rangle e & = & [\Theta, ([e_1/x]e)] \\
\langle\!\langle \textbf{let cdia } y = e_1 \textbf{ in } e_2/x\rangle\!\rangle e & = & \textbf{let cdia } y = e_1 \textbf{ in } [e_2/x]e \\
\langle\!\langle \textbf{let dia } y = e_1 \textbf{ in } f_2/x\rangle\!\rangle e & = & \textbf{let dia } y = e_1 \textbf{ in } \langle\!\langle f_2/x\rangle\!\rangle e \\
\langle\!\langle \textbf{let box } u = e_1 \textbf{ in } f_2/x\rangle\!\rangle e & = & \textbf{let box } u = e_1 \textbf{ in } \langle\!\langle f_2/x\rangle\!\rangle e
\end{array}$$

$$\begin{array}{rcl}
\langle\!\langle e_1/x\rangle\!\rangle f & = & [e_1/x]f \\
\langle\!\langle [\Theta, e_1]/x\rangle\!\rangle f & = & \{\Theta\}([e_1/x]f) \\
\langle\!\langle \textbf{let cdia } y = e_1 \textbf{ in } e_2/x\rangle\!\rangle f & = & \textbf{let dia } y = e_1 \textbf{ in } [e_2/x]f \\
\langle\!\langle \textbf{let dia } y = e_1 \textbf{ in } f_2/x\rangle\!\rangle f & = & \textbf{let dia } y = e_1 \textbf{ in } \langle\!\langle f_2/x\rangle\!\rangle f \\
\langle\!\langle \textbf{let box } u = e_1 \textbf{ in } f_2/x\rangle\!\rangle f & = & \textbf{let box } u = e_1 \textbf{ in } \langle\!\langle f_2/x\rangle\!\rangle f
\end{array}$$

The following theorem establishes that our calculus indeed satisfies the substitution principle for possibility from Section 2.2.1.

Theorem 16 (Phrase substitution principle)
If $\Sigma; \Delta; \Gamma \vdash f_1 \div_{C_1} A\,[D]$, *then the following holds:*

1. *if* $\Sigma; \Delta; x{:}A \vdash e : B\,[C_1]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\!\langle f_1/x\rangle\!\rangle e \div_{C_1} B\,[D]$.

2. *if* $\Sigma; \Delta; x{:}A \vdash f \div_{C_2} B\,[C_1]$, *then* $\Sigma; \Delta; \Gamma \vdash \langle\!\langle f_1/x\rangle\!\rangle f \div_{C_2} B\,[D]$.

Proof: By straightforward induction on the structure of f_1. We just present a selected case when $f_1 = \textbf{let cdia } y = e_1 \textbf{ in } e_2$. In this case, by assumption $\Sigma; \Delta; \Gamma \vdash e_1 : \Diamond_{C_1} A_1\,[D]$, and $\Sigma; \Delta; y{:}A_1 \vdash e_2 : A\,[C_1]$.

To establish the first statement, recall that $\Sigma; \Delta; x{:}A \vdash e : B\,[C_1]$. Then by the value substitution principle, $\Sigma; \Delta; y{:}A_1 \vdash [e_2/x]e : B\,[C_1]$. According to the typing rule for **let cdia**, $\Sigma; \Delta; \Gamma \vdash \textbf{let cdia } y = e_1 \textbf{ in } [e_2/x]e \div_{C_1} B\,[D]$, which was required to prove.

The proof of the second statement is similar. By assumption, $\Sigma; \Delta; x{:}A \vdash f \div_{C_2} B\,[C_1]$, and by the value substitution principle, $\Sigma; \Delta; y{:}A_1 \vdash [e_2/x]f \div_{C_2} B\,[C_1]$. The conclusion now follows by the typing rule for **let dia**. ■

2.3 Notes

Related and future work on names

The work that explicitly motivated the developments presented in this dissertation is described in the series of papers on Nominal Logic and FreshML [GP02, PG00, Pit01, Gab00]. The names of Nominal Logic are introduced as the urelements of Fraenkel-Mostowsky set theory. FreshML is a language for manipulation of object syntax with binding structure based on this model. Its primitive notion is that of swapping of two names which is then used to define the operations of name abstraction (producing an α-equivalence class with respect to the abstracted name) and name concretion (providing a specific representative of an α-equivalence class).

In FreshML, if X is a name appearing in the expression e, then the support of e will contain X, unless X occurrs in dead code or is otherwise abstracted using the construct for name abstraction. Names are introduced into the computation by **new** X **in** e which is roughly equivalent to our **let name** X **in** e. The typing rule for **new** X **in** e requires that X does not appear in the support of e. This way, the type system prevents unabstracted names from escaping the scope of their introducing **new**.

Keeping track of supports in the type system significantly simplifies FreshML when compared to some previous calculi that use names. For example, the ν-calculus of [PS93] describes a system similar to FreshML, but does not track supports of expressions, and does not insist that X is absent from the support of e in **new** $X{:}e$ **in** . This gives rise to a very powerful language, but also a very complicated one. For example, the ν-calculus has a rather involved equational theory; in particular, it does not equate a term with its β-reduct.

The $\lambda\nu$-calculus of [Ode94] introduces a somewhat different idea for treating names, characterized by reductions that push the name declaration inside other term constructors. A typical reduction rule in $\lambda\nu$ would be paraphrased in the notation of $\nu^{\Box}$ as

$$\mathbf{let\ name}\ X\ \mathbf{in}\ (\lambda x.\ e) \longmapsto \lambda x.\ \mathbf{let\ name}\ X\ \mathbf{in}\ e$$

Just like the ν-calculus, $\lambda\nu$ does not keep track of which names appear in the terms. As a consequence, it does not possess the usual progress and preservation properties, as well-typed expressions in $\lambda\nu$ may get stuck. A typical example is the expression $\nu X.\ X$, which does not denote any value.

All these cited name calculi are designed around the single goal: that of providing the operation of equality on names. In contrast to this goal, the modal ν-calculus uses names primarily as a way of describing supports, i.e. as a way of specifying the partiality of expressions. In fact, names in the ν-calculus are second-class objects – they cannot be passed as arguments to other functions, and may not be tested for equality *directly.*

The reason for second-class names has to do with the fact that names in the ν-calculus may be ascribed an arbitrary type; a dynamic introduction of a name of type A into a computation serves as a dynamic extension of the type A. Such an extension may render partial the previously defined functions with domain A. We discuss this issue in more detail in Section 3.2.3, where we define an operational semantics for the modal ν-calculus.

This is not to say that names cannot be tested for equality *indirectly.* As will be explained in Section 3, expressions of the type $\Box_C A$ may be interpreted as syntactic expression with free variables listed in the set C. In Section 3.4, we exploit this feature, and make some initial steps towards extending the ν-calculus with pattern-matching against syntactic expressions. Since the syntactic expressions may contain names, this will provide an indirect way to test for name equality.

Of course, other ways to extend the ν-calculus with first-class names and name equality may be possible. For example, it may be interesting to define a new type constructor

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

so that $\mathbf{N}(A)$ classifies all the names of type A. The question then becomes how names interact with the modal operators. Of course, it is likely that all the difficulties from the name calculi with first-class names (like the ν-calculus of [PS93]) will still be present. We leave this research direction as an important future work.

Even when dealing with second-class names, it seems possible that other approaches may be employed for dynamic name management. For example, the variable declaration $u{:}A\,[C]$ may be viewed as *binding* the names listed in C, so that these names have scope local to the explicit substitutions associated to u. This idea has been employed in [NPP03] to define a dependently typed calculus for representing meta variables in logical frameworks.

Similar idea has been used by Ancona and Moggi in their recent work motivated by the ν-calculus [AM03]. In their paper, name generation is tied to the **box** term constructor, which then also becomes a binder[10].

In this dissertation, we deliberately separate name generation from other language constructs, and give names global scope in order to avoid excessive renaming and rebinding. Moreover, in Chapter 4, we will consider effectful computations where names correspond to particular memory locations and exceptions. In practice today memory and exceptions are typically given global scope, so our approach will faitfully capture this aspect of effects.

Related and future work on contexts and partiality in modal logic

Since modal logic is invented in order to reason about truth relative to various worlds, it shouldn't be a surprise that modal logic and partiality are so closely related. This is especially true of the first-order modal logics *with equality* (and also of higher-order modal logics), where the research questions of interest are typically concerned with reasoning with and about individuals that – in an appropriate sense – do not really exist. Derivations produced in this way are *partial* in the existence of the individuals in question. The names from the modal ν-calculus serve to specify the partiality condition, and thus may be seen as a simplification (appropriate for the propositional partial CS4 that we investigate) of the more general concept of an individual. In this sense, names resemble the non-rigid designators considered by Fitting and Mendelsohn in [FM99], names of Kripke [Kri80], and the virtual individuals of Scott [Sco70], but also touch on the issues of existence and identity explored in [Sco79].

Frequently, the modal reasoning is only valid under a certain set of hypotheses, i.e. a *context*. A context need not include only the existence of individuals, but may contain more general propositions. The study of contexts as first-class logical object has been initiated by McCarthy [McC93], and we also list the work of Attardi and Simi [AS95] as a continuation of this line of research. Most of the work on formalizing contexts has been carried out in a classical setting, but there are also efforts related to the intuitionistic logic, like the recent work of de Paiva [dP03].

It may be particularly convenient to address the mentioned distinction between the partiality in individuals and the partiality in propositions within the framework of a modal type theory. As an illustration – and a rather far-fetched one, currently – consider the following example.

Let $X : \mathit{real}$ be an indeterminate number, for which we have derived that $X^2 = -1$. Such a real number clearly does not exist, and we may easily derive falsehood by instantiating the universal quantification $\forall x{:}\mathit{real}.\ x^2 \geq 0$. However, as argued by Scott in [Sco79], it may still be useful to use the fact that $X^2 = -1$ in order to derive $X^3 = -X$ or $X^4 + X^2 = 0$, without actually stipulating that these equalities are inconsistent.

If we had a modal theory with names, then perhaps the described equations may be obtained by using the following two names: the name $X : \mathit{real}$ to stands for the indeterminate number, and

[10] This paper is not published, and it is even taken out from Moggi's home page. Perhaps I shouldn't reference it, or if I do, then also give an example rule

the name $P : \mathit{Proof}(X^2 = -1)$ to stand for a non-existent proof that $X^2 = -1$. It is important that X and P are names, rather than ordinary variables; variables will only stand for individuals and proofs that can be exhibited, while names may remain partial. Using X, P and the usual arithmetic properties of real numbers, we can then easily produce a proof Q so that

$$Q : \mathit{Proof}(X^3 = -X)\,[X, P]$$

As expected, this proof would be partial in X and P, and could be turned into a total proof only if witnesses for X and P are exhibited. This partial derivation will actually not be inconsistent, as the proposition $\forall x{:}\mathit{real}.\ x^2 \geq 0$ may not be used to derive contradiction. Indeed, the proposition universally quantifies *only over variables* and thus may not be instantiated with X, which is only a name.[11]

[11] This is very esoteric. Do I really want such wild speculations in the dissertation? Does this even make sense technically?

Chapter 3

Staged computation and meta programming

3.1 Introduction

Staging is a programming technique for explicitly dividing a computation in order to exploit early availability of some arguments [Ers77, GJ95, DP01]. For example, consider filtering a set of points to see on which side of a line defined by two points they lie. This is a typical test used in many convex hull algorithms. The test can be staged by first forming the line and its normal, and then checking the position of each point from the set. Such a staged test obviates the need to repeat the part of the computation pertinent to the normal whenever a new point is tested, and can potentially save a lot of work.

Because it is often quite cumbersome to design programs that fully exploit the natural stage separation of their arguments, it is very desirable for a programming language to provide support for early detection and reporting of staging errors. As an illustration, let us look at the exponentiation function, presented below in ML-like notation.

```
fun exp1 (n : int) (x : int) : int =
    if n = 0 then 1 else x * exp1 (n-1) x
```

The function `exp1 : int -> int -> int` is written in curried form so that it can be applied when only a part of its input is known. For example, if an actual parameter for n is available, `exp1(`n`)` returns a function for computing the n-th power of its argument. From the computational standpoint, however, in most compilers the outcome of this partial instantiation will be a closure waiting to receive an actual parameter for x before it proceeds with evaluation. Thus, one can argue that the following reformulation of `exp1` is preferable.

```
fun exp2 (n : int) : int -> int =
    if n = 0 then λx:int.1
    else
        let val u = exp2 (n - 1)
        in
             λx:int. x * u(x)
        end
```

Indeed, when only n is provided, but not x, the expression `exp2(`n`)` performs computation steps based on the value of n to produce a function specialized for computing the n-th power of its

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

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

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

Figure 3.1: Typing rules for $\lambda^{\Box}$.

argument. In particular, the resulting function will not perform any operations or take decisions at run time based on the value of n; in fact, it does not even depend on n – all the computation steps dependent on n have been taken during the specialization.

A useful intuition for understanding the programming idiom of the above example, is to view `exp2` as a *code generator*; once supplied with n, it *generates* at run time a specialized function for computing n-th powers. This immediately suggests a stratification of expressions into two stages. Object stage (or the stage of generated expressions) comprises of expressions that are to be viewed as *data* – they are result of the process of code generation. In the `exp2` function, such expressions are `(λx:int.1)` and `(λx:int. x * u(x))`. Meta stage (or run-time stage) comprises of expressions that are *executable*, i.e. they describe computational steps to be performed at run time. This is why the above-illustrated programming style is referred to as *staged computation*.

We further postulate that there exists an inclusion from the object stage into the meta stage. In other words, code generated at the object stage as *data*, may be coerced into the meta stage, and *executed*. The opposite inclusion, however, does not exist, and in particular, we prohibit that meta-level variables appear in object-level expressions. For example, in the function `exp2`, the variable n is absent from the expressions `(λx:int.1)` and `(λx:int. x * u(x))`. This restriction guarantees that none of the computation steps dependent on n are postponed beyond the time at which n is specialized to a particular integer value.

As it has been noticed in the previous work [PD01, WLP98, WLPD98], the fragment of the constructive modal logic S4 containing the $\Box$ operator (Chapter 1), and the associated proof-term calculus (called $\lambda^{\Box}$-calculus) are naturally suited to capture many aspects of program staging. We recall the syntax of $\lambda^{\Box}$ below, and the relevant typing rules are presented in Figure 3.1.

$$\begin{array}{llcl}
\textit{Types} & A, B & ::= & P \mid A \to B \mid \Box A \\
\textit{Expressions} & e & ::= & x \mid u \mid \lambda x{:}A.\, e \mid e_1\; e_2 \mid \\
 & & & \mathbf{box}\; e \mid \mathbf{let\ box}\; u = e_1\; \mathbf{in}\; e_2 \\
\textit{Ordinary contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\textit{Modal contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}A
\end{array}$$

The main observation relating staged computation to modal logic is already illustrated by our analysis of the `exp2` function. Since generated code does not depend on meta-level variables, the object expressions are either *closed*, or are computed by substitution out of other object (and therefore closed) expressions. This operational property of the object stage exactly matches the notion of *categorical* proof in modal logic. As defined in Chapter 1.1.3, a categorical proof is closed with respect to value variables, but it may depend on modal variables (which stand for other categorical proofs).

Following the analogy between object expressions and categorical proofs, we can use the type $\Box A$ to classify *generated code* of type A. Under this computational interpretation of the $\lambda^{\Box}$ calculus, the introduction form **box** e serves to coerce the *closed* expression e into the object stage. The elimination form **let box** $u = e_1$ **in** e_2 allows code to be generated by means of substitution: a code generated by e_1 can be substituted for u in e_2. This way, the $\lambda^{\Box}$-calculus makes the distinction between stages explicit. The programmer can specify the intended staging using the term constructors **box** and **let box**. Then the type system can check whether the written program conforms to the staging specifications, turning staging errors into type errors.

Of course, in order to use the $\lambda^{\Box}$-calculus for programming, we need to extend it with some primitive types and recursion. In our examples we will assume the standard ML-like syntax and semantics for natural numbers, integers, booleans and conditionals, recursive functions and pairs. Addition of these features to the $\lambda^{\Box}$-calculus does not present any theoretical problems.

Figure 3.2 presents the small-step operational semantics of $\lambda^{\Box}$. We have decided on a call-by-value strategy which, in addition, prohibits reductions under **box**. Thus, if an expression is boxed, its evaluation will be suspended. Values of modal types are thus boxed *closed* expressions encoding object-level programs.

We can now use the type system of $\lambda^{\Box}$ to make explicit the staging of `exp2`.

```
fun exp3 (n : int) : □(int->int) =
    if n = 0 then box (λx:int. 1)
    else
        let box u = exp3 (n - 1)
        in
            box (λx:int. x * u(x))
        end
```

Application of `exp3` at argument 2 generates a function for squaring.

```
- sqbox = exp3 2;
val sqbox = box (λx:int. x *
                   (λy:int. y *
                      (λz:int. 1) y) x) : □(int -> int)
```

In the elimination form **let box** $u = e_1$ **in** e_2, the bound variable u belongs to the context Δ of modal variables, but it can be used in e_2 in both modal positions (i.e., under a box) and meta positions. Thus the calculus is not only capable of composing generated programs, but can also explicitly force their evaluation. For example we can use the generated function `sqbox` in the following way.

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

This example demonstrates how closed object expressions can be *reflected*, i.e. coerced from the object level into the meta level. The opposite coercion, referred to as *reification*, is not possible. This suggests that $\lambda^{\Box}$ could be given even a more specific model in which reflection naturally exists,

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

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

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

Figure 3.2: Operational semantics of $\lambda^\Box$.

but reification does not. A possible interpretation exhibiting this behavior considers object-level expressions as generated *source* code, i.e. actual closed *syntactic* expressions, or abstract syntax trees of closed $\lambda^\Box$-terms. In contrast, the meta-level expressions are compiled executables. The operation of reflection corresponds to the natural process of compiling a source program into an executable. The opposite operation of reconstructing source code out of its compiled equivalent is not usually feasible, so this interpretation does not support reification, just as required. Furthermore, the typing of $\lambda^\Box$ ensures that only *well-typed* syntactic expressions can be represented in the calculus. This property makes the $\lambda^\Box$ approach to syntax representation reminiscent of the well-known methodology of *higher-order abstract syntax* [PE88].

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

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

3.2 The $\nu^\Box$-calculus

3.2.1 Motivation

If we adhere to the interpretation of categorical proofs as generated *source* code, then the $\lambda^\Box$ staging of `exp3` is rather unsatisfactory. The problem is that the object programs generated by `exp3` (e.g., `sqbox`), contain unnecessary variable-for-variable redexes, and hence are not as optimal as one would want. From the standpoint of syntax manipulation, $\lambda^\Box$ is too restrictive. It cannot express this rather simple way in which well-typed syntactic expressions can be put together to form a more complex syntactic expression. The reason for the deficiency lies in the requirement that the syntactic object expressions that $\lambda^\square$ can represent and manipulate must always be *closed*.

Furthermore, if we only have a type of closed syntactic expressions at our disposal, we can't ever type the *body* of an object-level λ-abstraction in isolation from the λ-binder itself – subterms of a closed term are not necessarily closed themselves. Thus, it would be impossible to ever inspect, destruct or recurse over object-level expressions with binding structure.

What we need in order to avoid the problem of superfluous redexes, but also in order to support code inspection, is the ability to represent *open* expressions and specify *substitution with capture*. This need has long been recognized in the staged computation and meta programming community, and Section 3.6 discusses several different systems and their solution of the problem. The technique predominantly used in these solutions goes back to Davies' $\lambda^\bigcirc$-calculus [Dav96]. The type constructor $\bigcirc$ of this calculus corresponds to discrete temporal logic modality for propositions true at the subsequent time moment. In a meta programming interpretation, the modal type $\bigcirc A$ stands for open object expression of type A, where the free variables of the object expression are modeled by λ-bound variables from the subsequent time moment.

In this chapter, we present a different approach to the problem of spurious redexes. The approach is based on names and the fragment of the modal ν-calculus from Section 2.2 that contains the $\square$ operator. We call this fragment $\nu^\square$-calculus. The idea is to employ names to stand for the free variables of object expressions, and correspondingly, to employ explicit name substitutions to facilitate capture of free variables. Intuitively, the expressions of the $\nu^\square$-calculus are obtained by adjoining names to the expressions of the $\lambda^\square$-calculus. The situation is somewhat analogous to that in polynomial algebra, where one is given a base algebraic structure A and a set of indeterminates (or generators) $\{X_1, \dots, X_n\}$, which are then freely adjoined to A into a structure of polynomials $A[X_1, \dots, X_n]$. In our setup, the indeterminates are the names, and we build "polynomials" over the base structure of $\lambda^\square$ expressions.

When an object expression e contains a name X, we will say that e *depends* on X, or that X is in the *support* of e. For example, assuming for a moment that X and Y are names of type *int*, and that the usual operations of addition, multiplication and exponentiation of integers are primitive in $\nu^\square$, the term

$$e_1 = X^3 + 3X^2Y + 3XY^2 + Y^3$$

would have type *int* and support set $\{X, Y\}$. The names X and Y appear in e_1 at the meta level, and indeed, notice that in order to evaluate e_1 to an integer, we first need to provide definitions for X and Y. On the other hand, if we box the term e_1, we obtain

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

which has the type $\square_{X,Y} \mathit{int}$, but its support is the empty set, as the names X and Y only appear at the object level (i.e., under a box). Thus, the support of a term (in this case e_1) becomes part of the type once the term itself is boxed. This way, the types maintain the information about the support of subterms at all stages. For example, assuming that our language has pairs, the term

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

would have the type $\mathit{int} \times \square_Y \mathit{int}$ with support $\{X\}$.

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

Example 13 Assuming that X and Y are names of type `int`, the $\nu^{\Box}$ segment below creates a "polynomial" expression over X and Y and then evaluates it at the point $(X = 1, Y = 2)$.

```
- let box u = box (X^3 + 3X^2Y + 3XY^2 + Y^3)
  in
      ⟨X -> 1, Y -> 2⟩ u
  end

val it = 27 :  int
```

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

In addition to solving the problem of spurious redexes in staged computation, the $\nu^{\Box}$-calculus has an application in meta programming as well. In Section 3.4, we will extend the $\nu^{\Box}$-calculus with primitives for *intensional code analysis* i.e. pattern matching over syntactic structure of object expressions. It is interesting that intensional code analysis crucially depends on the fact that free variables of syntactic expressions are represented by names, rather than by λ-bound variables (as it is the case in $\lambda^{\bigcirc}$ and other modal type systems based on it). Indeed, imagine a function f that recurses over two expressions with binding structure to compare them for syntactic equality modulo α-conversion. Whenever a λ-abstraction is encountered in both expressions, f needs to introduce a new symbol to stand for the bound variable of that λ-abstraction, and then recursively proceed to compare the bodies of the abstractions. But the construct that generates this new symbol should *not* be a type introduction form. If it were, then the exact number, types and order of symbols that f may generate will be apparent from and fixed by the type of f. As a consequence, f could not be recursively invoked over the bodies of the abstractions, because of a type mismatch.

3.2.2 Syntax and type checking

Here we recall the constructs of the ν-calculus that are relevent for the $\nu^{\Box}$-fragment, and discuss these constructs in terms of their computational application to staging and meta programming. For the logical and type theoretic consideration, we refer the reader to Chapter 2 and Section 2.2.3. The table below recalls the syntax of the $\nu^{\Box}$-calculus.

$$\begin{array}{llcl}
\textit{Names} & X, Y & \in & \mathcal{N} \\
\textit{Supports} & C, D & ::= & \cdot \mid C, X \\
\textit{Types} & A, B & ::= & P \mid A \rightarrow B \mid A \twoheadrightarrow B \mid \Box_C A \\
\textit{Explicit substitutions} & \Theta & ::= & \cdot \mid X \rightarrow e, \Theta \\
\textit{Expressions} & e & ::= & X \mid x \mid \langle\Theta\rangle u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \\
 & & & \mid \textbf{box}\ e \mid \textbf{let box}\ u = e_1\ \textbf{in}\ e_2 \\
 & & & \mid \nu X{:}A.\ e \mid \textbf{choose}\ e \\
\textit{Ordinary contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\textit{Modal contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}A[C] \\
\textit{Name context} & \Sigma & ::= & \cdot \mid \Sigma, X{:}A
\end{array}$$

The type system of $\nu^{\Box}$ consists of two judgments of the modal ν-calculus:

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

Explicit substitutions

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

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

Hypothesis

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

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

λ-calculus

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

Modality

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

Names

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

Figure 3.3: Typing rules of the $\nu^{\Box}$-calculus.

and

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

The first judgment types expressions. Given an expression e it checks whether e has type A, and depends on the support C. The second judgment types explicit substitutions. Given a substitution Θ and two support sets C and D, the substitution has the type $[C] \Rightarrow [D]$ if it maps expressions of support C to expressions of support D.

Both judgments work with three contexts: Σ, Δ and Γ. The name context Σ ascribes types to names. Because each type may contain names, name contexts are dependent. We assume that a name declared in Σ may only be used to the right of its declaration. Context of modal variables Δ ascribes types and supports to modal variables. Modal variables are bound to object expressions – which are syntactic – by the term constructor **let box** $u = e_1$ **in** e_2. Context of value variables Γ ascribes types to ordinary variables (also called value variables). Ordinary variables are introduced into Γ by λ-abstraction, and are bound to expressions from the meta stage. As already described in the previous section, the meta-stage expressions correspond to compiled executables. The typing rules of the $\nu^\Box$-calculus are presented in Figure 3.3, and discuss them next.

A pervasive characteristic of the type system is *support weakening*. If the names that an expression depends on are contained in the support set C, then they are certainly contained in any support $D \supseteq C$. We recall here the formal statement of the support weakening principle for the two judgments of the $\nu^\Box$-calculus. The proof of the support weakening principle, as well as the proofs of the other formal statements that we present here, may be found in Section 2.2.3.

Principle (Support weakening)
Support weakening is covariant on the right-hand side and contravariant on the left-hand side of the judgments. More formally, let $C \subseteq D \subseteq \mathrm{dom}(\Sigma)$ be well-formed supports. Then the following holds:

1. if $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash e : A\,[D]$
2. if $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [C]$, then $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C_1] \Rightarrow [D]$
3. if $\Sigma; (\Delta, u{::}A[D]); \Gamma \vdash e : B\,[C_1]$, then $\Sigma; (\Delta, u{::}A[C]); \Gamma \vdash e : B\,[C_1]$
4. if $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [D] \Rightarrow [C_1]$, then $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [C_1]$

Explicit substitutions. As explained in Chapter 2, applying the empty substitution over a given term, does not change the term itself – the empty substitution corresponds to the identity function on expressions. Thus, when an empty substitution is applied to a term containing names from C, the resulting term obviously contains the same names. The typing rule for empty substitutions formalizes this property. We also allow weakening to an arbitrary superset D, in order to ensure that the support weakening principle holds. We implicitly require that both the sets are well-formed; that is, they both contain only names already declared in the name context Σ. The rule for non-empty substitutions recursively checks if each of component expressions is well-typed.

The result of applying the substitution Θ over an expression e is denoted as $\{\Theta\}e$. We denote by $\Theta_1 \circ \Theta_2$ the composition of the substitutions Θ_1 and Θ_2. Both of these operations are formally defined in Section 2.2.3.

When an explicit substitution $\Theta : [C] \Rightarrow [D]$ is applied over an expression $e : A\,[C]$, the result $\{\Theta\}e$ will have support D. Consider for example the explicit substitution $\Theta = (X \to 10, Y \to 20)$, with domain $\mathrm{dom}(\Theta) = \{X, Y\}$. This substitution 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 $[Z]$. This typing property of explicit substitutions are summarized by the following *explicit substitution principle*.

Principle (Explicit substitution)

Let $\Sigma; \Delta; \Gamma \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$. Then the following holds:

1. if $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$ then $\Sigma; \Delta; \Gamma \vdash \{\Theta\}e : A\,[D]$
2. if $\Sigma; \Delta; \Gamma \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$, then $\Sigma; \Delta; \Gamma \vdash \langle\Theta \circ \Theta'\rangle : [C_1] \Rightarrow [D]$

Hypothesis rules. Because there are three kinds of variable contexts, we have three hypothesis rules. First is the rule for names. A name X can be used provided it has been declared in Σ and is accounted for in the supplied support set. The implicit assumption is that the support set C is well-formed; that is, $C \subseteq \mathrm{dom}\,(\Sigma)$. The rule for value variables is straightforward. The typing $x{:}A$ can be inferred, if $x{:}A$ is declared in Γ. The actual support of such a term can be any support set C as long as it is well-formed, which is implicitly assumed. Modal variables occur in a term always prefixed with an explicit substitution. The rule for modal variables has to check if the modal variable is declared in the context Δ and if its corresponding substitution has the appropriate type.

λ-calculus fragment. The rule for λ-abstraction is quite standard. Its implicit assumption is that the argument type A is well-formed in name context Σ before it is introduced into the variable context Γ. The application rule checks both the function and the application argument against the same support set. Associated with the λ-calculus fragment is the *value substitution principle.*

Principle (Value substitution)
Let $\Sigma; \Delta; \Gamma \vdash e_1 : A\,[C]$. Then the following holds:

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

Modal fragment. Just as in $\lambda^{\Box}$-calculus, the meaning of the rule for $\Box$-introduction is to ensure the staging separation between expressions. In the term **box** e, the expression e belongs to the object stage, and may be treated as a syntactic entity. Correspondingly, the typing rule for **box** must typecheck e against an empty context of value variables Γ. Indeed, value variables are bound to meta-level expressions, and meta-level expressions correspond to compiled executables. If e is to be syntactic, it must not depend on compiled code.

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

Associated with modal variables and with the modal fragment of the calculus is the operation of *modal substitution* $[\![e/u]\!]e_2$, where u is a modal variable, and e is a *closed syntactic* expression. The operation substitutes e for u in e_2, but so that e is first transformed by the explicit substitution associated with each occurrence of u in e_2. For example, the following are the two most characteristic clauses in the definition of modal substitution.

$$\begin{array}{lcll} [\![e/u]\!]\langle\Theta\rangle u & = & \{[\![e/u]\!]\Theta\}e & \\ [\![e/u]\!]\langle\Theta\rangle v & = & \langle[\![e/u]\!]\Theta\rangle v & \quad u \neq v \end{array}$$

Note that the first clause of the definition actually applies to explicit substitution Θ to e. The typing properties of this operation are formally stated in the *modal substitution principle* below. Again, the complete definition of modal substitution and the proof of the modal substitution principle can be found in Section 2.2.3.

Principle (Modal substitution)
Let $\Delta; \cdot \vdash e : A\,[C]$. Then the following holds:

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

Names fragment. The introduction form for names is $\nu X{:}A.\ e$ with its corresponding type $A \not\twoheadrightarrow B$. It introduces a name $X{:}A$ into the computation determined by e. It is assumed that the type A is well-formed relative to the context Σ. The term constructor choose is the elimination form for $A \not\twoheadrightarrow B$. It picks a fresh name and substitutes it for the bound name in the ν-abstraction. In other words, the operational semantics of the redex **choose** $(\nu X{:}A.\ e)$ (formalized in Section 3.2.3) proceeds with the evaluation of e in a run-time context in which a fresh name has been picked for X. It is justified to do so because X is bound by ν and, by convention, can be renamed with a fresh name. In the ν-introduction rule, it is assumed that the name X is completely new – it does not appear in the contexts of the judgment, and in particular, it does not appear in the type B and support C. This typing discipline effectively limits X to appear only in subterms of e which are not encountered during evaluation (i.e. dead-code subterms), or in subterms from which it will eventually be removed by some explicit substitution. For example, consider the following expression.

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

This expression contains a substituted occurrence of X and a dead-code occurrence of Y, and is therefore well-typed (of type $int \not\twoheadrightarrow int \not\twoheadrightarrow \Box int$). Another way to paraphrase this typing discipline is the following: in order to prevent the name bound in $\nu X{:}A.\ e$ from escaping the scope of its definition, when leaving this scope we have to turn the "polynomials" depending on X into functions. An illustration of this technique is the program presented in Example 14. The described aspect of fresh name generation is important because it it ensures the preservation and progress properties of $\nu^{\Box}$ (Theorems 17 and 18). Indeed, if during evaluation, X is encountered outside its defining ν, the evaluation will get stuck, because there are no expression to substitute for X.

We will frequently abbreviate the β-redex

$$\textbf{choose}\ (\nu X{:}A.\ e)$$

simply as

$$\textbf{let name}\ X{:}A\ \textbf{in}\ e.$$

In fact, it will become apparent from the future examples in this document, that the only way we actually use **choose** and ν is in some β-redex **choose** $(\nu X{:}A.\ e)$, and never in isolation from each other. Of course, all of these uses may have been abbreviated into a **let name** construct, which raises the following question: why not define **let name** as primitive and omit **choose** and ν? The answer lies in the logical considerations from Section 2.1.6. If **let name** is taken as primitive, then the judgment $\Sigma; \Delta; \Gamma \vdash A\ \mathit{true}\,[C]$ obtained by erasing the proof term e from $\Sigma; \Delta; \Gamma \vdash e : A\,[C]$ would not be directed by the syntactic structure of the propositions A.

Example 14 To illustrate the language constructors, we present a version of the staged exponentiation function that we can write in $\nu^{\Box}$-calculus. In this and in other examples we resort to concrete syntax in ML fashion, and assume the presence of the base type of integers, recursive functions and let-definitions.

```
fun exp (n : int) : □(int -> int) =
    let name X : int
        fun exp' (m : int) : □_Xint =
            if m = 0 then box 1
            else
                let box u = exp' (m - 1)
                in
                    box (X * u)
                end
        box v = exp' (n)
    in
        box (λx:int. ⟨X -> x⟩ v)
    end

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

The function `exp` takes an integer n and generates a fresh name X of integer type. Then it calls the helper function `exp'` to build the expression $v = \underbrace{X * \cdots * X}_{n} * 1$ of type `int` and support $\{X\}$. Finally, it turns the expression v into a function by explicitly substituting the name X in v with a newly introduced bound variable x, incurring capture. Notice that the generated residual code for `sq` does not contain any unnecessary redexes, in contrast to the $\lambda^{\Box}$ version of the program from Section 3.1. ■

Example 15 This example presents the function `conv` for computing the convolution of two integer lists. Convolution of lists $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_n]$, is the list $[x_n y_1, \ldots, x_1 y_n]$. We ignore the possibility that the two lists can be of different sizes.

The function `conv`, which we present in Figure 3.4, is staged in the first argument, so that given the list x, `conv` outputs a source code specialized for computing the convolution with x. In this example, we assume the existence of a function `lift` : `int` $\rightarrow$ $\Box$`int`, mapping each integer n into **box** n. This is a reasonable assumption, as the base type of integers is always considered observable; in any realistic situation, it would be possible to coerce an integer value into its own syntactic representation. The helper function `conv'` recurses over the list x to build the output code; it keeps the unfinished part of the output abstracted using the variable $\mathtt{z{:}}\Box_{\mathtt{TL}}\mathtt{intlist}$.

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

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

fun conv (xs : intlist) =
  let name TL:intlist
  (*
   * conv' : intlist -> □TLintlist
   *           -> □(intlist -> intlist)
   *)

     fun conv' (nil) =
     λz:□TLintlist.
        let box u = z
        in
            box (λy:intlist.
                   <TL -> y>u)
        end
```

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

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

Figure 3.4: Staged convolution.

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

It remains a challenge to write a $\nu^{\Box}$ program that could generate even more concise specialized code, like for example the following fragment for convolution with [3,2]:

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

■

3.2.3 Operational semantics

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

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

which relates an expression e with its one-step reduct e'. The expressions e and e' do not contain any free variables, but they may contain free names. However, we require that e and e' must have *empty support*. In other words, we only consider for evaluation those terms whose names appear exclusively in boxed subterms, or are otherwise captured by some explicit substitution. Because free names are allowed under these conditions, the operational semantics has to keep track of them

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

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

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

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

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

Figure 3.5: Structured operational semantics of $\nu^{\Box}$-calculus.

in the run-time name contexts Σ and Σ'. The rules of the judgment are given in Figure 3.5, and the values of the language are generated by the grammar below.

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

The rules agree with the β-reductions from Section 2.2.3, and are standard except for two important observations. First of all, the β-redex for the type constructor $\twoheadrightarrow$ extends the run-time context with a fresh name before proceeding. This way, we keep track of names that have been generated in the course of evaluation, so that we can select a fresh name when it is needed.

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

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

Theorem 17 (Type preservation)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash e' : A\,[\,]$.

Proof: By a straightforward induction on the structure of e using the substitution principles. ■

Theorem 18 (Progress)
If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$, then either

1. *e is a value, or*

2. *there exist a term e' and a context Σ', such that $\Sigma, e \longmapsto \Sigma', e'$.*

Proof: By a straightforward induction on the structure of e. ■

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

Theorem 19 (Determinacy)
If $\Sigma, e \longmapsto^n \Sigma_1, e_1$, and $\Sigma, e \longmapsto^n \Sigma_2, e_2$, then there exists a permutation of names $\pi : \mathcal{N} \to \mathcal{N}$, fixing $\mathrm{dom}(\Sigma)$, such that $\Sigma_2 = \pi(\Sigma_1)$ and $e_2 = \pi(e_1)$.

Proof: By induction on the length of the reductions, using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$ and π is a permutation on names, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$. The only interesting case is when $n = 1$ and $e = \textbf{choose}\ (\nu X{:}A.\ e')$. In that case, it must be $e_1 = [X_1/X]e'$, $e_2 = [X_2/X]e'$, and $\Sigma_1 = (\Sigma, X_1{:}A)$, $\Sigma_2 = (\Sigma, X_2{:}A)$, where $X_1, X_2 \in \mathcal{N}$ are fresh. Obviously, the involution $\pi = (X_1\ X_2)$ which swaps these two names has the required properties. ■

3.3 Support polymorphism

It is frequently necessary to write programs that are polymorphic in the support of their arguments, because they manipulate syntactic expressions of unknown support. A typical example is a function that recurses over an expression with binding structure. When this function encounters a λ-abstraction, it has to place a fresh name instead of the bound variable, and recursively continue scanning the body of the λ-abstraction, which is itself a syntactic expression but depending on this newly introduced name[1]. For such uses, we extend the $\nu^{\Box}$-calculus with a notion of explicit *support polymorphism* in the style of Girard and Reynolds [Gir86, Rey83].

To add support polymorphism to the simple $\nu^{\Box}$-calculus, we create a new syntactic category of *support variables*, which stand for unknown support sets. Then the rest of the syntax of $\nu^{\Box}$ is extended to take support variables into account. We summarize the changes in the following table.

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

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

Before a support variable can be used, it has to be declared in the name context Σ. For the new definition of Σ, we retain the same well-formedness conditions as before. In particular, a support variable $p \in \Sigma$ may only be used to the right of its declaration. It is important that supports themselves are allowed to contain support variables, to express the situation in which only a portion of a support set is known. Consequently, the function $\mathrm{fn}(-)$ is updated to return the set of names *and support variables* appearing in its argument. The family of types is extended with the type $\forall p.\ A$ expressing universal support quantification. Its introduction form is $\Lambda p.\ e$, which binds a support variable p in the expression e. This Λ-abstraction will also be a value in the extended operational semantics. The corresponding elimination form is the application $e\ [C]$ whose meaning is to instantiate the unknown support set abstracted in e with the provided support set C.

The typing judgment has to be instrumented with new rules for typing support-polymorphic abstraction and application.

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

The $\forall$-introduction rule requires that the bound variable p is a fresh support variable, as customary in binding forms. In particular, $p \notin \Sigma$, and consequently, $p \notin \Delta, \Gamma, \mathrm{fn}(A[C])$. The rule for $\forall$-elimination substitutes the argument support set D into the type A. It assumes that D is well-formed relative to the context Σ; that is, $D \subseteq \mathrm{dom}(\Sigma)$. The operational semantics for the new constructs is also not surprising.

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

The extended language satisfies the following substitution principle.

Lemma 20 (Support substitution principle)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \mathrm{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Then the following holds.

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

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

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

1. by derivation, $\Sigma; \Delta; \Gamma \vdash e : A\,[C_2]$ and $\Sigma; \Delta; \Gamma \vdash \Theta_1 : [C_1 \setminus \{X\}] \Rightarrow [C_2]$

2. by first induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash e' : A' \, [C_2']$
3. by second induction hypothesis, $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : [(C_1 \setminus \{X\})'] \Rightarrow [C_2']$
4. because $(C_1' \setminus \{X\}) \subseteq (C_1 \setminus \{X\})'$, by support weakening (Lemma 10.5), $(\Sigma_1, \Sigma_2'); \Delta'; \Gamma' \vdash \Theta_1' : [C_1' \setminus \{X\}] \Rightarrow [C_2']$
5. result follows from (2) and (4) by the typing rule for non-empty substitutions

■

The structural properties presented in Section 2.2.3 readily extend to the new language with support polymorphism. The same is true of the type preservation (Theorem 17) and progress (Theorem 18) whose additional cases involving support abstraction and application are handled using the above Lemma 20.

Example 16 In a support-polymorphic $\nu^{\Box}$-calculus we can slightly generalize the program from Example 14 by pulling out the helper function `exp'` and parameterizing it over the exponentiating expression. In the following program, we use `[p]` in the function definition as a concrete syntax for Λ-abstraction of a support variable `p`.

```
fun exp' [p] (e : □_pint) (n : int) : □_pint =
    if n = 0 then box 1
    else
       let box u = exp' [p] e (n - 1)
           box w = e
       in
           box (u * w)
       end

fun exp (n : int) : □(int -> int) =
    let name X : int
        box w = exp' [X] (box X) n
    in
        box (λx:int. ⟨X -> x⟩ w)
    end

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

■

Example 17 As an example of a more realistic program we present the regular expression matcher from [DP01] and [Dav96]. The example assumes the declaration of the datatype of regular expressions:

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

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

fun acc1 (Empty) k s = k s

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

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

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

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


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

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

Figure 3.6: Unstaged regular expression matcher.

We also assume a primitive predicate `null : char list -> bool` for testing if the input list of characters is empty. Figure 3.6 presents an ordinary ML implementation of the matcher, and $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ versions can be found in [DP01, Dav96]. The helper function `acc1` in Figure 3.6 takes a regular expression e, a continuation function k, and an input string s (represented as a list of characters). The function attempts to match a *prefix* of s to the regular expression e. If the matching succeeds, the remainder of s is passed to the continuation k to determine if s is accepted or not.

We now want to use the $\nu^{\Box}$-calculus to stage the program from Figure 3.6 so that it can be *specialized* with respect to a given regular expression. For that purpose, it is useful to view the helper function `acc1` from Figure 3.6 as a code generator. Indeed, `acc1` may be seen as follows: it first generates code for matching a string against a regular expression e, and then appends k to that code. This is the main idea behind the function `acc`, and the $\nu^{\Box}$ program in Figure 3.7. In this program, we use the name S for the input string to be matched by the code that `acc` generates. The continuation k is not a function anymore, but code to be attached at the end of the generated result. We want code k to contain further names standing for yet unbound variables, and hence the support-polymorphic typing `acc : regexp  -> `$\forall$`p.(`$\Box_{S,p}$`bool -> `$\Box_{S,p}$`bool)`. The support polymorphism pays off when generating code for alternation `Plus(`e_1`, `e_2`)` and iteration `Star(`e`)`. For example, observe in the alternation case that the generated code does not duplicate the "continuation" code of k. Rather, k is emitted as a separate function which is a joining point for the computation branches corresponding to e_1 and e_2. Similarly, in the case of iteration, we set up a loop in the output code that would attempt zero or more matchings against e. The support polymorphism of `acc` enables us to produce code in chunks without knowing the exact identity of

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

fun accept (e : regexp) =
  let name S : char list

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

      fun acc (Empty) [p] k = k

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

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

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

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

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

Figure 3.7: Regular expression matcher staged in the $\nu^{\Box}$-calculus.

```
▷ accept (Star (Empty))

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

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

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

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

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

Figure 3.8: Example execution trace for a regular expression matcher in $\nu^{\Box}$. Function calls are marked by ▷ and the corresponding return results are marked by an aligned ◁.

the above-mentioned joining or looping points. Once all the parts of the output code are generated, we just stitch them together by means of explicit substitutions.

At this point, it may be illustrative to trace the execution of the program on a concrete input. Figure 3.8 presents the function calls and the intermediate results that occur when the $\nu^{\Box}$ matcher is applied to the regular expression `Star(Empty)`. The resulting specialized program does not contain variable-for-variable redexes, thanks to the features and expressiveness of $\nu^{\Box}$, but it unnecessarily tests if `t = t`. Removing these extraneous tests requires some further examination and preprocessing of e, but the thorough description of such a process is beyond our scope. We refer to [Har99] for an insightful analysis. ■

3.4 Intensional program analysis

3.4.1 Syntax and type checking

As explained in Section 3.2, it is possible to intuitively consider the type $\Box_C A$ as the set of closed syntactic expressions e, such that $\Sigma; \cdot; \cdot \vdash e : A\,[C]$. The calculus presented so far contains constructs for creating elements of type $\Box_C A$, but it is impossible to inspect the syntactic structure of these elements, let alone take them apart.

In this section, we extend the support-polymorphic $\nu^{\Box}$-calculus with primitives for pattern matching against syntactic expressions with binding structure. Our extension is limited to only test if an expression is a name, a λ-abstraction or an application, and limit all other cases for future work. It is not clear, however, whether the expressiveness of pattern matching can be extended to handle a larger subset of the object stage of $\nu^{\Box}$, without significant additions to the meta stage. The problem is that any such addition would require extensions to pattern match against the additions, which would itself require new extensions to the meta stage, and so on.

The syntactic extensions that we consider in this section are summarized in the the table below.

$$\begin{array}{llll}
\textit{Pattern variables} & w & \in & \mathcal{W} \\
\textit{Higher-order patterns} & \pi & ::= & (w\ x_1 \ldots x_n){:}A[C] \mid X \mid x \mid \lambda x{:}A.\ \pi \mid \pi_1\ \pi_2 \\
\textit{Pattern assignments} & \sigma & ::= & \cdot \mid w \to e, \sigma \\
\textit{Terms} & e & ::= & \ldots \mid \textbf{case}\ e\ \textbf{of}\ \textbf{box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2
\end{array}$$

We use higher-order patterns [Mil90] to match against syntactic expressions with binding structure. In higher-order patterns, we distinguish between *pattern variables* and *bound variables.* Pattern variables are placeholders intended to bind syntactic subexpressions in the process of matching and pass them to the subsequent computation. Bound variables are introduced by patterns for binding structure $\lambda x{:}A.\ \pi$ and are syntactic entities that can match only themselves. We use x, y and variants to range over bound variables, and w and variants to range over pattern variables.

The basic pattern $(w\ x_1 \ldots x_n){:}A[C]$ declares a pattern variable w which matches a syntactic expression of type A and support C subject to the condition that the expression's bound variables are among $x_1, \ldots, x_n$. Pattern X matches a name X from the global name context. Pattern $\lambda x{:}A.\ \pi$ matches a λ-abstraction of domain type A. It declares a new bound variable x which is local to the pattern, and demands that the body of the matched expression conforms to the pattern π. The bound variable x matches only the pattern x. Pattern $\pi_1\ \pi_2$ matches a syntactic expression representing application. Notice that the decision to explicitly assign types to every pattern variable forces the pattern for application to be monomorphic. In other words, the application pattern cannot match a pair of expressions representing a function and its argument if the domain type of the function is now known in advance. It is an important future work to extend intensional analysis to allow patterns which are type-polymorphic in this sense. The patterns are assumed to be linear, i.e. no pattern variable occurs twice.

The typing judgment for patterns has the form

$$\Sigma; \Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1.$$

The judgment is hypothetical in the global context of names Σ, and the context of *locally declared* bound variables Γ. It checks if the pattern π has type A and support C and if the pattern variables from π conform to the typings given in the residual context Γ_1. The typing rules are presented in Figure 3.9. Most of them are straightforward and we do not explain them, but the rule for pattern variables deserves special attention. As it shows, in order for the pattern expression $(w\ x_1 \ldots x_n){:}A[C]$ to be well-typed, the bound variables $x_1{:}A_1, \ldots, x_n{:}A_n$ have to be declared in the local context Γ. We also allow *strengthening of the support*: if w is required to match expressions of support C, than any expression with support $D \subseteq C$ is eligible for matching. If the pattern expression $(w\ x_1 \ldots x_n){:}A[C]$ is well-typed, then w will match only expressions of type A with the given bound variables and the names declared in D. *The residual context types w as a function over types $\Box_p A_i$ with polymorphic support.* This hints at the operational semantics that will be assigned to higher-order patterns. If an expression e with a local bound variable $x{:}A$ matches to a pattern variable w, then w will residualize to a meta-level function whose meaning is as follows: it takes a syntactic expression $e'{:}A$ and returns back the syntactic expression $[e'/x]e$.

In order to incorporate pattern matching into $\nu^{\Box}$, the syntax is extended with a new term constructor **case** e **of box** $\pi \Rightarrow e_1$ **else** e_2. The intended operational interpretation of **case** is to evaluate the argument e to obtain a boxed expression **box** e', then match e' to the pattern π. If

$$\frac{D \subseteq C \qquad p \notin \Sigma}{\begin{array}{l}\Sigma; (\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n) \vdash ((w\ x_1 \ldots x_n){:}A[D]) : A\,[C] \\ \qquad\qquad\qquad\qquad \Longrightarrow w{:}\forall p.\ \Box_p A_1 \to \cdots \to \Box_p A_n \to \Box_{p,D} A\end{array}}$$

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

$$\frac{\Sigma; (\Gamma, x{:}A) \vdash \pi : B\,[C] \Longrightarrow \Gamma_1}{\Sigma; \Gamma \vdash \lambda x{:}A.\ \pi : A \to B\,[C] \Longrightarrow \Gamma_1}$$

$$\frac{\Sigma; \Gamma \vdash \pi_1 : A \to B\,[C] \Longrightarrow \Gamma_1 \qquad \Sigma; \Gamma \vdash \pi_2 : A\,[C] \Longrightarrow \Gamma_2 \qquad \mathrm{fn}(A) \subseteq \mathrm{dom}(\Sigma)}{\Sigma; \Gamma \vdash \pi_1\ \pi_2 : B\,[C] \Longrightarrow (\Gamma_1, \Gamma_2)}$$

Figure 3.9: Typing rules for patterns.

the matching is successful, it creates an environment with bindings for the pattern variables, and then evaluates e_1 in this environment. If the matching fails, the branch e_2 is taken.

Example 18 Consider the (rather restricted) function `reduce` that takes a syntactic expression of type A, and checks if it is a β-redex $(\lambda x{:}A.\ w_1)\ (w_2)$. If the answer is yes, it applies the "call-by-value" strategy: it reduces w_2, substitutes the reduct for x in w_1 and then continue reducing thus obtained expression. If the answer is no, it simply returns the argument.

```
fun reduce (e : □A) : □A =
    case e of
         box ((λx:A. ((w1 x):A[])) (w2:A[])) =>
                (* w1 : ∀q. □_qA -> □_qA *)
                (* w2 : ∀q. □_qA *)
             let val e2 = reduce (w2 [])
             in
                  reduce (w1 [] e2)
             end
         else e
```

Ideally, one would want to `reduce` an arbitrary expression, not just simple top-level redexes. We cannot currently write such a function mainly because our language lacks type-polymorphic patterns and type-polymorphic recursion. In particular, if the syntactic argument we are dealing with is an application of a general term of type $A \to A$ rather than a λ-abstraction, we cannot recursively reduce that term first unless the language is equipped with type-polymorphic recursion.

Nevertheless, `reduce` is illustrative of the way higher-order patterns work. Patterns transform an expression with a bound variable into a function on syntax that substitutes the bound variable with the argument. *That way we can employ meta-level reduction to perform object-level substitution.* This is reminiscent of the idea of normalization-by-evaluation [BS91, BES98] and type-directed partial evaluation [Dan96]. ■

The typing rule for **case** is:

$$\frac{\Sigma;\Delta;\Gamma \vdash e : \Box_D A\,[C] \quad \Sigma;\cdot \vdash \pi : A\,[D] \Longrightarrow \Gamma_1 \quad \Sigma;\Delta;(\Gamma,\Gamma_1) \vdash e_1 : B\,[C] \quad \Sigma;\Delta;\Gamma \vdash e_2 : B\,[C]}{\Sigma;\Delta;\Gamma \vdash \textbf{case}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2 : B\,[C]}$$

Observe that the second premise of the rule requires an empty variable context, so that patterns cannot contain outside value or modal variables. However (and this is important), they can contain names. It is easy to incorporate the new syntax into the language. We first extend explicit substitution over the new **case** construct

$$\{\Theta\} \quad (\textbf{case}\ e\ \textbf{of box}\ \pi \Rightarrow e_1\ \textbf{else}\ e_2) \quad = \quad \textbf{case}\ (\{\Theta\}e)\ \textbf{of box}\ \pi \Rightarrow (\{\Theta\}e_1)\ \textbf{else}\ (\{\Theta\}e_2)$$

and similarly for expression substitution, and then all the structural properties derived in Section 2.2.3 easily hold. The only complication comes in handling names and support substitution because patterns are allowed to depend on names and support variables from the global context Σ. However, the lemmas below establish the required invariants.

Lemma 21 (Structural properties of pattern matching)

1. **Exchange** *Let Σ', Γ' and Γ'_1 be well-formed contexts obtained by permutation from Σ, Γ and Γ_1 respectively and $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $\Sigma';\Gamma' \vdash \pi : A\,[C] \Longrightarrow \Gamma'_1$*

2. **Weakening** *Let $\Sigma \subseteq \Sigma'$ and $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $\Sigma';\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$*

Proof: By straightforward introduction on the structure of the typing derivations. ■

Lemma 22 (Support substitution principle for pattern matching)
Let $\Sigma = (\Sigma_1, p, \Sigma_2)$ and $D \subseteq \text{dom}(\Sigma_1)$ and denote by $(-)'$ the operation of substituting D for p. Assume also that $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$. Then $(\Sigma_1,\Sigma'_2);\Gamma' \vdash \pi' : A'\,[C'] \Longrightarrow \Gamma'_1$.

Proof: By straightforward induction on the structure of π. ■

3.4.2 Operational semantics

Operational semantics for pattern matching is established by the new judgment

$$\Sigma;\Gamma \vdash e \rhd \pi \Longrightarrow \sigma$$

which reads: in a global context of names and support variables Σ and a context of locally declared free variables Γ the matching of the expression e to the pattern π generates an assignment of values σ to the pattern variables of π. The rules for this judgment are given in Figure 3.10. Most of the rules are self-evident, but the rule for pattern variables deserves more attention. Its premise requires a run-time typecheck of the expression e, in order to preserve soundness. Because of this reason, the judgment for operational semantics of $\nu^\Box$-calculus with pattern matching must keep track of a run-time name context Σ. The context Σ not only lists the used names, but it also *assigns types* to the used names. The following lemma relates the typing judgment for patterns and their operational semantics.

Lemma 23 (Soundness of pattern matching)
Let π be a pattern such that $\Sigma;\Gamma \vdash \pi : A\,[C] \Longrightarrow \Gamma_1$, where $\Gamma_1 = (w_1{:}A_1,\ldots,w_n{:}A_n)$. Furthermore, let e be an expression matching π to produce a pattern assignment σ, i.e. $\Sigma;\Gamma \vdash e \rhd \pi : A \Longrightarrow \sigma$. Then $\sigma = (w_1 \to e_1,\ldots,w_n \to e_n)$ where $\Sigma;\cdot;\cdot \vdash e_i : A_1$, for every $i = 1,\ldots,n$.

$$\frac{\Sigma; \cdot; (x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e : A\,[D]}{\begin{array}{l}\Sigma; (\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e \rhd ((w\ x_1 \ldots x_n){:}A[D]) : A \\ \qquad\qquad\qquad \Longrightarrow [w \to \Lambda p.\ \lambda y_i{:}\Box_p A_i.\ \mathbf{let\ box}\ x_i = y_i\ \mathbf{in\ box}\ e]\end{array}}$$

$$\frac{}{(\Sigma, X{:}A); \Gamma \vdash X \rhd X : A \Longrightarrow \cdot} \qquad \frac{}{\Sigma; (\Gamma, x{:}A) \vdash x \rhd x : A \Longrightarrow \cdot}$$

$$\frac{\Sigma; (\Gamma, x{:}A) \vdash e \rhd \pi : B \Longrightarrow \sigma}{\Sigma; \Gamma \vdash \lambda x{:}A.\ e \rhd \lambda x{:}A.\ \pi : (A \to B) \Longrightarrow \sigma}$$

$$\frac{\Sigma; \Gamma \vdash e_1 \rhd \pi_1 : A \to B \Longrightarrow \sigma_1 \qquad \Sigma; \Gamma \vdash e_2 \rhd \pi_2 : A \Longrightarrow \sigma_2}{\Sigma; \Gamma \vdash e_1\ e_2 \rhd \pi_1\ \pi_2 : B \Longrightarrow (\sigma_1, \sigma_2)}$$

Figure 3.10: Operational semantics for pattern matching.

Notice that in the lemma we did not require that e be well-typed, or even syntactically well-formed. If it were not well-formed, the matching simply would not succeed.

Proof: By induction on the structure of π. We present the base case below.

case $\pi = (w\ x_1 \ldots x_n){:}A[D]$, where $\Gamma = \Gamma_2, x_i{:}A_i$.

1. let $e' = (\Lambda p.\ \lambda y_i{:}\Box_p A_i.\ \mathbf{let\ box}\ x_i = y_i\ \mathbf{in\ box}\ e)$ and $A' = \forall p.\ \Box_p A_1 \to \cdots \to \Box_p A_n \to \Box_{p,D} A$
2. by typing derivation, $D \subseteq C$ and $x_i{:}A_i \in \Gamma$ and also $\Gamma_1 = (w{:}A')$
3. by matching derivation, $\Sigma; \cdot; (x_1{:}A_1, \ldots, x_n{:}A_n) \vdash e : A\,[D]$, and $\sigma = (w \to e')$
4. by straightforward structural induction, $\Sigma; (x_1{:}A_1, \ldots, x_n{:}A_n); \cdot \vdash e : A\,[D]$
5. it is simply to show now that, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash e : A\,[D, p]$
6. and thus also, $(\Sigma, p); (x_1{:}A_1[p], \ldots, x_n{:}A_n[p]); \cdot \vdash \mathbf{box}\ e : \Box_{D,p} A\,[\,]$
7. and $(\Sigma, p); \cdot; (y_1{:}\Box_p A_1, \ldots, y_n{:}\Box_p A_n) \vdash \mathbf{let\ box}\ x_i = y_i\ \mathbf{in\ box}\ e : \Box_{D,p} A\,[\,]$
8. and finally, $\Sigma; \cdot; \cdot \vdash e' : A'\,[\,]$

■

The last piece to be added is the operational semantics for the **case** statement, and the required rules are given below. Notice that the premise of last rule makes use of the fact that the operational semantics for patterns is deterministic; the rule applies if the expression and e and the pattern π cannot be matched.

$$\frac{\Sigma, e \longmapsto \Sigma', e'}{\Sigma, (\mathbf{case}\ e\ \mathbf{of\ box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2) \longmapsto \Sigma', (\mathbf{case}\ e'\ \mathbf{of\ box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2)}$$

$$\frac{\Sigma; \cdot \vdash e \rhd \pi : A \Longrightarrow (w_1 \to e'_1, \ldots, w_n \to e'_n)}{\Sigma, (\mathbf{case\ box}\ e\ \mathbf{of\ box}\ \pi \Rightarrow e_1\ \mathbf{else}\ e_2) \longmapsto \Sigma, [e'_1/w_1, \ldots, e'_n/w_n]e_1}$$

$$\frac{\Sigma; \cdot \vdash e \rhd \pi \not\Longrightarrow \sigma \qquad \text{for any } \sigma}{\Sigma, (\textbf{case box } e \textbf{ of box } \pi \Rightarrow e_1 \textbf{ else } e_2) \longmapsto \Sigma, e_2}$$

Finally, using the lemmas established in this section, we can easily augment the proof of the preservation and progress theorems (Theorem 17 and 18) to cover the extended language. The statements of the theorems are unchanged.

Example 19 The following examples present a generalization of our old exponentiation function. Instead of computing only powers of integers, we can compute powers of functions too, i.e. have a functional for mapping $f \mapsto \lambda x.\ (fx)^n$. The functional is passed the source code for f, and an integer n, and returns the source code for $\lambda x.\ (fx)^n$. The idea is to have the resulting source code be as optimized as possible, while still computing the extensionally same result. We rely on programs presented in Section 3.2 and Examples 14 and 16.

For comparison, we first present a $\lambda^{\Box}$ version of the function-exponentiating functional.

```
fun fexp1 (f : □(int->int)) (n : int) : □(int->int) =
    let box g = f
        box p = exp3 n
    in
        box (λv:int. (p (g v)))
    end

- fexp1 (box λw:int. w + 1) 2;
val it = box (λv:int. (λx.x*(λy.y*(λz.1)y)x) ((λw.w+1)v)) : □(int->int)
```

Observe that the residual program contains a lot of unnecessary redexes. As could be expected, the $\nu^{\Box}$-calculus provides a better way to stage the code[2], simply by using the function `exp` from Example 14 instead `exp3` from Section 3.1.

```
fun fexp2 (f : □(int->int)) (n : int) : □(int->int) =
    let box g = f
        box p = exp n
    in
        box (λv:int. p (g v))
    end

-fexp2 (box λw:int. w + 1) 2;
val it = box (λv:int. (λx.x*(x*1)) ((λw.w+1) v)) : □(int->int)
```

In fact, there is at least one other way to program this functional: we can eliminate the outer β-redex from the residual code, at the price of duplicating the inner one.

[2] And similar programs can be written in $\lambda^{\bigcirc}$ and MetaML, as well.

```
fun fexp3 (f : □(int->int)) (n : int) : □(int->int) =
    let name X : int
        box g = f
        box e = exp' [X] (box (g X)) n
    in
        box (λv:int. ⟨X -> v⟩e)
    end

- fexp3 (box (λw:int. w + 1)) 2;
val it = box (λv:int. ((λw.w+1) v) * ((λw.w+1) v) * 1) : □(int->int)
```

However, neither of the above implementations is quite satisfactory, since, evidently, the residual code in all the cases contains unnecessary redexes. The reason is that we do not utilize the *intensional* information that the passed argument is actually a boxed λ-abstraction, rather than a more general expression of a functional type. In a language with intensional code analysis, we can do a bit better. We can test the argument at run time and output a more optimized result if the argument is a λ-abstraction. This way we can obtain the most simplified, if not the most efficient residual code.

```
fun fexp (f : □(int->int)) (n : int) : □(int->int) =
    case f of
       box (λx:int. (w x:int[])) =>
             (* w : ∀q. □_q int -> □_q int *)
             let name X : int
                 box F = exp' [X] (w [X] (box X)) n
             in
                 box (λv:int. ⟨X->v⟩F)
             end
    else fexp2 f n

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

■

Example 20 This example is a (segment of the) meta function for symbolic differentiation with respect to a distinguished indeterminate X.

```
fun diff (e : □_X real) : □_X real =
    case e of
     box X => box 1

    | box ((w1:real[X]) + (w2:real[X])) =>
           let box e1 = diff (w1 [])
               box e2 = diff (w2 [])
           in
               box (e1 + e2)
           end
```

```
| box ((λx:real. ((FX x):real[X])) (GX:real[X])) =>
    (* FX : ∀q. □_q real -> □_{q,X} real *)
    (* GX : ∀q. □_{q,X} real *)
    (* check if FX really depends on X *)
    let name Y : real
    in
      case (FX [Y] (box Y)) of
       box (F:real[Y]) =>
       (* FX is independent of X; apply the chain rule *)
          let box f = F []
            box f' = diff (box ⟨Y->X⟩f)
            box gx = GX []
            box gx' = diff (GX [])
          in
            box (⟨X->gx⟩f' * gx')
          end
      else  diff (FX [X] (GX []))
    end
else (box 0) (* the argument is a constant *)
```

The most interesting part of diff is its treatment of application. The same limitations encountered in Example 18 apply here too, in the sense that we can pattern match only when the applying function is actually a λ-abstraction. Although it is wrong, we currently let all the other cases pass through the default case. Nevertheless, the example is still illustrative.

After splitting the application into the function part f and the argument part g we test if f is independent of X. If that indeed is the case, it means that our application was actually a composition of functions $f\ (g\ X)$, and thus we can apply the chain rule to compute the derivative as $f'\ (g\ X) * (g'\ X)$. Otherwise, if f contains occurrences of X, the chain rule is inapplicable, so we only reduce the β-redex and differentiate the result. ■

3.5 Logical relations for program equivalence

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

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

One of the questions that we explore in this section is an interplay between intensional and extensional equivalences of programs. The $\nu^\Box$-calculus is particularly appropriate for investigating and combining the two notions, because we can use the modal constructs as explicit boundaries between the different notions of equivalence. In particular, we can treat values of modal types as being *observable*, i.e. amenable to inspection of their structure. Then two general expressions of modal type will be extensionally equivalent if and only if their values are intensionally equivalent. We are also interested in exploring the properties of the calculus when only extensional equivalence is used, as the present formulation of $\nu^{\Box}$ does not contain any constructs for inspecting the structure of modal values. In both of these cases, we will establish that our formulation of $\nu^{\Box}$ is purely functional, in the sense that it satisfies the logical equivalences arising from the β-reductions and η-expansions of the language. The development presented here will follow the methodology of logical relations, as used, for example, in other works concerned with names in functional programming [PS93]. However, the details of our approach are different because we want to make the identity of locally declared names irrelevant for the purposes of expression comparison.

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

Example 21 In the examples below, we assume that X is a name of integer type.

1. $(\lambda x{:}\mathbf{int}.\ x+1)\ 2 \cong (\lambda x{:}\mathbf{int}.\ x+2)\ 1 \cong 3 : \mathbf{int}$, because all three terms evaluate to 3; however, neither of them is intensionally equivalent to any other.

2. $(\lambda x{:}\mathbf{int}.\ x+X)\ 2 \cong 2+X \cong X+2 : \mathbf{int}\,[X]$, because whenever X is substituted by e (and x is not free in e), the three terms evaluate to the same value.

3. $(\lambda x{:}\Box_X\mathbf{int}.\ 2)\ (\mathbf{box}\ X) \cong (1+1) : \mathbf{int}$, because both terms evaluate to 2. Notice that X does not appear in the second term, nor in the type and support of comparison.

4. $\mathbf{box}\ (X+1) \cong \mathbf{box}\ (X+1) : \Box_X\mathbf{int}$, because $X+1 \equiv_\alpha X+1 : \mathbf{int}\,[X]$ intensionally, as syntactic expressions.

■

As illustrated by this example, in our equivalence relations we should distinguish between two different kinds of names: (1) names which may appear in either of the compared terms, as well as their type and support (Example 21 cases 2 and 4), and (2) names which are local to some of the terms (Example 21 case 3). The later kind of names should not influence the equivalence relations – these names could freely be renamed.

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

$$e_1 \equiv_\alpha e_2,$$

and the judgment for extensional equivalence has the form

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

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

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

1. a type A is *simple* iff $A = b$, or $A = A_1 \to A_2$ or $A = A_1 \nrightarrow A_2$ where A_1, A_2 are simple types

2. a term e is *simple* if it does not contain the modal constructs box and $\mathsf{let\ box}$.

Then we only allow modal types $\Box_C A$ if A is simple, and modal terms **box** e if e is simple. We justify this restriction by a desire to avoid impredicativity arising in a language that can intensionally analyse the whole set of its expressions. In fact, it seems rather improbable that a language with such strong intensional capabilities can be designed at all. [3] Indeed, we added names and modal constructs in order to represent syntax with free variables. But, the modal constructs can also bind variables, so a new category of names and modalities seems to be required in order to analyze these new bindings, and then a new category of names and modalities is required for the bindings by the previous class of modalities, etc. Thus, here we limit the intensional equivalence to the simply-typed fragment, and leave the possible extensions to larger fragments for future work.

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

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

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

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

and the judgment for extensional equivalence of values has the form

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

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

$$\begin{array}{lll}
\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[\,] & \text{iff} & (\Sigma, \Sigma_1), e_1 \longmapsto^* (\Sigma, \Sigma_1'), v_1, \text{ and } (\Sigma, \Sigma_2), e_2 \longmapsto^* \\
& & (\Sigma, \Sigma_2'), v_2, \text{ and } \Sigma \vdash \Sigma_1'.\ v_1 \sim \Sigma_2'.\ v_2 : A \\
\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C] & \text{iff} & \Sigma \vdash \Sigma_1'.\ \{\sigma_1\}e_1 \cong \Sigma_2'.\ \{\sigma_2\}e_2 : A\,[\,] \text{ for any } \Sigma_i' \supseteq \Sigma_i, \\
& & \text{such that } \Sigma \vdash \Sigma_1'.\ \sigma_1 \cong \Sigma_2'.\ \sigma_2\,[C] \\
\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : b & \text{iff} & v_1 = v_2 \in \mathbb{N}
\end{array}$$

[3] here I'm talking about pattern matching, but if only equality is required, that may be possible

$\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A \to B$ iff $v_i = \lambda x{:}A.\ e_i$ and $\Sigma \vdash \Sigma'_1.\ [v'_1/x]e_1 \cong \Sigma'_2.\ [v_2/x]e_2 : B$, for any $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma \vdash \Sigma'_1.\ v'_1 \sim \Sigma'_2.\ v'_2 : A$

$\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : \Box_C A$ iff $v_i = \mathbf{box}\ e_i$ and $e_1 \equiv_\alpha e_2$ and $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$

$\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A \not\twoheadrightarrow B$ iff $v_i = \nu X{:}A.\ e_i$ and $\Sigma \vdash (\Sigma_1, X{:}A).\ e_1 \cong (\Sigma_2, X{:}A).\ e_2 : B\,[\,]$, where X is a fresh name.

Here we abbreviated:

$\Sigma \vdash \Sigma_1.\ \sigma_1 \cong \Sigma_2.\ \sigma_2\,[C]$ iff σ_1, σ_2 are explicit substitutions for the names in C, such that $\Sigma \vdash \Sigma_1.\ \sigma_1(X) \cong \Sigma_2.\ \sigma_2(X) : B\,[\,]$ for any name $X \in C$ such that $X{:}B \in \Sigma$.

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

Notice that the above definitions are well-founded. In order to establish this fact, let us define $\mathrm{ord}_\Sigma(X)$ to be the position in which the name X first appears in the name context Σ. Also, given a type A and support C, let

$$\max_\Sigma(A[C]) = \max\{\mathrm{ord}_\Sigma(X) \mid X \in \mathrm{fn}(A[C])\}.$$

Because of the restriction that each type in Σ may only refer to the names to the left of it, it is clear that if $X{:}A \in \Sigma$, then $\max_\Sigma(A) < \mathrm{ord}_\Sigma(X)$. We can now order the pairs of type A and support C as follows. The pair $A[C]$ is smaller than $B[D]$ iff

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

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

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

$\Sigma; \cdot; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[\,]$ iff $\Sigma \vdash \Sigma'_1.\ [\rho_1/\Gamma]e_1 \cong \Sigma'_2.\ [\rho_2/\Gamma]e_2 : A\,[\,]$ for any $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma \vdash \Sigma'_1.\ \rho_1 \sim \Sigma'_2.\ \rho_2 : \Gamma$

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

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

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

$\Sigma; \cdot; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$ iff $\Sigma; \cdot; \Gamma \vdash \Sigma'_1.\ \{\sigma_1\}e_1 \cong \Sigma'_2.\ \{\sigma_2\}e_2 : A\,[\,]$ for any $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma; \Gamma \vdash \Sigma'_1.\ \sigma_1 \cong \Sigma'_2.\ \sigma_2\,[C]$

where σ_1, σ_2 are explicit substitutions, and

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

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

$\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$ iff $\Sigma; \cdot; \Gamma \vdash \Sigma'_1.\ [\![\delta_1/\Delta]\!]e_1 \cong \Sigma'_2.\ [\![\delta_2/\Delta]\!]e_2 : A\,[C]$ for any $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma \vdash \Sigma'_1.\ \delta_1 \equiv_\alpha \Sigma'_2.\ \delta_2 : \Delta$

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

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

The above definitions are well-founded, as each one refers only to already introduced definitions. For the sake of completeness, we also parametrize the intensional relation $\equiv_\alpha$ with the context Δ, as this will be needed in the statement of Lemma 29.

$\Sigma; \Delta \vdash \Sigma_1.\ e_1 \equiv_\alpha \Sigma_2.\ e_2 : A\,[C]$ iff $[\![\delta_1/\Delta]\!]e_1 \equiv_\alpha [\![\delta_2/\Delta]\!]e_2$ for any $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma \vdash \Sigma'_1.\ \delta_1 \equiv_\alpha \Sigma'_2.\ \delta_2 : \Delta$

Example 22 Let $\Sigma = X{:}\mathbf{int}$. Then the following are valid instances of intensional equivalence.

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

■

Example 23 Consider the simple expression e such that $\Sigma; \Delta; \Gamma \vdash \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) : \Box\mathbf{int}$. In such a case, it is easy to see that $\Sigma; \Delta; \Gamma \vdash \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) \cong \mathbf{choose}\ (\nu X{:}B.\ \mathbf{box}\ e) : \Box\mathbf{int}$.

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

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

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

Lemma 24 (Name permutation)
Let $R_1 : \Sigma_1 \to \Sigma_1'$ and $R_2 : \Sigma_2 \to \Sigma_2'$ be bijections where Σ_1' and Σ_2' are well-formed in Σ. Then:

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

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

For the first induction hypothesis, we start by considering the base case when C is empty. In this case, if $(\Sigma, \Sigma_i), e_i \longmapsto^* (\Sigma, \Sigma_i, \Psi_i), v_i$, then by parametricity of the evaluation judgment, we also have $(\Sigma, \Sigma_i'), e_i \longmapsto^* (\Sigma, \Sigma_i', \Psi_i), R_i\ v_i$. Then we appeal to the second induction hypothesis, to derive that $\Sigma \vdash (\Sigma_1', \Psi_1).\ R_1\ v_1 \sim (\Sigma_2', \Psi_2).\ R_2\ v_2 : A$. The result is easily extended to the case when C is not empty.

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

Lemma 25 (Name localization)
If C is a well-formed support in Σ, then the following holds:

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

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

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

For the second induction hypothesis, we present the case when $A = A_1 \to A_2$, and $v_i = \lambda x{:}A_1.\ e_i$. In this case, consider $\Sigma_i' \supseteq \Sigma_i$, such that $\Sigma \vdash (\Sigma', \Sigma_1').\ v_1' \sim (\Sigma', \Sigma_2').\ v_2' : A_1$. We need to show $\Sigma \vdash (\Sigma', \Sigma_1').\ [v_1'/x]e_1 \cong (\Sigma', \Sigma_2').\ [v_2'/x]e_2 : A_2$. By induction hypothesis at type A_1, we have that $(\Sigma, \Sigma') \vdash \Sigma_1'.\ v_1' \sim \Sigma_2'.\ v_2' : A_1$, and therefore $(\Sigma, \Sigma') \vdash \Sigma_1'.\ [v_1'/x]e_1 \cong \Sigma_2'.\ [v_2'/x]e_2 : A_2$. By induction hypothesis at type A_2, we can push Σ' back inside to get $\Sigma \vdash (\Sigma', \Sigma_1').\ [v_1'/x]e_1 \cong (\Sigma', \Sigma_2').\ [v_2'/x]e_2 : A_2$. The opposite direction is symmetric. ■

Lemma 26 (Weakening)
Let $\Sigma' \supseteq \Sigma$, $\Sigma_1' \supseteq \Sigma_1$ and $\Sigma_2' \supseteq \Sigma_2$, so that Σ_1' and Σ_2' are well-formed with respect to Σ'. Then the following holds:

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

Proof: By name localization (Lemma 25), it suffices to consider $\Sigma' = \Sigma$. The proof is by simultaneous induction on the definition of the two judgments.

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

For the second induction hypothesis, the only interesting case is when $A = A' \to A''$, and $v_i = \lambda x{:}A'.\ e_i$. In this case, consider $\Sigma''_i \supseteq \Sigma'_i$, such that $\Sigma \vdash \Sigma''_1.\ v''_1 \sim \Sigma''_2.\ v''_2 : A'$. By definition, $\Sigma \vdash \Sigma''_1.\ [v''_1/x]e_1 \cong \Sigma''_2.\ [v''_2/x]e_2 : A''$, simply because $\Sigma''_i \supseteq \Sigma'_i \supseteq \Sigma_i$. ■

Lemma 27 (Symmetry and transitivity)

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

2. *If* $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, *then* $\Sigma \vdash \Sigma_2.\ v_2 \sim \Sigma_1.\ v_1 : A$.

3. *If* $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, *and* $\Sigma \vdash \Sigma_2.\ e_2 \cong \Sigma_3.\ e_3 : A\,[C]$, *then* $\Sigma \vdash \Sigma_1.\ e_1 \cong \Sigma_3.\ e_3 : A\,[C]$

4. *If* $\Sigma \vdash \Sigma_1.\ v_1 \sim \Sigma_2.\ v_2 : A$, *and* $\Sigma \vdash \Sigma_2.\ v_2 \sim \Sigma_3.\ v_3 : A$, *then* $\Sigma \vdash \Sigma_1.\ e_1 \sim \Sigma_3.\ v_3 : A$

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

By assumptions, $(\Sigma, \Sigma_1), e_1 \longmapsto (\Sigma, \Psi_1), v_1$, and $(\Sigma, \Sigma_2), e_2 \longmapsto (\Sigma, \Psi_2), v_2$, such that $\Sigma \vdash \Psi_1.\ v_1 \sim \Psi_2.\ v_2 : A$. Also, $(\Sigma, \Sigma_2), e_2 \longmapsto (\Sigma, \Psi'_2), v'_2$, and, $(\Sigma, \Sigma_3), e_3 \longmapsto (\Sigma, \Psi_3), v_3$, such that $\Sigma \vdash \Psi'_2.\ v'_2 \sim \Psi_3.\ v_3 : A$.

By determinacy of evaluation, we know that there is a permutation of names π such that $\Psi_2 = \pi(\Psi'_2)$ and $v_2 = \pi(v'_2)$, and thus by Lemma 24, $\Sigma \vdash \Psi_2.\ v_2 \sim \Psi_3.\ v_3 : A$. Then, by the last induction hypothesis, $\Sigma \vdash \Psi_1.\ v_1 \sim \Psi_3.\ v_3 : A$, and therefore, $\Sigma \vdash \Sigma_1.\ e_1 \sim \Sigma_3.\ e_3 : A$.

For the relation on values, we only present the case $A = A_1 \to A_2$ and $v_i = \lambda x{:}A_1.\ e_i$. In this case, let $\Sigma'_1 \supseteq \Sigma_1$ and $\Sigma'_3 \supseteq \Sigma_3$, such that $\Sigma \vdash \Sigma'_1.\ v'_1 \sim \Sigma'_3.\ v'_3 : A_1$. By name permutation, we can assume that Σ'_3 and Σ_2 are disjoint; otherwise, we can just rename the conflicting names in Σ_2. By symmetry and transitivity at type A_1, we obtain $\Sigma \vdash \Sigma'_3.\ v'_3 \sim \Sigma'_3.\ v'_3 : A_1$. By weakening, $\Sigma \vdash \Sigma'_1.\ v'_1 \sim \Sigma_2, \Sigma'_3.\ v'_3$ and $\Sigma \vdash \Sigma_2, \Sigma'_3.\ v'_3 \sim \Sigma'_3.\ v'_3$; therefore $\Sigma \vdash \Sigma'_1.\ [v'_1/x]e_1 \cong (\Sigma_2, \Sigma'_3).\ [v'_3/x]e_2 : A_2$ and $\Sigma \vdash (\Sigma_2, \Sigma'_3).\ [v'_3/x]e_2 \cong \Sigma'_3.\ [v'_3/x]e_3 : A_2$. Finally, by first induction hypothesis at type A_2, we get $\Sigma \vdash \Sigma'_1.\ [v'_1/x]e_1 \cong \Sigma'_3.\ [v'_3/x]e_3 : A_2$. ■

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

Lemma 28

1. *(Name permutation) Let* $R_1 : \Sigma_1 \to \Sigma'_1$ *and* $R_2 : \Sigma_2 \to \Sigma'_2$ *be bijections where* Σ'_1 *and* Σ'_2 *are well-formed in* Σ. *If* $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, *then* $\Sigma; \Delta; \Gamma \vdash \Sigma'_1.\ R_1\ e_1 \cong \Sigma'_2.\ R_2\ e_2 : A\,[C]$.

2. *(Name localization) Let* Δ, Γ, A, C *are well-formed in* Σ. *Then* $(\Sigma, \Sigma'); \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$ *if and only if* $\Sigma; \Delta; \Gamma \vdash (\Sigma', \Sigma_1).\ e_1 \cong (\Sigma', \Sigma_2).\ e_2 : A\,[C]$.

3. *(Weakening) Let $\Sigma' \supseteq \Sigma$, and $\Sigma_1' \supseteq \Sigma_1$, $\Sigma_2' \supseteq \Sigma_2$, $\Delta' \supseteq \Delta$, $\Gamma' \supseteq \Gamma$ and $C' \supseteq C$, so that Σ_1',Σ_2',Δ', Γ' and C' are well-formed with respect to Σ'. If $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, then $\Sigma'; \Delta'; \Gamma' \vdash \Sigma_1'.\ e_1 \cong \Sigma_2'.\ e_2 : A\,[C']$.*

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

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

Proof:

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

To complete the logical relations argument, we need to define the notion of extensional relation on the remaining syntactic category of $\nu^\Box$ – the category of explicit substitutions. This definition will be utilized in the statement and the proof of Lemma 29 to establish that term constructors of $\nu^\Box$ (in particular, the constructs for explicit substitutions and modal variables) preserve extensional equivalence.

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

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

and is defined by the following clauses:

$$\begin{array}{lll}
\Sigma; \cdot; \Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong & & \\
\qquad\qquad \Sigma_2.\ \langle\Theta_2\rangle : [C] \Rightarrow [D] & \text{iff} & \Sigma; \cdot; \Gamma \vdash \Sigma_1'.\ \{\Theta_1\}e_1 \cong \Sigma_2'.\ \{\Theta_2\}e_2 : A\,[D], \text{ for any} \\
& & \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma; \cdot; \Gamma \vdash \Sigma_1'.\ e_1 \cong \Sigma_2'.\ e_2 : A\,[C] \\
\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong & & \\
\qquad\qquad \Sigma_2.\ \langle\Theta_2\rangle : [C] \Rightarrow [D] & \text{iff} & \Sigma; \cdot; \Gamma \vdash \Sigma_1'.\ \langle[\delta_1/\Delta]\Theta_1\rangle \cong \Sigma_2'.\ \langle[\delta_2/\Delta]\Theta_2\rangle : [C] \Rightarrow [D] \\
& & \text{for any } \Sigma_i' \supseteq \Sigma_i, \text{ such that } \Sigma \vdash \Sigma_1'.\ \delta_1 \equiv_\alpha \Sigma_2'.\ \delta_2 : \Delta
\end{array}$$

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

Lemma 29

Logical relation is preserved by all the expression constructors of $\nu^\Box$. More precisely:

1. $(\Sigma, X{:}A); \Delta; \Gamma \vdash \Sigma_1.\ X \cong \Sigma_2.\ X : A\,[X, C]$

2. $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \Sigma_1.\ x \cong \Sigma_2.\ x : A\,[C]$

3. *if $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [D] \Rightarrow [C]$, then* $\Sigma; (\Delta, u{:}A[D]); \Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle u \cong \Sigma_2.\ \langle\Theta_2\rangle u : A\,[C]$

4. *if $\Sigma; \Delta; (\Gamma, x{:}A) \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : B\,[C]$, then $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ \lambda x{:}A.\ e_1 \cong \Sigma_2.\ \lambda x{:}A.\ e_2 : A \rightarrow B\,[C]$*

5. *if $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A \rightarrow B\,[C]$ and $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1' \cong \Sigma_2.\ e_2' : A\,[C]$, then $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1\ e_1' \cong \Sigma_2.\ e_2\ e_2' : B\,[C]$*

6. *If* $\Sigma;\Delta \vdash \Sigma_1.\ e_1 \equiv_\alpha \Sigma_2.\ e_2 : A\,[C]$, *and* $\Sigma;\Delta;\cdot \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, *then* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{box}\ e_1 \cong \Sigma_2.\ \mathbf{box}\ e_2 : \Box_C A\,[D]$

7. *if* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : \Box_D A\,[C]$ *and* $\Sigma;(\Delta, u{:}A[D]);\Gamma \vdash \Sigma_1.\ e'_1 \cong \Sigma_2.\ e'_2 : B\,[C]$, *then* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e'_1 \cong \Sigma_2.\ \mathbf{let\ box}\ u = e_2\ \mathbf{in}\ e'_2 : B\,[C]$

8. *if* $\Sigma;\Delta;\Gamma \vdash (\Sigma_1, X{:}A).\ e_1 \cong (\Sigma_2, X{:}A).\ e_2 : B\,[C]$, *then* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \nu X{:}A.\ e_1 \cong \Sigma_2.\ \nu X{:}A.\ e_2 : A \twoheadrightarrow B\,[C]$

9. *if* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A \twoheadrightarrow B\,[C]$ *then* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \mathbf{choose}\ e_1 \cong \Sigma_2.\ \mathbf{choose}\ e_2 : B\,[C]$

10. $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\ \rangle \cong \Sigma_2.\ \langle\ \rangle : [C] \Rightarrow [D]$ *if* $C \subseteq D$

11. *if* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[D]$, *and* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle\Theta_1\rangle \cong \Sigma_2.\ \langle\Theta_2\rangle : [C \setminus X] \Rightarrow [D]$, *and* $X{:}A \in \Sigma$, *then* $\Sigma;\Delta;\Gamma \vdash \Sigma_1.\ \langle X \rightarrow e_1, \Theta_1\rangle \cong \Sigma_2.\ \langle X \rightarrow e_2, \Theta_2\rangle : [C] \Rightarrow [D]$

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

In case of (3), consider $\Sigma'_i \supseteq \Sigma_i$ such that $e_1 \equiv_\alpha e_2$, and $\Sigma \vdash \Sigma'_1.\ e_1 \cong \Sigma'_2.\ e_2 : A\,[D]$. We need to show that $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \{[\![e_1/u]\!]\Theta_1\}e_1 \cong \Sigma'_2.\ \{[\![e_2/u]\!]\Theta_2\}e_2 : A\,[\,]$. From the assumption, we have $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \langle[\![e_1/u]\!]\Theta_1\rangle \cong \Sigma'_2.\ \langle[\![e_2/u]\!]\Theta_2\rangle : [D] \Rightarrow [\,]$, and then the required equality follows by definition of extensional equivalence for explicit substitutions

In case of (7), by equivalence of e_1 and e_2, there exist name sets Ψ_1, Ψ_2, such that $(\Sigma, \Sigma_1), e_1 \longmapsto^* (\Sigma, \Psi_1), \mathbf{box}\ t_1$ and $(\Sigma, \Sigma_2), e_2 \longmapsto^* (\Sigma, \Psi_2), \mathbf{box}\ t_2$, where $t_1 \equiv_\alpha t_2 : A\,[D]$, and $\Sigma \vdash \Psi_1.\ t_1 \cong \Psi_2.\ t_2 : A\,[D]$. Then it suffices to show that $\Sigma;\cdot;\cdot \vdash \Psi_1.\ [\![t_1/u]\!]e'_1 \cong \Psi_2.\ [\![t_2/u]\!]e'_2 : B\,[\,]$. But this follows from the second assumption, by definition of extensional equivalence.

In case of (11), again consider $\Sigma'_i \supseteq \Sigma_i$, such that $\Sigma';\cdot;\cdot \vdash \Sigma'_1.\ e'_1 \cong \Sigma'_2.\ e'_2 : B\,[C]$. To be consistent with the notation, in this case we assume that D, rather than C, is empty. To reduce clutter, denote by σ_1, σ_2 the explicit substitutions $\sigma_1 = \langle X \rightarrow e_1, \Theta_1\rangle$ and and $\sigma_2 = \langle X \rightarrow e_2, \Theta_2\rangle$. Then we need to show that $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \{\sigma_1\}e'_1 \cong \Sigma'_2.\ \{\sigma_2\}e'_2 : B\,[\,]$. To establish this, it suffices to prove that $\Sigma;\cdot \vdash \Sigma'_1.\ \sigma_1 \cong \Sigma'_2.\ \sigma_2\,[C]$, i.e. that $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \sigma_1(Z) \cong \Sigma'_2.\ \sigma_2(Z) : A'\,[\,]$ for any name $Z \in C$ such that $Z{:}A' \in \Sigma$. Then the result would follow from the extensional equivalence of e'_1 and e'_2. We consider two cases: $Z = X$, and $Z \in C \setminus X$. If $Z = X$, then $A' = A$ and $\sigma_i(Z) = e_i$ and by first assumption, $\Sigma;\cdot;\cdot \vdash \Sigma_1.\ \sigma_1(Z) \cong \Sigma_2.\ \sigma_2(Z) : A$. By weakening, this implies $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \sigma_1(Z) \cong \Sigma'_2.\ \sigma_2(Z) : A$. If $Z \in C \setminus X$, then $\sigma_i(Z) = \{\Theta_i\}Z$, and also obviously $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ Z \cong \Sigma'_2.\ Z : A'\,[C \setminus X]$. Then by the second assumption, $\Sigma;\cdot;\cdot \vdash \Sigma'_1.\ \sigma_1(Z) \cong \Sigma'_2.\ \sigma_2(Z) : A'\,[\,]$. The two cases combined demonstrate $\Sigma;\cdot \vdash \Sigma'_1.\ \sigma_1 \cong \Sigma'_2.\ \sigma_2\,[C]$, and this completes the proof. ■

Now we can prove that our logical relations are reflexive, and thus indeed equivalences.

Lemma 30 (Reflexivity)

1. *If* $\Sigma;\Delta;\Gamma \vdash e : A\,[C]$, *then* $\Sigma;\Delta;\Gamma \vdash e \cong e : A\,[C]$

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

Proof: By induction on the structure of e and Θ, using Lemma 29. ■

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

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

Lemma 31 (Fundamental property of logical relations)
If $\Sigma; \Delta; \Gamma \vdash \Sigma_1.\ e_1 \cong \Sigma_2.\ e_2 : A\,[C]$, *then*

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

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

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

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

Lemma 32
In the logical equivalences below we assume that all the judgments are well-formed and that the terms are well-typed in appropriate contexts.

1. $\Sigma; \Delta; \Gamma \vdash (\lambda x.\ e_1)\ e_2 \cong [e_2/x]e_1 : A\,[C]$
2. $\Sigma; \Delta; \Gamma \vdash e \cong \lambda x.\ (e\ x) : A \to B\,[C]$
3. $\Sigma; \Delta; \Gamma \vdash$ **let box** $u =$ **box** e_1 **in** $e_2 \cong [\![e_1/u]\!]e_2 : B\,[C]$
4. $\Sigma; \Delta; \Gamma \vdash e \cong$ **let box** $u = e$ **in box** $u : \Box_D B\,[C]$

5. $\Sigma; \Delta; \Gamma \vdash \textbf{choose}\ (\nu X{:}A.\ e) \cong (X{:}A).\ e : B\,[C]$

6. $\Sigma; \Delta; \Gamma \vdash (X{:}A).\ e \cong \nu X{:}A.\ \textbf{choose}\ e : A \not\twoheadrightarrow B\,[C]$

7. $\Sigma; \Delta; \Gamma \vdash \lambda z{:}A.\ \textbf{choose}\ (\nu X{:}A_1.\ e) \cong \textbf{choose}\ (\nu X{:}A_1.\ \lambda z{:}A.\ e) : A \to B\,[C]$

8. $\Sigma; \Delta; \Gamma \vdash \nu X.\ \nu Y.\ e \cong \nu Y.\ \nu X.\ e : A \not\twoheadrightarrow A \not\twoheadrightarrow B\,[C]$

9. $\Sigma; \Delta; \Gamma \vdash e_1\ (\textbf{choose}\ (\nu X{:}A.\ e_2)) \cong \textbf{choose}\ (\nu X{:}A.\ (e_1\ e_2)) : B\,[C]$

10. $\Sigma; \Delta; \Gamma \vdash (\textbf{choose}\ (\nu X{:}A.\ e_1))\ e_2 \cong \textbf{choose}\ (\nu X{:}A.\ (e_1\ e_2)) : B\,[C]$

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

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

To establish (1), let $\Sigma; \cdot; x{:}B \vdash e_1 : A$, and $\Sigma; \cdot; \cdot \vdash e_2 : B$. As the calculus is termination, there exist Ψ and v_2 such that $\Sigma, e_2 \longmapsto^* (\Sigma, \Psi), v_2$, and therefore also $\Sigma \vdash e_2 \cong \Psi.\ v_2 : B$. By the fundamental property of logical relations (Lemma 31), $\Sigma \vdash [e_2/x]e_1 \cong \Psi.\ [v_2/x]e_1 : A$. But it is also the case that $\Sigma \vdash (\lambda x.\ e_1)\ e_2 \cong \Psi.\ [v_2/x]e_1 : A$, simply because the two expressions evaluate to the same value. Then by transitivity, we get $\Sigma \vdash (\lambda x.\ e_1)\ e_2 \cong [e_2/x]e_1 : A$.

To establish (2), let $\Sigma, e \longmapsto^* (\Sigma, \Psi), (\lambda x.\ e')$, so that $\Sigma; \cdot; \cdot \vdash e \cong \Psi.\ (\lambda x.\ e') : A \to B$. By transitivity, this holds if $\Sigma \vdash \Psi.\ \lambda x.\ e' \sim \lambda x.\ (e\ x) : A \to B$. In order to prove this, consider Σ'_1, Σ'_2 such that $\Sigma \vdash \Psi, \Sigma'_1.\ v_1 \sim \Sigma'_2.\ v_2 : A$. It suffices to show $\Sigma \vdash (\Psi, \Sigma'_1).\ [v_1/x]e' \cong \Sigma'_2.\ (e\ v_2) : B$. By the name permutation property (Lemma 24), we can assume that Ψ and Σ_2 are disjoint. By the properties of evaluation, $(\Sigma', \Sigma'_2), (e\ v_2) \longmapsto^* (\Sigma', \Sigma'_2, \Psi), [v_2/x]e'$, and thus

$$\Sigma \vdash \Sigma'_2.\ (e\ v_2) \cong (\Psi, \Sigma'_2).\ [v_2/x]e' \qquad (*)$$

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

$$\Sigma \vdash (\Psi, \Sigma'_1).\ [v_1/x]e' \cong (\Psi, \Sigma'_2).\ [v_2/x]e' : B \qquad (**)$$

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

$$\Sigma \vdash (\Psi, \Sigma'_1).\ [v_1/x]e' \cong \Sigma'_2.\ (e\ v_2) : B.$$

To establish (6), let $(\Sigma, X{:}A), e \longmapsto (\Sigma, X{:}A, \Psi), (\nu Y{:}A.\ e')$. Then, by definition, we have $\Sigma \vdash (X{:}A).\ e \cong (X{:}A, \Psi).\ (\nu Y{:}A.\ e') : A \not\twoheadrightarrow B$. By transitivity, it suffices to show that $\Sigma \vdash (X{:}A, \Psi).\ \nu Y{:}A.\ e' \sim \nu X{:}A.\ \textbf{choose}\ e : A \not\twoheadrightarrow B$

By definition of the logical relation for values at the type $A \not\twoheadrightarrow B$, this holds if and only if $\Sigma \vdash (X{:}A, \Psi, Y{:}A).\ e' \cong X{:}A.\ \textbf{choose}\ e : B$. Indeed, we could chose $X{:}A$ in the local context of the second argument by the name permutation property. But the last equation is obviously true, as $(\Sigma, X{:}A), \textbf{choose}\ e \longmapsto^* (\Sigma, X{:}A, \Psi), \textbf{choose}\ (\nu Y{:}A.\ e') \longmapsto (\Sigma, X{:}A, \Psi, Y{:}A), e'$.

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

To establish (8), notice that by definition, the required equivalence holds if and only if $\Sigma \vdash (X{:}A, Y{:}A).\ e \cong (Y{:}A, X{:}A).\ e : B$. In this equation, we are justified in choosing the same names X and Y in both sides, by the name permutation property (Lemma 24). But the contexts $(X{:}A, Y{:}A)$ and $(Y{:}A, X{:}A)$ are same, because the type A does not depend on neither X nor Y. Thus, the result follows by reflexivity of $\cong$.

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

3.6 Notes

Related work on staged computation and run-time code generation

An early reference to staged computation is [Ers77] which introduces staged computation under the name of "generating extensions". Generating extensions for purposes of partial evaluation were also foreseen by [Fut71], and the concept is later explored and eventually expanded into multi-level generating extensions by [JSS85, GJ95, GJ97]. Most of this work is done in an untyped setting.

The typed calculus that provided the direct motivation and foundation for our system is the $\lambda^{\Box}$-calculus. It evolved as a type theoretic explanation of staged computation [DP01, WLPD98], and run-time code-generation [LL96, WLP98], and we described it in Section 3.1.

Related work on meta programming

Most of the work on functional meta programming today is related to the development of MetaML [TS97, MTBS99, Tah99, Tah00].

The core fragment of MetaML is based on the $\lambda^{\bigcirc}$-calculus. Formulated by [Dav96], $\lambda^{\bigcirc}$ is the proof-term calculus for discrete temporal logic, and it provides a notion of open object code where the free variables of the object expressions are represented by meta variables on a subsequent temporal level. The original motivation of $\lambda^{\bigcirc}$ was to develop a type system for binding-time analysis in the setup of partial evaluation, but it was quickly adopted for meta-programming through the development of MetaML.

MetaML builds upon the open code type constructor of $\lambda^{\bigcirc}$ and generalizes the language with several features. The most important one is the addition of a type refinement for closed code. Values classified by the closed code types are those open code expressions that do not contain any free meta variables. If an expression is typed as a closed code, then it may be evaluated at run time.

It might be of interest here to point out a certain similarity between our concept of supports and the dead-code annotations used in MetaML with references [CMT00, CMS01]. MetaML cannot naively allow references to open code, in order to avoid the extrusion of scope of bound variables. At the same time, limiting references to closed code types is too restrictive, as it rules out some programs that are well-typed in ML. Scope extrusion has to be allowed, but only if the extruding variables are never encountered during evaluation. As a solution, MetaML with references annotates terms with the list of free variables that the term is allowed to contain in dead-code positions.

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

Taha and Nielsen present another system for combining closed and open code in [NT03]. The system can explicitly name the object stages of computation through the notion of *environment classifiers*. Because the stages are explicitly named, each stage can be revisited multiple times and variables declared in previous visits can be reused. This feature provides the functionality of open code. The environment classifiers are related to our support variables in the sense that they both are bound by universal quantifiers and they both abstract over sets. Indeed, our support polymorphism explicitly abstracts over sets of names, while environment classifiers are used to name parts of the variable context, and thus implicitly abstract over sets of variables.

Related work on higher-order abstract syntax

Coming from the direction of higher-order abstract syntax, probably the first work pointing to the importance of a non-parametric binder like our ν-abstraction is [Mil90]. The connection of higher-order abstract syntax to modal logic has been recognized by Despeyroux, Pfenning and Schürmann in the system presented in [DPS97], which was later simplified into a two-level system in Schürmann's dissertation [Sch00]. The system presented in [Bjø99] is capable of pattern matching against object-level programs, but is not concerned with their evaluation. There is also [Hof99] which discusses various presheaf models for higher-order abstract syntax, then [FPT99] which explores untyped abstract syntax in a categorical setup, and an extension to arbitrary types [Fio02].

Related work on logic

The representation of syntactic expressions has been investigated in terms of modal logic of provability for quite some time. The connection between the two arises from Gödel's Incompleteness theorems, as for example described by Smorynski in [Smo85]. Montague's work [Mon63] is an early reference towards the impossibilty of a formal system that can reason about its own syntax and at the same time reflect the syntactically obtained results and treat them as true.

Chapter 4

Modal theory of effects

4.1 Propositional lax logic

4.1.1 Judgments and propositions

Lax logic [FM97] is a logic for reasoning about truth of propositions under certain constraints. Unlike in modal logic of partial judgments (Section 2), where the partiality conditions are explicitly specified by the support of the judgment and can be manipulated using the reflection principle, in lax logic the constraints are left abstract and unspecified.

Following closely Pfenning and Davies [PD01], we start the judgmental formulation of lax logic with the hypothetical judgments, one for the unconstrained truth and one for lax truth:

$$A_1\ \mathit{true}, \dots, A_n\ \mathit{true} \vdash A\ \mathit{true}$$

and

$$A_1\ \mathit{true}, \dots, A_n\ \mathit{true} \vdash A\ \mathit{lax}$$

In the development of lax logic, we use Δ, rather than Γ to vary over sets of true hypotheses. The reasons for this change of notation will become clear subsequently, when we present the embedding of propositional lax logic into the propositional modal logic. With this notational convention in mind, we write our two judgments as $\Delta \vdash A\ \mathit{true}$ and $\Delta \vdash A\ \mathit{lax}$.

Just as usual, the hypothetical truth is internalized using implication, except that in this case we denote the constructor as $\Rightarrow$, to differentiate the lax implication from the implication used in modal logic. Thus, we will have the following standard rules for implication

$$\frac{\Delta, A\ \mathit{true} \vdash B\ \mathit{true}}{\Delta \vdash A \Rightarrow B\ \mathit{true}} \qquad \frac{\Delta \vdash A \Rightarrow B\ \mathit{true} \qquad \Delta \vdash A\ \mathit{true}}{\Delta \vdash B\ \mathit{true}}$$

On the other hand, $A\ \mathit{lax}$ is supposed to hold if, intuitively, the proposition A is true under some, unspecified constraints. The following two statements formally capture this intuition and can be taken as definitional clauses for $A\ \mathit{lax}$.

Definition of lax truth

1. If $\Delta \vdash A\ \mathit{true}$ then $\Delta \vdash A\ \mathit{lax}$.

2. If $\Delta \vdash A\ \mathit{lax}$ and $\Delta, A\ \mathit{true} \vdash B\ \mathit{lax}$, then $\Delta \vdash B\ \mathit{lax}$.

The first clause states that if A is true, then A is certainly true under some constraint (namely, the trivial constraint that is always satisfied). In the second clause, if A is true under some constraint, then any consequence of the unconditional truth of A will itself be constrained by the original conditions imposed on A.

Internalizing lax truth into the unconstrained truth judgment proceeds along the familiar lines. We introduce a new unary connective $\bigcirc$ on propositions, with the formation rule

$$\frac{A\ \mathit{prop}}{\bigcirc\, A\ \mathit{prop}}$$

and with the introduction rule that relates the new connective to the lax judgment.

$$\frac{\Delta \vdash A\ \mathit{lax}}{\Delta \vdash \bigcirc A\ \mathit{true}}$$

As customary, here we assume that each proposition A appearing in the judgments is well-formed.

The elimination rule for $\bigcirc$ follows the second definitional principle above, but combines it with the introduction rule for $\bigcirc$.

$$\frac{\Delta \vdash \bigcirc A\ \mathit{true} \qquad \Delta, A\ \mathit{true} \vdash B\ \mathit{lax}}{\Delta \vdash B\ \mathit{lax}}$$

We also need a rule to realize the first definitional principle and provide a coercion from true to lax propositions.

$$\frac{\Delta \vdash A\ \mathit{true}}{\Delta \vdash A\ \mathit{lax}}$$

This axiomatization is locally sound and complete, as witnessed by local reduction and expansion. The local reduction is justified by the definitional property (2) above, from the premises $\Delta \vdash A\ \mathit{lax}$ and $\Delta, A\ \mathit{true} \vdash B\ \mathit{lax}$.

$$\frac{\dfrac{\Delta \vdash A\ \mathit{lax}}{\Delta \vdash \bigcirc A\ \mathit{true}} \quad \Delta, A\ \mathit{true} \vdash B\ \mathit{lax}}{\Delta \vdash B\ \mathit{lax}} \quad \Longrightarrow_R \quad \Delta \vdash B\ \mathit{lax}$$

$$\Delta \vdash \bigcirc A\ \mathit{true} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\Delta \vdash \bigcirc A\ \mathit{true} \quad \dfrac{\overline{\Delta, A\ \mathit{true} \vdash A\ \mathit{true}}}{\Delta, A\ \mathit{true} \vdash A\ \mathit{lax}}}{\Delta \vdash A\ \mathit{lax}}}{\Delta \vdash \bigcirc A\ \mathit{true}}$$

Example 24 The following are some judgments derivable in lax logic.

1. $\vdash A \Rightarrow \bigcirc A\ \mathit{true}$
2. $\vdash \bigcirc \bigcirc A \Rightarrow \bigcirc A\ \mathit{true}$
3. $\vdash (A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B\ \mathit{true}$

The derivations proceed as follows.

1. $\vdash A \Rightarrow \bigcirc A\ \mathit{true}$

$$\frac{\frac{\frac{\frac{}{A\ \mathit{true} \vdash A\ \mathit{true}}}{A\ \mathit{true} \vdash A\ \mathit{lax}}}{A\ \mathit{true} \vdash \bigcirc A\ \mathit{true}}}{\vdash A \Rightarrow \bigcirc A\ \mathit{true}}$$

2. $\vdash \bigcirc \bigcirc A \Rightarrow \bigcirc A\ \mathit{true}$

To reduce cluter, here we abbreviate by Δ the context consisting of a single hypothesis $\Delta = \bigcirc \bigcirc A\ \mathit{true}$.

$$\frac{\frac{\frac{\frac{}{\Delta \vdash \bigcirc \bigcirc A\ \mathit{true}} \quad \frac{\frac{}{\Delta, \bigcirc A\ \mathit{true} \vdash \bigcirc A\ \mathit{true}} \quad \frac{\frac{}{\Delta, A\ \mathit{true} \vdash A\ \mathit{true}}}{\Delta, A\ \mathit{true} \vdash A\ \mathit{lax}}}{\Delta, \bigcirc A\ \mathit{true} \vdash A\ \mathit{lax}}}{\Delta \vdash A\ \mathit{lax}}}{\Delta \vdash \bigcirc A\ \mathit{true}}}{\vdash \bigcirc \bigcirc A \Rightarrow \bigcirc A\ \mathit{true}}$$

3. $\vdash (A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B\ \mathit{true}$.

In this case, we introduce the following context abbreviations: $\Delta_1 = (A \Rightarrow B)\ \mathit{true}$, $\Delta_2 = \Delta_1, \bigcirc A\ \mathit{true}$, and $\Delta_3 = \Delta_2, A\ \mathit{true}$.

$$\frac{\frac{\frac{\frac{\frac{}{\Delta_2 \vdash \bigcirc A\ \mathit{true}} \quad \frac{\frac{\frac{}{\Delta_3 \vdash A \Rightarrow B\ \mathit{true}} \quad \frac{}{\Delta_3 \vdash A\ \mathit{true}}}{\Delta_3 \vdash B\ \mathit{true}}}{\Delta_3 \vdash B\ \mathit{lax}}}{\Delta_2 \vdash B\ \mathit{lax}}}{\Delta_2 \vdash \bigcirc B\ \mathit{true}}}{\Delta_1 \vdash \bigcirc A \Rightarrow \bigcirc B\ \mathit{true}}}{\vdash (A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B\ \mathit{true}}$$

■

Lax logic and modalities

From the logical standpoint, one can imagine that each possible world of modal logic represents a certain – abstract – constraint from the lax logic. Then the judgment A *lax* expresses that there *exists* a world (i.e. a constraint) in which A is true. Thus, the judgment for lax truth is semantically very similar to the judgment for possibility, because both represent a form of existential quantification.

Indeed, the two judgments share very similar typing rules and substitutions principles. In fact, upon inspection of the typing rules, there appears only one distinction: the judgment for lax truth has only one context of hypotheses Δ, while the judgment for modal possibility has two contexts

Δ and Γ, distinguishing between necessary and true hypotheses. Intuitive reasoning then leads to the following conclusion: if truth and necessity of modal logic are equated, that will have as a consequence the equating of lax truth with modal possibility, and respectively, $\bigcirc$ with $\Diamond$. Note that conflating truth and necessity does not conflate these two with possibility. If a proposition A is possible, then it is true at some accessible world (and hence necessary at that world). But it need not be true and necessary at the current world.

A precise statement of this observation involves embedding lax logic into modal logic. In particular, if A *true* and A *nec* are equated on the modal side, then the propositions A and $\Box A$ become logically equivalent. Henceforth, a lax proof depending on a hypothesis A *true*, will correspond to a modal proof that depends on $\Box A$ *true*. Similarly, a lax proof depending on A *lax*, will correspond to a modal proof that depends on $\Box A$ *poss*. Because the judgments for lax truth and for possibility are not used as hypotheses, the embedding has to manipulate the internalized forms of the two judgments. Thus a lax proof depending on $\bigcirc A$ *true* should correspond to a modal proof depending on $\Diamond\Box A$ *true*.

More formally, consider the translation $(-)^+$ of lax propositions into modal propositions, discovered by Pfenning and Davies in [PD01]:

$$\begin{array}{rcll}
(A \Rightarrow B)^+ & = & \Box A^+ \to B^+ & \\
(\bigcirc A)^+ & = & \Diamond\Box A^+ & \\
P^+ & = & P & \text{for atomic } P \\
 & & & \\
(\cdot)^+ & = & \cdot & \\
(\Delta, A\ \mathit{true})^+ & = & \Delta^+, A^+\ \mathit{nec} &
\end{array}$$

Then the following lemmas establishes the formal correspondence between the two logics.

Lemma 33

1. *If $\Delta \vdash A$ true then $\Delta^+; \cdot \vdash A^+$ true in modal logic.*

2. *If $\Delta \vdash A$ lax then $\Delta^+; \cdot \vdash \Box A^+$ poss.*

Proof: By simultaneous induction on the derivations of the first judgments [PD01]. ■

For the opposite direction, we need an inverse translation $(-)^-$, mapping modal propositions into lax propositions.

$$\begin{array}{rcll}
(A \to B)^- & = & A^- \Rightarrow B^- & \\
(\Box A)^- & = & A^- & \\
(\Diamond A)^- & = & \bigcirc A^- & \\
P^- & = & P & \text{for atomic } P \\
 & & & \\
(\Delta, A\ \mathit{nec})^- & = & \Delta^-, A^-\ \mathit{true} & \\
(\Gamma, A\ \mathit{true})^- & = & \Gamma^-, A^-\ \mathit{true} &
\end{array}$$

Notice that $(A^+)^- = A$.

Lemma 34

1. *If $\Delta; \Gamma \vdash A$ true in modal logic, then $(\Delta^-, \Gamma^-) \vdash A^-$ true in lax logic.*

2. *If* $\Delta; \Gamma \vdash A$ *poss, then* $(\Delta^-, \Gamma^-) \vdash A^-$ *lax.*

Proof: By simultaneous induction on the given derivation. ■

Theorem 35

1. $\Delta \vdash A$ *true in lax logic if and only if* $\Delta^+; \cdot \vdash A^+$ *true in modal logic.*

2. $\Delta \vdash A$ *lax if and only if* $\Delta^+; \cdot \vdash A^+$ *poss*

Proof: The left-to-right direction is Lemma 33. For the right-to-left direction of the first statement, if $\Delta^+; \cdot \vdash A^+$ *true* in modal logic, then by Lemma 34, $(\Delta^+)^- \vdash (A^+)^-$ *true*, and therefore $\Delta \vdash A$ *true* in lax logic. Similar reasoning proves the second statement as well. ■

From the axiomatic standpoint, the identification of truth and necessity in constructive S4 modal logic can be accomplished by addition of the single axiom scheme (or inference rule)

$$\frac{}{A \to \Box A \ \mathit{true}}$$

Indeed, because constructive S4 already proves $\Box A \to A$ *true*, adjoining $A \to \Box A$ *true* annihilates the logical distinction between A and $\Box A$, and correspondingly, between truth and necessity. Notice that if A and $\Box A$ are equivalent in modal logic, then instead of the translation $(-)^+$ we could use the translation $(-)^*$ (defined below), as A^+ and A^* are equivalent for any A.

$$\begin{array}{rcll}
(A \Rightarrow B)^* & = & A^* \to B^* & \\
(\bigcirc A)^* & = & \Diamond A^* & \\
P^* & = & P & \text{for atomic } P \\
 & & & \\
(\cdot)^* & = & \cdot & \\
(\Delta, A\ \mathit{true})^* & = & (\Delta^*, A^*\ \mathit{nec}) &
\end{array}$$

Moreover, the equivalence between A^+ and A^* leads to the following theorem.

Theorem 36

1. *If* $\Delta \vdash A$ *true in lax logic, then* $\Delta^*; \cdot \vdash A^*$ *true in modal logic with* $A \to \Box A$.

2. *If* $\Delta \vdash A$ *lax in lax logic, then* $\Delta^*; \cdot \vdash A^*$ *poss in modal logic with* $A \to \Box A$.

3. *If* $\Delta; \Gamma \vdash A$ *true in modal logic with* $A \to \Box A$*, then* $(\Delta^-, \Gamma^-) \vdash A^-$ *true in lax logic.*

4. *If* $\Delta; \Gamma \vdash A$ *poss in modal logic with* $A \to \Box A$*, then* $(\Delta^-, \Gamma^-) \vdash A^-$ *lax in lax logic.*

Proof: The first two statements trivially follow from Lemma 33 by the equivalence of the translations $(-)^+$ and $(-)^*$. For the third statement, assume that $\Delta; \Gamma \vdash A$ *true* in modal logic extended with $B \to \Box B$. Then by Lemma 34, $(\Delta^-, \Gamma^-) \vdash A^-$ *true* in lax logic extended with $(B \to \Box B)^-$. But, $(B \to \Box B)^-$ is equal to $B^- \Rightarrow B^-$, which is already derivable in lax logic. Thus, $(\Delta^-, \Gamma^-) \vdash A^-$ *true* in lax logic with no additions. The proof of the fourth statement is similar. ■

As a consequence, $\Delta \vdash A$ *true* and $\Delta \vdash A$ *lax* are derivable in lax logic if and only if $\Delta^*; \cdot \vdash A^*$ *true* and $\Delta^*; \cdot \vdash A^*$ *poss*, are derivable in modal logic with $A \to \Box A$, respectively. Notice, however, that the translation $(-)^*$ simply *renames* the lax connectives into modal connectives. In other words, the intuituionistic lax logic is obtained when the constructive modal S4 is extended with the axiom scheme $A \rightarrow \Box A$. In that case, modal possibility attains the properties of lax truth, and correspondingly, the operator $\Diamond$ becomes $\bigcirc$.

The described embedding also explains why lax logic has only one modal constructor, corresponding to $\Diamond$, and lacks a constructor corresponding to $\Box$.

4.1.2 Lax λ-calculus

In this section, we decorate the judgments of lax logic with proof terms. The obtained proof term system, called lax λ-calculus, extends the ordinary λ-calculus with new syntactic categories to account for the specifics of lax logic. Again, we follow Pfenning and Davies [PD01] in the presentation. The judgments $\Delta \vdash A\ \mathit{true}$ and $\Delta \vdash A\ \mathit{lax}$ are now changed into $\Delta \vdash e : A$ and $\Delta \vdash f{\div}A$, where e and f are proof terms witnessing the judgments. The syntax of the calculus is summarized below.

$$\begin{array}{lrcl}
\textit{Types} & A, B & ::= & P \mid A \Rightarrow B \mid \bigcirc A \\
\textit{Expressions} & e & ::= & x \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{val}\ f \\
\textit{Phrases} & f & ::= & e \mid \mathbf{let\ val}\ x = e\ \mathbf{in}\ f \\
\textit{Variable contexts} & \Delta & ::= & \cdot \mid \Delta, x{:}A
\end{array}$$

As can be noticed, the syntactic categories of expressions and phrases are slightly different from the categories of expressions and phrases used in the modal λ- and ν-calculi. We retain the same terminology, however, in order emphasize the relationship between the modal and lax calculi.

As customary in the transition from logic to λ-calculus, the the context Δ now contains propositions labeled with variables, so that instead of $A\ \mathit{true}$ we write $x{:}A$. We present the type system below.

$$\frac{}{\Delta, x{:}A \vdash x : A}$$

$$\frac{\Delta, x{:}A \vdash e : B}{\Delta \vdash \lambda x{:}A.\ e : A \Rightarrow B} \qquad \frac{\Delta \vdash e_1 : A \Rightarrow B \qquad \Delta \vdash e_2 : A}{\Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Delta \vdash e : A}{\Delta \vdash e{\div}A}$$

$$\frac{\Delta \vdash f{\div}A}{\Delta \vdash \mathbf{val}\ f : \bigcirc A} \qquad \frac{\Delta \vdash e : \bigcirc A \qquad \Delta, x{:}A \vdash f{\div}B}{\Delta \vdash \mathbf{let\ val}\ x = e\ \mathbf{in}\ f{\div}B}$$

As can be seen, the proof terms constructors and the typing rules for unconstrained truth define a fragment of the system that corresponds to the ordinary λ-calculus. On the other hand, the constructors and the rules for lax truth are similar to the rules for the possibility fragment of the modal λ-calculus from Section 1.2.

Example 25 The following are well-typed terms in the lax λ-calculus.

1. $\vdash \lambda x.\, \mathbf{val}\; x : A \Rightarrow \bigcirc A$
2. $\vdash \lambda x.\, \mathbf{val}\; (\mathbf{let\ val}\; y = x\; \mathbf{in}\; \mathbf{let\ val}\; z = y\; \mathbf{in}\; z) : \bigcirc \bigcirc A \Rightarrow \bigcirc A$
3. $\vdash \lambda f.\, \lambda x.\, \mathbf{val}\; (\mathbf{let\ val}\; y = x\; \mathbf{in}\; f\; y) : (A \Rightarrow B) \Rightarrow \bigcirc A \Rightarrow \bigcirc B$

■

We now restate the definitional properties for the lax modalities using the newly introduced proof terms of the lax λ-calculus.

1. If $\Delta \vdash e : A$, then $\Delta \vdash e \rightsquigarrow A$.
2. If $\Delta \vdash f_1 \rightsquigarrow A$ and $\Delta, x{:}A \vdash f_2 \rightsquigarrow B$, then $\Delta \vdash \langle\!\langle f_1/x \rangle\!\rangle f_2 \rightsquigarrow B$.

The definitional property (1) simply expresses that each expression can be coerced into a phrase. The property (2) is a substitution principle for phrases. It uses a similar form of phrase substitution $\langle\!\langle f'/x \rangle\!\rangle f$ as the one defined in the case of modal possibility (Section 1.2).

$$\begin{array}{rcl} \langle\!\langle e/x \rangle\!\rangle f & = & [e/x]f \\ \langle\!\langle \mathbf{let\ val}\; y = e\; \mathbf{in}\; f'/x \rangle\!\rangle f & = & \mathbf{let\ val}\; y = e\; \mathbf{in}\; \langle\!\langle f'/x \rangle\!\rangle f \end{array}$$

The local reductions and expansions of the calculus are

$$\begin{array}{rcl} (\lambda x{:}A.\, e_1)\; e_2 & \Longrightarrow_R & [e_2/x]e_1 \\ e : A \Rightarrow B & \Longrightarrow_E & \lambda x{:}A.\, e\; x \end{array}$$

$$\begin{array}{rcl} \mathbf{let\ val}\; x = \mathbf{val}\; f_1\; \mathbf{in}\; f_2 & \Longrightarrow_R & \langle\!\langle f_1/x \rangle\!\rangle f_2 \\ e : \bigcirc A & \Longrightarrow_E & \mathbf{val}\; (\mathbf{let\ val}\; x = e\; \mathbf{in}\; x) \end{array}$$

4.1.3 Values and computations

In this section we review the main results on a monadic treatment of effects. The idea, originally proposed by Moggi [Mog89, Mog91] for structuring denotational semantics, and then adopted by Wadler [Wad92, Wad95, Wad98] for functional programming, is to use a unary type constructor $\bigcirc$ (called *monad*), to distinguish in the type system between values and effectful computations. We deliberately use the notation $\bigcirc$ from lax logic, to emphasize the connection between the lax λ-calculus and effectful computations. We will make this connection more explicit subsequently.

For example, if A is a type of values, then $\bigcirc A$ classifies computations of type A. The reason for this distinction is that computations do not need to pure. In the course of its evaluation, a computation is not limited to only compute a value – in fact, it is not even required to – it may be evaluated in order to perform an effect. For example, a computation may update the global store, raise an exception, perform I/O, or perhaps diverge. As argued by many works on type-and-effect systems ([GL86, LG88, Mog91, Wad92, Wad95, Wad98, JG91, TJ94, TT97] among others), and explored in the context of the programming language Haskell [Pey03], it may be beneficial for the programming practice to make explicit in the type system that a certain program expression may perform an effect. Such a type system restricts the class of environments that an expression may interact with and makes the reasoning about effectful programs much more modular, and hence simpler. This in turn facilitates the compile-time discovery of programming errors related to effects, and enables more agressive optimization.

The exact effects that a computation may perform may vary. However, independently of the nature of particular effects, there are two generic operations applicable to any notion of computation:

1. Every value e can be coerced into an effectful computation that trivially returns that value.
2. Two effectful computations f_1 and f_2 can be composed as follows: first f_1 is evaluated, and its value (if it exists) is supplied as an input to f_2. The result is a computation "inheriting" the effects of both f_1 and f_2.

It is no accident that the description of these two generic operations relates so closely to the definitional principles of lax logic and the lax λ-calculus from the previous section. In fact, the lax λ-calculus perfectly embodies the described distinction between values and computations, as witnessed by the following interpretation of its syntactic categories.

1. An expression $e : A$ describes a pure computations, which evaluates with no side effects, and therefore produces a value of type A. From the operational standpoint, an expression e is observationally equivalent to its value.
2. The phrase $f \rightsquigarrow A$ describes an effectful computation of type A. Two effectful computations can be combined, as described by the phrase substitution principle from the previous section.
3. An effectful computation $f \rightsquigarrow A$ can be internalized as an expression **val** $f : \bigcirc A$.
4. An expression $e : A$ (or more precisely, its value), can be coerced into an effectful computation $e \rightsquigarrow A$ and then internalized into an expression **val** $e : \bigcirc A$.

In the original papers on monadic treatment of effects [Mog89, Mog91], Moggi has proposed a *monadic meta language* as a general framework for describing operations on effectful computations. The monadic meta language is very similar to the lax λ-calculus, but it does not make a judgmental separation between pure and effectful computations. Rather, it conflates the notions of expressions and phrases, and contains only one judgment $\Delta \vdash e : A$, with the following typing rules.

$$\frac{}{\Delta, x{:}A \vdash x : A}$$

$$\frac{\Delta, x{:}A \vdash e : B}{\Delta \vdash A \Rightarrow B} \qquad \frac{\Delta \vdash e_1 : A \Rightarrow B \qquad \Delta \vdash e_2 : A}{\Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Delta \vdash e : A}{\Delta \vdash \mathbf{comp}\ e : \bigcirc A} \qquad \frac{\Delta \vdash e_1 : \bigcirc A \qquad \Delta, x{:}A \vdash e_2 : \bigcirc B}{\Delta \vdash \mathbf{let\ comp}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc B}$$

In fact, Moggi's formulation of the monadic meta language uses proof terms **val** and **let val**, which we rename here into **comp** and **let comp**, to avoid confusion with the constructors of the lax

λ-calculus.

The local reductions and expansions of the monadic meta language are given as follows.

$$\begin{array}{rcl} (\lambda x{:}A.\ e_1)\ e_2 & \Longrightarrow_R & [e_2/x]e_1 \\ e : A \Rightarrow B & \Longrightarrow_E & \lambda x{:}A.\ e\ x \end{array}$$

$$\begin{array}{rcl} \textbf{let comp}\ x = \textbf{comp}\ e_1\ \textbf{in}\ e_2 & \Longrightarrow_R & [e_1/x]f_2 \\ e : \bigcirc A & \Longrightarrow_E & \textbf{let comp}\ x = e\ \textbf{in comp}\ x \end{array}$$

These reductions and expansions, however, are not sufficient to explain all the interactions between effectful expressions. Because of the unusual elimination rule for $\bigcirc$, expressions of monadic type may be introduced using both **comp** and **let comp** forms, but the local reduction for $\bigcirc$ only accounts for the first possibility. Thus, the monadic meta language requires an additional equational rule to treat the *commuting conversions* between nested **let comp** expressions.

$$\begin{array}{l} \textbf{let comp}\ x = (\textbf{let comp}\ y = e_1\ \textbf{in}\ e_2)\ \textbf{in}\ e \quad \Longrightarrow \\ \qquad\qquad\qquad\qquad \textbf{let comp}\ y = e_1\ \textbf{in}\ (\textbf{let comp}\ x = e_2\ \textbf{in}\ e) \end{array}$$

Example 26 In the monadic meta language, the particular notions of effects are usually specified by a notational definition of the type $\bigcirc A$ and its corresponding expressions, in terms of already available language constructs.

For example, if we want a language capable of raising an exception of type E, we use disjoint sums to define the *exception monad* $\bigcirc$ and its corresponding monadic term constructors [Mog91, Wad95].

$$\begin{array}{rcl} \bigcirc A & = & A + E \\ \textbf{comp}\ e & = & \textbf{inl}\ e \\ \textbf{let comp}\ x = e_1\ \textbf{in}\ e_2 & = & \textbf{case}\ \ e_1\ \textbf{of inl}\ x \Rightarrow e_2 \mid \textbf{inr}\ y \Rightarrow \textbf{inr}\ y \end{array}$$

There are also additional term constructors used to raise and handle the exception associated with the monad $\bigcirc$.

$$\begin{array}{rcl} \textbf{raise} & : & E \Rightarrow \bigcirc A \\ \textbf{raise}\ e & = & \textbf{inr}\ e \\ \textbf{handle} & : & \bigcirc A \Rightarrow (E \Rightarrow A) \Rightarrow A \\ \textbf{handle}\ e\ h & = & \textbf{case}\ \ e\ \textbf{of inl}\ v \Rightarrow v \mid \textbf{inr}\ exn \Rightarrow h\ exn \end{array}$$

The constructor **raise** takes an expression $e : E$ and coerces it into **inr** e. This way, it implements exception raising, passing the value of e along. The constructor **handle** takes an expression $e : \bigcirc A$ and a function h representing an exception handler. If e evaluates to a value $v : A$, the result of handling is v. If e raises the exception with a value $exn : E$, then the result of handling is $h\ exn$.

The operational semantics follows the standard operational semantics associated with disjoint sums. For example, let us assume that $\bigcirc A = A + E$ is an exception monad, and that $f : \textbf{int} \Rightarrow \bigcirc\textbf{int}$. The following program adds the results of $f\ 1$ and $f\ 2$. If the evaluation of any of the two function applications raises an exception, the overall computed result is zero.

```
handle (let comp x1 = f 1
            comp x2 = f 2
        in
            comp (x1 + x2)
        end) (λexn. 0)
```

■

Example 27 In this example, we present the monad of side effects. The monad of side effects defines computations that execute in a state. The computation can read from the state, and modify it. Let S be a set of possible states. A stateful computation of type A is a computation that may read from the current state, before returning a value of type A, and a new state. Hence, stateful computations are classified by the the monad defined as follows.

$$\begin{array}{rcl} \bigcirc A & = & S \Rightarrow (A \times S) \\ \textbf{comp}\ e & = & \lambda s{:}S.\ \langle e, s\rangle \\ \textbf{let comp}\ x = e_1\ \textbf{in}\ e_2 & = & \lambda s{:}S.\ \textbf{let}\ \langle x, s'\rangle\ =\ (e_1\ s)\ \textbf{in}\ (e_2\ s') \end{array}$$

The type $\bigcirc A = S \Rightarrow (A \times S)$ expresses the fact that a stateful computation is a function: it reads from a state before returning a value and a new state. The constructor **comp** coerces a value e into a trivial stateful computation that returns e and the unchanged state. The constructor **let comp** evaluates e_1 in the current state, before passing the obtained value x and the new state s' to e_2.

The type S and the notion of state associated with this monad may be defined in many different ways, depending on the wanted side effects. For example, S may represent memory store in which mutable references may be allocated, read from and written into [LP95, BHM02]. For simplicity, in this example we assume that the state consists of a single integer location which can be read and written. Correspondingly, we set $S = \textbf{int}$, and adjoin the following specific constructors to the state monad $\bigcirc$.

$$\begin{array}{rcl} \textbf{read} & : & \bigcirc\textbf{int} \\ \textbf{read} & = & \lambda s{:}\textbf{int}.\ \langle s, s\rangle \\ \textbf{write} & : & \textbf{int} \Rightarrow \bigcirc\textbf{unit} \\ \textbf{write}\ e & = & \lambda s{:}\textbf{int}.\ \langle (), e\rangle \\ \textbf{init} & : & \textbf{int} \Rightarrow \bigcirc A \Rightarrow A \\ \textbf{init}\ e_1\ e_2 & = & \textbf{fst}\ (e_2\ e_1) \end{array}$$

The stateful computation **read** returns the value of the integer location from the state s; s remains unchanged. The computation **write** e changes s so that the value of e is now stored into it. This computation is not evaluated for its value, so that it returns the trivial value ():**unit**. The constructor **init** initializes the state location with the value of e_1, then executes the stateful computation e_2 and returns the computed value.

As an example of the constructors for stateful computations, consider the program below. In this program, we assume a function $add : \textbf{int} \Rightarrow \bigcirc\textbf{int}$ which adds its argument to the value of the state location, while returning the old state value as a result.

```
init 1 (let comp x = read
            comp y = add (x)
            comp dummy = write (y + 1)
        in
            read
        end)
```

The program first initializes the state with 1, and then increments it by means of the function *add*. The value bound to y is 1, which is the old value of the state. Then $y + 1 = 2$ is re-written into the state, and it is this value that is finally computed by the program. ■

As established by Pfenning and Davies in [PD01] and Benton, Bierman, de Paiva in [BBdP98] and Kobayashi [Kob97], both the lax λ-calculus and the monadic meta language are computationally adequate. However, because the lax λ-calculus does not require any special treatment for commuting conversions, it has a bit simpler and more pleasant proof-theoretic properties.

4.2 Modalities for effectful computation

As summarized and illustrated in the previous section, monads and lax logic can be used to differentiate in the type system between values and effectful computations. Having in mind that the monadic meta language and the lax λ-calculus very closely correspond to modal possibility, a natural question arises: does a *dual* development to modal possibility and monads have any computational import to the treatment of effects? In other words, can we employ modal necessity to capture some invariants of effectful computations, and if so, which invariants does modal necessity represent?

We start our analysis of this question by making a distinction similar to the one made in the monadic meta language and the lax λ-calculus in Section 4.1.3. We assume that the non-modal type A corresponds to *values*, and that the modal types $\Box A$ and $\Diamond A$ stand for some kind of *computations* of type A. But, what kind of computations exactly do the two different modalities represent?

Let us first consider modal possibility, because it is related to lax logic and monads from Section 4.1, and these have been extensively studied in the literature. We recall the relevant typing rules and the substitution principle, in a version decorated with the calculus of proof terms (Section 1.1.4).

$$\frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash e \div A}$$

$$\frac{\Delta; \Gamma \vdash f \div A}{\Delta; \Gamma \vdash \textbf{dia}\ f : \Diamond A} \qquad \frac{\Delta; \Gamma \vdash e : \Diamond A \qquad \Delta; x{:}A \vdash f \div B}{\Delta; \Gamma \vdash \textbf{let dia}\ x = e\ \textbf{in}\ f \div B}$$

Substitution principle for possibility

If $\Delta; \Gamma \vdash f_1 \div A$ and $\Delta; x{:}A \vdash f_2 \div B$, then $\Delta; \Gamma \vdash \langle\!\langle f_1/x \rangle\!\rangle f_2 \div B$.

In the substitution principle for possibility, the operation of phrase substitution $\langle\!\langle f'/x \rangle\!\rangle f$ is defined as

$$\begin{array}{rcl} \langle\!\langle e/x \rangle\!\rangle f & = & [e/x]f \\ \langle\!\langle \mathbf{let\ dia}\ y = e\ \mathbf{in}\ f'/x \rangle\!\rangle f & = & \mathbf{let\ dia}\ y = e\ \mathbf{in}\ \langle\!\langle f'/x \rangle\!\rangle f \end{array}$$

The important observation about modal possibility is that it enforces a programming style by which the computations (and therefore, the corresponding effects) are *serialized*, i.e. totally ordered. Indeed, each phrase witnessing a possibility judgment is a nested list of **let dia** clauses. Thus, for any two computations of types $\Diamond A$ and $\Diamond B$ respectively, it is always evident from the program which of the two takes precedence. For example, let $e_1 : \Diamond A$ and $e_2 : \Diamond B$ be two computations, and consider the phrase

$$F = \mathbf{let\ dia}\ x_1 = e_1\ \mathbf{in}\ (\mathbf{let\ dia}\ x_2 = e_2\ \mathbf{in}\ f)$$

It is clear from the form of F that e_1 takes precedence over e_2, and that any sound operational semantics for phrases will have to evaluate e_1 first, before attempting e_2. Moreover, the definition of modal possibility prohibits writing phrases in which this ordering is not immediately evident. In particular, let $F_1 \div A \rightarrow B$ and $F_2 \div A$ be two phrases defined as follows:

$$F_1 = \mathbf{let\ dia}\ x_1 = e_1\ \mathbf{in}\ f_1 \quad \text{and} \quad F_2 = \mathbf{let\ dia}\ x_2 = e_2\ \mathbf{in}\ f_2$$

Then it is impossible to put F_1 and F_2 together into an application like $(F_1\ F_2)$ where it is unclear which of two phrases – and which of the two computations e_1 and e_2 – comes first. Indeed, $F_1\ F_2$ is not a well-formed element of the category of phrases, as defined in Section 1.1.4.

The operation of phrase substitution $\langle\!\langle f/x \rangle\!\rangle f'$ combines the substituted phrases by giving precedence to the effects of f over the effects of f'. As an illustration, let F' be another phrase with its own computational effects, and consider the phrase substitution $\langle\!\langle F/x \rangle\!\rangle F'$, where F is defined above.

$$\langle\!\langle F/x \rangle\!\rangle F' \quad = \quad (\mathbf{let\ dia}\ x_1 = e_1\ \mathbf{in\ let\ dia}\ x_2 = e_2\ \mathbf{in}\ \langle\!\langle f/x \rangle\!\rangle F')$$

Notice that the effectful computations e_1 and e_2 are the first two computations in the result of the substitution, and therefore take precedence over the computations of F'. As a conclusion, any operational semantics based on the substitution principle for possibility will respect the serialization specified by the phrase constructors and appropriately order the computational effects of the program.

It is this property, shared by both monads and modal possibility, that makes them very appropriate for representing *persistent* effectful computations where an effect may *change the environment* in which the program executes. A change inflicted upon the environment may influence the subsequent computations. Therefore, in order to have a well-defined semantics, it is important that the program effects are always performed in a strictly specified order. A typical example of the persistent kind of effects is writing into a memory location. And indeed, as it is well-known from many practical algorithmic and systems applications, writing into memory locations must typically be serialized, so that the value stored in the location is always well-defined.

Of course, another way to specify the ordering of program effects is to define it by the operational semantics. This strategy is adopted by many programming languages, a typical example being Standard ML [MTHM97]. But, a type system – like that associated with monads or modal possibility – that makes it explicit which expressions are effectful and which are not, has a certain advantage. It not only specifies the ordering of effects, but it provides the compiler with the knowledge of effectful properties of program expressions. This knowledge can be utilized to perform better optimizations. For example, if an expression is effectful, then it should be evaluated in the serialized order given by the program. But if an expression is pure, then its subterms may freely be rearranged, optimized, and evaluated out of order.

Let us now inspect the possible use of modal necessity for representation of effects. We recall the relevant typing rules and the substitution principle for necessity, in its version decorated with proof terms, as presented in Section 1.1.3.

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

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

Substitution principle for necessity
If $\Delta; \Gamma \vdash e_1 : A$ and $(\Delta, u{::}A); \Gamma \vdash e_2 : B$, then $\Delta; \Gamma \vdash [e_1/u]e_2 : B$.

Unlike modal possibility, notice that modal necessity does not prescribe any particular ordering among effects. To contrast this with our previous discussion of possibility, let $e_1 : \Box A$ and $e_2 : \Box B$ be two computations, and consider the expressions $E_1 : A \to B$ and $E_2 : A$, defined as follows:

$$E_1 = \mathbf{let\ box}\ u_1 = e_1\ \mathbf{in}\ e_1' \quad \text{and} \quad E_2 = \mathbf{let\ box}\ u_2 = e_2\ \mathbf{in}\ e_2'$$

Then it is perfectly well-defined to put together E_1 and E_2 into an application like $(E_1\ E_2) : B$. Observe that the language constructs used in this expression do not specify which of the expressions E_1 and E_2 – and therefore which of the computations e_1 and e_2 – takes precedence over the other. It must be left to the operational semantics of the language to determine the evaluation order between the two, but any strategy is sound. Furthermore, unlike the phrase substitution principle, the substitution principle for necessity relies on ordinary substitution $[e_1/u]e_2$ — it freely propagates and even duplicates effectful computations, without any concern for the ordering of the effects involved.

As a consequence, if modal necessity is to represent effectful computations, these could only be computations that do not change the run-time environment of the program. The computations may *depend on the environment*, but they should not change it — they are *benign*. Examples of benign effects abound: non-termination, memory reads and control-flow effects like exceptions, to mention but a few.

The simple modal type system in itself, however, is not strong enough to represent benign effects. In many cases of benign effects, results of benign computations depend on the evaluation environment. It is of paramount importance, therefore, to prevent evaluating effectful expressions within environments that cannot deal with the effect in question. For example, an expression that reads from a memory location X should only be evaluated when a memory location X is actually allocated and initialized. An expression raising the exception X should only be evaluated when a handler for X is active. Thus, it is necessary for soundness purposes that the type of a benign computation captures the relevant aspects of the environment on which the computation depends on.

This is where names and supports, as developed in Section 2.2, become important. Henceforth, rather than using a simple modal type system, we will consider a modal type system with names and indexed modalities. For example, if a computation of type A needs to read from the memory location X, or may raise the exception X, we will ascribe it the type $\Box_X A$. Names and supports provide yet further possibilities. Using indexed necessity types, we can encode in the type system the notion of *handling*, i.e. restoring the purity of an impure computation by means of some action. Handling will be related to the principle of reflection from Section 2.1. When the effect X in a computation of type $\Box_X A$ is *handled*, we obtain a *pure* computation of type $\Box A$, and then a value of type A.

A following logical analogy can be made about modal types for effects. A computation of type A with a benign effect identified by the name X is, in a sense, a partial computation. In order to produce a value of type A, it needs to be evaluated in an environment capable of dealing with X. But it can be successfully evaluated in *all* such environments — hence we can ascribe it the the bounded universal type $\Box_X A$. On the other hand, a persistent computation of type A that changes the aspect of the run-time environment associated with the name X (for example, writes into the memory location X), will be ascribed the bounded existential type $\Diamond_X A$. Indeed, such a computation is a witness that there *exists* an environment – the one obtained after changing X – in which a value of type A can be computed.

To summarize, we can use the modal type system with names to distinguish between following computational categories: (1) *values*, which are associated with non-modal types A, (2) *computations with benign effects*, which are associated with necessitation types $\Box_C A$, and (3) *persistent computations*, which are associated with possibility types $\Diamond_C A$. In a modal type system with names, we can also make a characterization of *pure computations*. A pure computation of type A is a computation with no effects. In particular, it does not depend on any aspects of the run-time environment, and can therefore be ascribed a type $\Box A$, where the index support on the modal operator is empty. A pure computation is not necessarily a *value* itself, but it may be evaluated to produce a value. This property is logically characterized by the axiom $\Box A \to A$ of constructive S4 modal logic.

Just as in the case of the monadic meta language, we will also want to coerce values into computations. But in the modal system, we can actually express that a computation obtained by coercing a value is, in fact, pure. An appropriate logical analog of this coercion is the proposition

$$A \to \Box A$$

As already discussed in Section 4.1, adjoining this proposition to CS4 modal logic results in two things: (1) modal possibility becomes lax truth, and correspondingly, $\Diamond$ becomes a strong monad in the sense of Moggi [Mog91], and (2) the logical distinction between A and $\Box A$ is annihilated. In lax logic, this resulted in removing the operator $\Box$ from considerations. If this axiom is adjoined to modal logic with names, it again makes the types A and $\Box A$ logically equivalent. However, this does not remove the need for the operator $\Box$ and its associated proof terms. In modal logic with names, there is a whole family of necessitation operators $\Box_C$, indexed by supports C. Identifying A and $\Box A$ certainly does not collapse this whole indexed family. The operator $\Box$ can still make distinctions between propositions. For example, one proposition that does *not* become derivable after equating A and $\Box A$ is the implication $\Box_X A \to A$. The computational content of this proposition states that every computation with a benign effect X evaluates to a value. But this is obviously false. For example, a computation of type A that may raise the exception X, certainly need not evaluate to a value. Indeed, it may actually raise the exception.

Before we proceed with the technical details of a modal type system for effectful computations, we need to answer the following important question: do benign computations indeed present a separate category and require their own type constructor? Is it possible to perhaps treat benign computations using monads or modal possibility, or to simply ignore their effects and consider them pure?

Of course, every benign computation may be considered as trivially persistent, and represented using the same mechanism of monads or modal possibility. But that representation would fail to capture the important invariant that benign computations do not change the run-time environment, and therefore do not need to be serialized. Indeed, why serialize two computations that both read from a memory location X, when they could easily be evaluated out of order.

On the other hand, perhaps benign computations may be considered pure? After all, this is exactly how non-termination is often treated in practice. Because diverging expressions do not change the run-time environment (in fact, they do not even depend on the environment), non-termination in most cases is not even considered an effect. Unlike non-termination, however, not all benign effects are independent of the run-time environment in which they are evaluated. For example, a computation that reads from the memory location X will produce a different result, depending on the content of X at the time of evaluation. Such a computation may therefore be optimized, rearranged, memoized, evaluated out of order, or in parallel with many other computations reading from X, but only as long as the content of X is unchanged. In particular, this evaluation cannot be postponed beyond the first subsequent write into X. This is very different from pure computations which can be postponed indefinitely, and only evaluated when their result is needed.

As a conclusion then, it is sensible to employ a modal type system to distinguish between values, pure computations, computations with benign effects, and computations with persistent effects. We proceed in the following section with a description of the technical details of such a type system.

4.3 A modal type system for benign effects

The main judgment of the modal type system for benign effects is a variant of the partial truth judgment for modal logic from Sections 2.1 and 2.2:

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

We recall here the relevant syntactic conventions. For example, the typing ascriptions in the context Δ are of the form $u{:}A\,[C]$, assigning the type A and support C to the variable u. The name context Σ consists of type assignments $X_1{:}A_1, \dots, X_n{:}A_n$, associating names $X_1, \dots, X_n$ with types $A_1, \dots, A_n$, respectively. All the names used in the typing judgment are required to be declared and typed in Σ. It is assumed that all the names $X_1, \dots, X_n$ are distinct, and the set $\{X_1, \dots, X_n\}$ is denoted by $\mathrm{dom}(\Sigma)$. The context Σ is dependently typed, because each type A_i may depend on names. Thus, each X_i may be used only to the right of its declaration in Σ.

In the modal system for benign effects, names stand for the particular notion of effects, and this notion may differ from application to application. For example, if we want to design a type system that tracks location reads in order to prevent reading from uninitialized locations, we will use names to declare memory locations. If we want to design a type system that tracks raising and handling of exceptions, we will use names to declare individual exceptions.

In the modal system for benign effects, the support C associated with the expression e lists the effects that may be enacted during the evaluation of e. For example, if the expression e may *read* from a location $X{:}A$, then the name X will be in the support of e. If the expression e may *raise* the exception $X{:}A$, then the name X will be in the support of e. Support C will typically be a finite set of names, but we will also consider an application in Section 4.8, where C is a finite *list* of names. What is important, however, is that supports come equipped with a partial ordering

$$C \sqsubseteq D$$

whose minimal element is the empty support (be it a set or a list). This is analogous to the development of partial judgments in Chapter 2. The idea behind the partial ordering of supports is the following: if the expression e has support C, then all the effects that may arise during the evaluation of e are listed in C. But then, trivially, all these effects are listed in $D \sqsupseteq C$, and thus e could be ascribed a support D as well. Thus, one of the important structural properties of the type system is the *support weakening* principle phrased as follows.

Principle (Support weakening for expressions)
If $\Sigma; \Delta \vdash e : A\,[C]$ and $C \sqsubseteq D$, then $\Sigma; \Delta \vdash e : A\,[D]$.

By declaring which effects may be enacted by the expression e, the support C also determines in which run-time environments the expression e may be evaluated. For example, if e may read from the location X, then e must be evaluated in an environment in which X is initialized. Or, if e may raise an exception X, then e must be evaluated in an environment with an active handler for X. Thus, our type system will have a judgment for typing *environments*, in order to determine when an environment Θ matches a support C. The general form of the judgment for environments[1] is:

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

An expression e of support C may only appear in a context of an environment Θ that is typed as $[C] \Rightarrow [D]$ (for some D). Thus, the typing $\langle\Theta\rangle : [C] \Rightarrow [D]$ declares that Θ can appropriately deal with the effects C. We will keep the environment judgment undefined for a moment, and provide definitions for each particular notion of effect that we consider in the subsequent sections. Obviously, the environment judgment corresponds to the support judgment C *holds* $[D]$ from Section 2.1 and the judgment of explicit substitutions $\langle\Theta\rangle : [C] \Rightarrow [D]$ from Section 2.2. The environments are subject to the similar support weakening principles as explicit substitutions and C *holds* $[D]$.

Principle (Support weakening for environments)
If $\Sigma; \Delta \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$ and $D \sqsubseteq D'$, then $\Sigma; \Delta \vdash \langle\Theta\rangle : [C] \Rightarrow [D']$.

The relationship between expressions and environments is established in the type system via the following rule corresponding to the rule for *reflection* in Section 2.1.

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

This rule ensures that an expression e is always evaluated in a context of an environment Θ that can deal with the effects of e. In this sense, the type system of benign effects may be seen as a particular version of modal logic of partial truth from Section 2, in which the process of reflection is defined as evaluation.

There is one notable distinction, however, between benign effects and partial truth. As the reader may have already noticed, *none of the judgments for benign effects uses the context* Γ*, which*

[1] Although, in specific cases we will deviate slightly from this form in order to provide more information relevant to the environments.

is pervasive in modal logic of partial truth. There is a reason for this omission. When expressions are treated as effectful computations, then values naturally must be considered as pure, i.e. effect free. Indeed, values can never enact any effects, simply because their evaluation is already finished. Because a pure computation returning a value of type A is itself typed as $\Box A$, treating values like pure computations logically corresponds to extending the modal type system with the axiom

$$A \to \Box A$$

This move is identical to the way lax logic and the lax λ-calculus are obtained from modal logic and the modal λ-calculus (Section 4.1.2), where we used the above axiom to identify truth and necessity. It is only that in the system for benign effects, we start with a modal logic for partial judgments (Chapter 2), rather than the propositional modal logic (Chapter 1). But if truth and necessity are identified, then the context of truth hypotheses Γ is subsumed by the context of necessity hypotheses Δ, as part of Δ that declares variables of empty support. Correspondingly, in our notation we will use x, y and variants to range over variables with *empty* support, and we write $x{:}A$, instead of $x{:}A\,[\,]$, when a variable x with empty support is declared in Δ.

We immediately put this this notational convention to use in our formulation of the typing rules for function types $A \to B$.

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

The typing rules follow the customary formulations for λ-abstraction and application, but there are several important observations to be made about the support C in these rules. First of all, notice that the abstraction $\lambda x{:}A.\ e$ requires the body e to be typed with *empty support.* The motivation for this typing is purely computational. In the usual formulation of operational semantics for functional programming languages, λ-abstractions are always considered to be *values.* Because we want to identify values and pure computations, we must require that function bodies be *pure.* The whole λ-abstraction itself may be ascribed an arbitrary support C, which is a formulation required by the support weakening principle.

Example 28 Anticipating section 4.6, suppose that our language contains a constructor **raise**, such that $\mathbf{raise}_X\ e$ raises an exception X, passing the value of e along (assuming that both X and e have the same type). Expressions that may potentially raise the exception X, will be ascribed a support X by the type system. That way, the type system keeps track of the effects that an expression may cause. Assuming that X is an exception of integer type, the following expression F is not well-typed.

$$F = \lambda y{:}\mathbf{int}.\ 1 + \mathbf{raise}_X\ y$$

The body $1 + \mathbf{raise}_X\ y$ of F is effectful and has support X. But then F itself cannot be typed, because of the restriction on the rules for λ-abstraction, as explained above.

Notice that the restriction on the typing of F is necessary. Even if F is a value, and does not immediately perform an effect, it still cannot be considered pure. Indeed, F has the *potential* to perform an effect, once it is applied to an argument. If F is typed as pure, the type system will not be able to account for the effect of F. This is not to say that function bodies in our calculus cannot contain effectful terms. They can, but the effects have to be *encapsulated* by the constructs for modal necessity. For example, the term F' below is a well-typed counterpart to F.

$$F' = \lambda y{:}\mathbf{int}.\ \mathbf{box}\ (1 + \mathbf{raise}_X\ y) : \mathbf{int} \to \Box_X \mathbf{int}$$

The typing of F' will be explained in detail in the forthcoming developments. ■

A further observation about the typing rules for functions concerns the seeming mismatch between the support of the argument e_2 in the application rule, and the support with which the variables are introduced in the context Δ in the λ-abstraction rule. Indeed, λ-bound variables are declared in Δ with empty support, but e_2 may have an arbitrary support C. This mismatch is resolved by requiring that e_2 must always be *evaluated under the current environment* before its value is passed to e_1. Because the value of e_2 is pure (just like any value), it matches the empty support used to declare bound variables in Δ. As a consequence, the calculi that we design in this section will inherently be *call-by-value*. To make our operational semantics concrete, we will also impose a left-to-right evaluation strategy. Notice however, that we deal with benign effects, and therefore the evaluations of the function and the evaluation of function arguments do not interfere with each other. The type system may in fact be soundly ascribed right-to-left or any other call-by-value evaluation order.

From the logical standpoint, the described mismatch in supports is justified by the observation that our type system identifies truth and necessity, in the same ways it is done in the formulation of lax logic (Section 4.1). Because of this identification, *all* of our expressions are actually *categorical*, and are therefore subject to *reflection*. We are free to reflect the argument e_2 before substituting into e_1. As already discussed, in the type system for benign effects reflection corresponds to evaluation, so we simply rely on the operational semantics to specify that e_2 should be reflected before we pass it to e_1.

The notion of computation with benign effects is internalized into the calculus by using the modal type constructor for necessity $\Box$. For example, given a type A, the type $\Box_C A$ will classify the computations of type A, whose evaluation may cause the benign effects determined by the support C. The appropriate typing rules are obtained by erasing the context Γ from the standard formulation of the typing rules for $\Box$ (Section 2.2.1).

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

We also have the following hypothesis rule

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

The term $\mathbf{box}\ e : \Box_C A$ is a *value* that *encapsulates* an effectful computation e. As already explained, when e is evaluated, it may enact the effects whose names are listed in C. Because $\mathbf{box}\ e$ is a value, and therefore pure, it may be weakened to an arbitrary support D. From the operational standpoint, boxing an expression e *suspends* its evaluation. On the other hand, performing $\mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ e'$ binds e to u, but does not necessarily evaluate e itself. The expression e will be evaluated only if u appears in e' outside of boxed expressions.

It is interesting here to draw a parallel between the operational behavior of modal constructors with the behavior of λ-abstraction in impure functional languages. Suspending an effectful expression e in an impure functional language is usually achieved by creating a λ-abstraction $\lambda x.\ e$ (where $x \notin \mathrm{fv}(e)$). For example, in a typical type-and-effect system [GL86, LG88, JG91, TJ94], a computation is represented as a λ-abstractions whose type is annotated with a list of effects. The characteristic typing rules are usually a variation on the following.

$$\frac{\Sigma; (\Delta, x{:}A) \vdash e : B\,[C]}{\Sigma; \Delta \vdash \lambda x{:}A.\ e : A \xrightarrow{C} B\,[\ ]}\,(*) \qquad \frac{\Sigma; \Delta \vdash e_1 : A \xrightarrow{C} B\,[D_1] \qquad \Sigma; \Delta \vdash e_2 : A\,[D_2]}{\Sigma; \Delta \vdash e_1\ e_2 : B\,[C, D_1, D_2]}\,(**)$$

Does this similarity indicate that modal constructs are perhaps superfluous and may be removed in favor of functional abstraction?

The answer to the above question is negative, as the import of the modal constructors in the language of effects is not solely operational. Their main role is *not* to suspend the evaluation of expressions, but to *internalize* the notion of effectful computation. For example, note that the rules (*) and (**) are not locally complete, and therefore are not logically justified. The local expansion of $e : A \overset{C}{\to} B\,[D]$ is given as

$$e : A \overset{C}{\to} B\,[D] \qquad \Longrightarrow_E \qquad \lambda x.\, e\; x : A \overset{C,D}{\to} B$$

and the expression e has a *different* type and support from its expansion. To contrast this, local expansion in the calculus of benign effects preserves types and supports, as can easily be checked from the equation below.

$$e : \Box_C A\,[D] \qquad \Longrightarrow_E \qquad \textbf{let box}\ u = e\ \textbf{in box}\ u$$

In fact, when effectful computations are internalized as a separate semantic category which is different from functions, then functions and function types are fred from the responsibility to track effects. Moreover, in such situations functions are usually required to be *pure*. This is not only the case in our calculus of benign effects, but is also true of the monadic meta language [Mog91, Wad92]. In both our calculus and the monadic meta language, a function body may contain an effect only if the effect is encapsulated by a computation-forming construct. And in both languages, the range type of such a function will be a computation type (monadic type $\bigcirc A$ in the monadic meta language, and a modal type $\Box_C A$ in the calculus of benign effects).

Finally, our type system needs constructs for introduction of fresh effect instances into the computation. Again, we adopt the approach from the modal calculus of Section 2.2 with certain modifications.

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

The term constructor $\nu X{:}A.\, e$ is the introduction form for the new type $A \twoheadrightarrow B$. It declares a fresh effect instance under the name X and introduces X into the context of names Σ. Any unused name $X \notin \mathrm{dom}(\Sigma)$ would produce the same result, as justified by the renaming principle below. As a consequence, the form $\nu X{:}A.\, e$ actually *binds* the name X, which can therefore be α-renamed into any other unused name of type A. The elimination form **choose** e *allocates* a new effect instance of an appropriate type, and uses it instead of the name bound by e. The abstraction $\nu X{:}A.\, e$ is a value in our calculus, just like all the other type introduction forms that we introduced so far. For the same reason as in the case of λ-abstraction, we require that the body of ν-abstraction has empty support, in order to preserve the purity of values.

Principle (Renaming)
If $(\Sigma, X{:}A, \Sigma_1); \Delta \vdash e : B\,[C]$ and $Y{:}A$ is a fresh name, i.e. Y does not appear anywhere in this judgments, then

$$(\Sigma, Y{:}A, [Y/X]\Sigma_1); [Y/X]\Delta \vdash [Y/X]e : ([Y/X]B)\,[[Y/X]C].$$

To summarize, the calculus of benign effects is very similar to the fragment of the ν-calculus from Section 2 containing the $\Box$ operator, with several important distinctions. First of all, the calculus of benign effects admits the axiom $A \to \Box A$, which is not realized in the ν-calculus. The operational import of this axiom is to coerce values into pure computations. As a consequence, the context Γ of value variables, which is characteristic of the judgmental formulations of modal logic and modal calculi, is subsumed by the context Δ in the calculus of benign effects. Second, bodies of λ- and ν-abstractions in the calculus of benign effects must have empty support, while in the ν-calculus this support may be arbitrary. Third, and probably the most important is that reflection in the ν-calculus is performed *eagerly*, upon modal substitution, and is defined on expressions that may contain free modal variables. In the calculus of benign effects, reflection of the expression e under the environment Θ is specified by a separate term constructor $\langle\Theta\rangle e$. It is not tied to modal variables and modal substitution.

Before we conclude this section, we summarize the syntax, typing and operational semantics of the modal calculus for benign effects. Just as in Section 2.1, this will not be a complete system, but rather only the common core fragment that we extend in future section with constructs defining particular effects. In each of these cases we will provide the appropriate proofs of progress and type preservation.

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

The type system consists of the following three judgments: $\Sigma \vdash C\ \textit{supp}$, $\Sigma \vdash A\ \textit{type}$ and $\Sigma; \Delta \vdash e : A\,[C]$. The first two judgments deal with formation of supports and types. The third judgment $\Sigma; \Delta \vdash e : A\,[C]$ checks the expression e against the type A and support C. In this judgment, it is assumed that A and C are well-formed.

Definition of $\Sigma \vdash C\ \textit{supp}$.

$$\frac{}{\Sigma \vdash \cdot\ \textit{supp}} \qquad \frac{\Sigma \vdash C\ \textit{supp} \qquad X \in \mathrm{dom}(\Sigma)}{\Sigma \vdash C, X\ \textit{supp}}$$

Definition of $\Sigma \vdash A\ \textit{type}$.

$$\frac{}{\Sigma \vdash P\ \textit{type}} \qquad \frac{\Sigma \vdash A\ \textit{type} \qquad \Sigma \vdash B\ \textit{type}}{\Sigma \vdash A \to B\ \textit{type}} \qquad \frac{\Sigma \vdash A\ \textit{type} \qquad \Sigma \vdash B\ \textit{type}}{\Sigma \vdash A \nrightarrow B\ \textit{type}}$$

$$\frac{\Sigma \vdash A\ \textit{type} \qquad \Sigma \vdash C\ \textit{supp}}{\Sigma \vdash \Box_C A\ \textit{type}}$$

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

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

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

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

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

Example 29 If C, C_1, C_2 and D are well-formed supports, then the following are derivable typing judgments in the calculus of benign effects.

1. $\vdash \lambda x.\ \mathbf{box}\ x : A \to \Box_D A$
2. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u : \Box A \to A\,[-]$
3. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ u : \Box_{C_1} A \to \Box_C A,$ where $C_1 \sqsubseteq C$
4. $\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box\ box}\ u : \Box_{C_1} A \to \Box_D \Box_C A,$ where $C_1 \sqsubseteq C$
5. $\vdash \lambda x.\ \lambda y.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ let\ box}\ v = y\ \mathbf{in\ box}\ u\ v$
   $: \Box_{C_1}(A \to B) \to \Box_{C_2} A \to \Box_C B,$ where $C_1, C_2 \sqsubseteq C$

Notice that the judgment (2) requires that the type of the abstraction argument is $\Box A$, where the index on the modal operator is empty. Indeed, the following generalization of (2) to non-empty supports is not derivable in the calculus of bening effects, because of the previously discussed restriction that bodies of λ-abstractions must be pure.

$$\not\vdash \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u : \Box_{C_1} A \to A\,[C]$$

However, the hypothetical judgment corresponding to this implication is derivable, as shown below.

$$x{:}\Box_{C_1} A \vdash \mathbf{let\ box}\ u = x\ \mathbf{in}\ u : A\,[C], \qquad \text{where } C_1 \sqsubseteq C$$

■

Example 30 To abbreviate notation and reduce clutter, we introduce into the calculus the term constructor **unbox** e as a syntactic abbreviation for **let box** $u = e$ **in** u. The new term constructor has the following derived typing rule

$$\frac{\Sigma; \Delta \vdash e : \Box_C A\,[D] \qquad C \sqsubseteq D}{\Sigma; \Delta \vdash \mathbf{unbox}\ e : A\,[D]}$$

We also define **let val** $x = e_1$ **in** e_2 to stand for **unbox** $((\lambda x.\ \mathbf{box}\ e_2)\ e_1)$, rather than the usual $(\lambda x.\ e_2)\ e_1$. The additional complication arises because we have to box e_2 and make it pure before we can put it under a λ-abstraction. The derived typing rule for **let val** is

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

Similarly, the term constructor **let name** $X{:}A$ **in** e is an abbreviation for

$$\mathbf{unbox}\ (\mathbf{choose}\ (\nu X{:}A.\ \mathbf{box}\ e)),$$

with the typing rule below. It is assumed that X is a fresh name which does not appear in dom(Σ).

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

■

The operational semantics of this core fragment of the modal calculus of benign effects is defined through the judgment

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

which relates an expression e with its one-step reduct e'. The expressions e and e' must not contain any free variables. However, both e and e' may contain effects, whose names are declared in Σ and Σ', respectively. The name context Σ' is always an extension of Σ, as the reduction step may introduce new names to stand for new effect instances.

The operational semantics is a call-by-value, left-to-right, evaluation context semantics in the style of Wright and Felleisen [WF94]. In order to perform one evaluation step, the expression e is decomposed uniquely as $e = E[r]$, where r is a redex, and E is an evaluation context, capturing the environment in which r is reduced. Then it suffices 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.

$$\begin{array}{llcl}
\textit{Values} & v & ::= & \lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e \mid \ldots \\
\textit{Redexes} & r & ::= & (\lambda x.\ e)\ v \mid \mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ e \mid \mathbf{choose}\ (\nu X.\ e) \\
\textit{Evaluation contexts} & E & ::= & [\,] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E
\end{array}$$

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

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

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

4.4 Dynamic binding

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 in a given environment will be an effect, and substituting a name with a term by means of an *explicit substitution* will *handle* this effect. The operational semantics evaluates expressions with empty support, and hence permits dereferencing of only those names that are captured by some explicit substitution. 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 values to names via explicit substitutions is not a destructive operation. Each name can be assigned a value an arbitrary number of times (including zero), but the assignment only have local scope, and dereferencing a name will use the nearest assignment. 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 4.5 to obtain a more general calculus of state with destructive update. The previous work related to dynamic binding is discussed in at the end of this chapter in Section 4.9.

The syntax of the calculus for dynamic binding extends the core fragment with new constructs for name lookup and substitution. The modal constructor $\Box$ is used to internalize effectful computations. An expression of type $\Box_C A$ is a computation that produces a value of type A when executed, but in the course of evaluation may need to dereference the names listed in the support C. In the case of dynamic binding, supports are sets of names, and the partial ordering on supports is defined as the subset ordering on sets. In other words, $C \sqsubseteq D$ if and only if $C \subseteq D$. Obviously, the empty set is the minimal element of this ordering. The resulting language is very similar to the ν-calculus from Section 2.2. However, dynamic binding is an example of a calculus of benign effects, and it inherits the distinctive features of the core calculus for benign effects (summarized in Section 4.3).

In dynamic binding, the environment in which expressions are evaluated is a *store*, consisisting of a set of names (i.e., memory locations) each of which is associated with a value. We represent stores using explicit substitutions. An explicit substitution Θ is syntactically defined as a set of assignments of expressions to names. A name X is referenced by simply using it in some term. The construct $\langle\Theta\rangle e$ applies Θ over the expression e, or alternatively, evaluates e in the store represented by Θ.

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

Example 31 Let us assume that X and Y are integer names. The code segment below defines a benign computation u that reads from X and Y to return $X^2 + Y^2$. Then X and Y are initialized to 1 and 2, respectivelly, before $u + 2XY$ is evaluated.

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

  val it = 9 :  int
```

■

The semantics of explicit substitutions is defined as in Section 2.2.3, subject to some minor modification. We repeat the definition here in a more compact form, and point out the differences from the previous sections.

Explicit substitutions are partial functions from names to terms. In other words, an explicit substitution never assigns an expression to a name more than once, and there is no ordering between the substitution assignments. Given a substitution Θ, the domain and range of Θ are the sets

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

and

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

The set $\mathrm{fn}(\Theta)$ of free variables of Θ is defined as the set of free variables of expressions in $\mathrm{range}(\Theta)$. The set $\mathrm{fn}(\Theta)$ of free names of Θ is the set of names in the domain and range of Θ. We denote the empty substitution simply by $\langle\ \rangle$.

Every substitution Θ defines a unique function of *substitution application* $\{\Theta\}$ on expressions. Substitution application $\{\Theta\}e$ is capture-avoiding and is defined by induction of the structure of e as follows.

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

As usual, substitution application does not descend under **box**. Names appearing in a internalized computations need not be initialized because an internalized computation is suspended, and hence its names are not dereferenced. However, when a computation 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 31. This aspect of explicit substitutions emphasizes and illustrates our observation from Section 4.3 that modal constructors do not simply serve to suspend computations. As the above definition shows, the construct **box** e, in addition to suspending the evaluation of e, also "protects" the expression e from the surrounding explicit substitutions.

To outline some further aspects of the above definition, 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, substitution application does not descend into λ- and ν-abstractions, because the type system guarantees that abstraction bodies are pure, and therefore name-free.

The operation of substitution application depends upon the operation of *substitution composition* $\Theta_1 \circ \Theta_2$, which is defined as in Section 2.2.3.

$$\Theta_1 \circ \Theta_2 = \{X \to \{\Theta_1\}(\llbracket\Theta_2\rrbracket(X)) \mid X \in \mathrm{dom}(\Theta_1) \cup \mathrm{dom}(\Theta_2)\}$$

The operation is well-founded – computing $\Theta_1 \circ \Theta_2$ only requires applying Θ_1 to subterms in range(Θ_2). On the other hand, substitution application is defined inductively, so the size of terms on which it operates is always decreasing.

The type system for dynamic binding extends the core system for benign effects with rules that describe the specific aspects of name dereference and substitution. In particular, the judgment for expressions is extended with the rules

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

where the judgment $\Sigma; \Delta \vdash \langle\Theta\rangle : [C] \Rightarrow [D]$ types explicit substitutions, and is axiomatized as follows.

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

Support of an expression describes which names the expression may dereference. In line with this semantics, the rule for name dereferencing allows X to be used only if it is present in the support set C, X. 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 32 Consider the ML-like program below.

```
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 the calculus of dynamic binding as follows.

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

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

The variable u is bound to the computation `(X + 1)`, and thus 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 the substitution `<X -> 0>` and the second one is in the scope of `<X -> 1>`. ■

Operational semantics

The evaluation judgment 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. And we still only consider evaluation of expressions e which have empty support.

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 and redexes of the calculus of benign effects. The definition of the syntactic categories that are immediately relevant to the operational semantics of the calculus are sumarized below.

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

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

$$\frac{Y \notin \mathrm{dom}(\Sigma)}{\Sigma, \mathbf{choose}\ (\nu X{:}A.\ e) \longrightarrow (\Sigma, Y{:}A), [Y/X]e} \qquad \frac{}{\Sigma, \langle \sigma \rangle e \longrightarrow \Sigma, \{\sigma\} e}$$

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

Note that the operational semantics does not evaluate under explicit substitutions, and thus uninitialized names will never be encountered during the evaluation. Rather, the expression $\langle \sigma \rangle e$ is reduced by first employing the meta operation $\{\sigma\}e$ to carry out the substitution σ over e, before the evaluation can proceed.

Structural properties and type soundness

The structural properties and the main substitution principles of the calculus for dynamic binding follow closely the presentation from Section 2.2.3. This is not surprising, as the calculus of dynamic binding differs very slightly from the $\Box$ fragment of the modal ν-calculus. As already argued in the previous sections of this chapter, the main distinctions between the two calculi involve: (1) the context Γ is ommitted in the calculus of dynamic binding; (2) functional and ν-abstractions are restricted to bodies with empty support, and (3) explicit substitutions are not restricted to appear only around modal variables. These distintions, however, do not seriously influence the proofs of the main properties.

For example, the explicit substitution principle is a straightforward adaptation of the corresponding explicit substitution principle from Section 2.2.3.

Lemma 37 (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 $\Sigma; \Delta \vdash \langle \Theta \circ \Theta' \rangle : [C_1] \Rightarrow [D]$*

Proof:
The proof is by simultaneous induction on the structure of the derivations. The interesting part is the second induction hypothesis, whose proof utlizies the splitting of $\Psi = \Theta \circ \Theta'$ into two disjoint sets

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

The argument proceeds in an identical as in Section 2.2.3. ■

The calculus of benign effects (and thus, the calculus of dynamic bindin as well), does not contain a notion of ordinary value variables, so the Value substitution principle of the modal ν-calculus (Theorem 12) does not have an equivalent in dynamic binding. However, the Modal substitution principle (Theorem 14) does, because the variables in calculus of dynamic binding really correspond to the modal variables of the modal ν-calculus. Because these are the *only* variables in dynamic binding, we emphasize this fact by renaming the principle into Expressions substitution principle.

Lemma 38 (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: By simultaneous induction on the two derivations. Selected cases are presented below.

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

By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D']$. By the first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D']$. Now the result follows by the definition of substitution, and the typing rule for **box**.

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

By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : \Box_{D'} B'\,[D]$ and also $\Sigma; (\Delta, u{:}A[C], u'{:}B'[D']) \vdash e'' : B\,[D]$. By induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : \Box_{D'} B'\,[D]$ and $\Sigma; (\Delta, u'{:}B'[D']) \vdash [e_1/u]e'' : B\,[D]$. This immediately leads to the result, by the typing rule for **let box**.

■

The next lemma allows for exchanging expressions in context, as long as their types agree. It will be used later in the proofs of Preservation (Lemma 42) and Progress (Lemma 44).

Lemma 39 (Replacement)
If $\Sigma; \cdot \vdash E[e] : A\,[-]$, then there exist a type B such that

1. *$\Sigma; \cdot \vdash e : B\,[-]$, and*

2. *if Σ' extend Σ, and $\Sigma'; \cdot \vdash e' : B\,[-]$, then $\Sigma'; \cdot \vdash E[e'] : A\,[-]$*

Proof:

By induction on the structure of E. The base case when $E = [\ ]$ is obvious. For a more complicated case, consider $E = \langle \sigma, X \to E_1, \Theta \rangle e_1$, where $X{:}B' \in \Sigma$. By derivation, $\Sigma; \cdot \vdash E_1[e] : B'\,[-]$, and the first statement of the lemma follows immediately by the induction hypothesis.

For the second statement of the lemma, consider $\Sigma' \supseteq \Sigma$ and e' such that $\Sigma'; \cdot \vdash e' : B\,[-]$. By induction hypothesis, $\Sigma'; \cdot \vdash E_1[e'] : B'\,[-]$. The result now follows by the typing rules for explicit substitutions.

■

Lemma 40 (Canonical forms)
Let v be a 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 = $ **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 is empty, and can be weakened arbitrarily.

Proof: By a straightforward case analysis. ■

Primitive reduction in the calculus of dynamic binding preserves types, as the Subject reduction lemma shows.

Lemma 41 (Subject reduction)
If $\Sigma; \cdot \vdash e : A\,[-]$ and $\Sigma, e \longrightarrow \Sigma', e'$, then Σ' extends Σ and $\Sigma'; \cdot \vdash e' : A\,[-]$.

Proof: The cases when $e = (\lambda x.\ e')\ v$ or $e = $ **let box** u = **box** e_1 **in** e_2 follow by the expression substitution principle. If $e = $ **choose** $\nu X.\ e_1$ follows by the definition of primitive reduction, and the typing rules.

The only mildly interesting case is when $e = \langle \sigma \rangle e_1$. In this case, by derivation, $\Sigma; \cdot \vdash e_1 : A\,[C_1]$, and $\Sigma; \cdot \vdash \langle \sigma \rangle : [C_1] \Rightarrow [-]$. By the explicit substitution principle, $\Sigma; \cdot \vdash \{\sigma\} e_1 : A\,[-]$. But, by definition of the primitive reductions, it is exactly $\Sigma' = \Sigma$ and $e' = \{\sigma\} e_1$; this concludes the proof. ■

Lemma 42 (Preservation)
If $\Sigma; \cdot \vdash e : A\,[-]$ and $\Sigma, e \longmapsto \Sigma', e'$, then Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : A\,[-]$.

Proof: By evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$. By replacement, there exists B such that $\Sigma; \cdot; \cdot \vdash r : B\,[-]$.

By subject reduction, Σ' extends Σ, and $\Sigma'; \cdot \vdash r' : B\,[-]$. By relacement again, $\Sigma'; \cdot \vdash E[r'] : A\,[-]$. Since $e' = E[r']$, this proves the lemma.

■

Lemma 43 (Unique decomposition)
If e is a closed expression (i.e., e does not contain any free variables, but may contain free names) then 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 stucture of e. A representative case is when e is an application of an explicit substitution. In this case we distinguish three possibilites:

1. $e = \langle\sigma, X \to E_1[Y], \Theta\rangle e_2$. In this case, just pick $E = \langle\sigma, X \to E_1, \Theta\rangle e_2$, and the second statement of the lemma holds.

2. $e = \langle\sigma, X \to e_1, \Theta\rangle e_2$, where e_1 is not a name in context (this case was considered above), nor a value. In this case, by induction hypothesis, $e_1 = E_1[r]$. We pick $E = \langle\sigma, X \to E_1, \Theta\rangle e_2$, and the third statement of the lemma holds.

3. $e = \langle\sigma\rangle e_2$. In this case, pick $E = [\ ]$, $r = e$, and the third statement of the lemma holds.

■

Finally, we can now show that the calculus of dynamic binding satisfies the the usual progress properties, i.e., that the evaluation of well typed closed expressions do not get stuck.

Lemma 44 (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: Because e has empty support, by unique decomposition, e is either a value, or there exists unique E and r such that $e = E[r]$. In case e is not a value, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[-]$. By case analysis of the structure of r, it is clear that there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$. By the rules for evaluation, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$, so we simply pick $e' = E[e_1]$. ■

The progress lemma proves that a well typed term can always be reduced, but does not say anything about the uniqueness of this reduct. And indeed, just as in the modal ν-calculus, this reduct is *not* unique, but the only difference between reducts is due to the different choices of fresh names that may be allocated during the reduction.

Lemma 45 (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: Analogous to the proof of determinacy for the modal ν-calculus (Theorem 19). ■

4.5 State

Syntax and typing

In the calculus of dynamic binding from Section 4.4, names stand for (possibly uninitialized) memory locations and explicit substitutions assign values to locations. In this sense, dereferencing a name corresponds to a *read*, and substituting for a name corresponds to an *update*. But, as the following dynamic binding program illustrates, explicit substitutions may not perform the update *destructively*.

```
let name X : int
in
    <X -> 0>
    let fun f(y: int) : □_X int = box (X + y)
        box u = f 1
    in
        (<X -> 1>u, u + 1)
    end
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 and dynamic binding alone are too weak. This limitation, however, is only to be expected. After all, the calculus of dynamic binding is a calculus of *benign effects*. The modal operator $\Box_C$ may only classify effectful computations that *do not* change the run-time environment in which the program evaluates. Destructively writing into memory certainly performs exactly such a change.

The solution is to *serialize* 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 substitutions. 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 Section 4.2, the serialization of effectful computations is exactly the duty of modal possibility. Thus, if we want to use explicit substitutions to model destructive state update, we need to tie explicit substitutions to $\Diamond$. Intuitively then, we should obtain a whole family $\Diamond_C$ of possibility operators indexed by support sets, where the type $\Diamond_C A$ classifies an explicit substitution for C *paired up* with a computation of type A. More concretely, $\Diamond_C A$ types programs of type A that first *write destructively* into locations C and then compute a value of type A in the new state. This would pleasantly contrast the type $\Box_C A$ that we already used in Section 4.4 to type programs that *read* from locations C before computing a value of type A.

The described typing of the calculus for destructive update will obviously be very similar to simultaneous possibility from Sections 2.1.4 and 2.2. We start the development by defining the following syntactic categories on top of the syntax of the calculus of dynamic binding.

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

As expected, the grammar of types is extended with the family $\Diamond_C A$, whose term constructor is **dia** f, encapsulating a *phrase* f. Phrases are a new syntactic category intended to describe computations which change the global store. The basic phrase 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 empty substitution, we will simply write e instead of $[\cdot, e]$. The changes to the global store are actually enacted by the elimination form **let dia**. This form takes an expression e which evaluates to a phrase, thus carrying a substitution Θ and an expression e_1. The substitution Θ is then promoted into a global store, after which e_1 is evaluated and bound to x, before the evaluation of f is undertaken. The phrase form **let box** $u = e$ **in** f takes a computation internalized by the expression e and binds it to u to be used in the phrase f.

Example 33 Assuming that X and Y are integer names, the expression

```
let dia z = dia [<X->1, Y->2>, 2XY]
in
    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 the evaluation steps to the phrase `[<X->1, Y->2>, X^2 + Y^2 + 4]`. ■

The type system for state with destructive update consists of two mutually recursive judgments: one for typing expressions, and another one for typing phrases. The expression judgment extends the system from Section 4.3, 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 phrase judgment has the form

$$\Sigma; \Delta \vdash f \div_C A\,[D]$$

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

Definition of $\Sigma; \Delta \vdash f \div_C A\,[D]$.

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

$$\frac{\Sigma; \Delta \vdash e : \Diamond_{C_1} A\,[D] \qquad \Sigma; (\Delta, x{:}A) \vdash f \div_{C_2} B\,[C_1]}{\Sigma; \Delta \vdash \textbf{let dia}\ x = e\ \textbf{in}\ f \div_{C_2} B\,[D]}$$

$$\frac{\Sigma; \Delta \vdash e : \Box_C A\,[D] \qquad \Sigma; (\Delta, u{:}A[C]) \vdash f \div_{C_2} B\,[D]}{\Sigma; \Delta \vdash \textbf{let box}\ u = e\ \textbf{in}\ f \div_{C_2} B\,[D]}$$

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

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

The phrase $[\Theta, e]$ is a computation that, when executed, changes the global store according to Θ, and then evaluates e in the changed store. Thus, the typing rule for $[\Theta, e]$ requires that the names used in e are all defined by Θ. In other words, the support of e must match the domain type of Θ. In this respect, the phrase constructor $[\Theta, e]$ is similar, somewhat curiosly, to the constructor for substitution application $\langle\Theta\rangle e$, as indeed witnessed by their typing rules (see Section 4.4). The two constructors, however, have very different operational meanings. The explicit substitution $\langle\Theta\rangle e$ carries out Θ over the expression e. In the phrase $[\Theta, e]$, the substitution Θ is not applied over e; rather, it is composed with the current global store to affect a change of the environment. The first construct provides non-destructive location update, while the second is used when destructive update is required. What is interesting is that both capabilities harmoniously coexist within the system.

The typing rule for **dia** is a judgmental coercion from phrases to expressions. It internalizes persistent computations, so that they can be used as ordinary expressions. To justify the typing rule for **let dia** $x = e$ **in** f on the grounds of its intended operational behavior, observe that $e : \Box_{C_1} A\,[D]$, and therefore e internalizes a phrase consisting of substitution $\Theta : [C_1] \Rightarrow [D]$ and expression $e' : A\,[C_1]$. The role of **let dia** is to institute the substitution Θ as a new global store providing definitions for names in the support C_1, 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 C_1, because the new global store in which f is evaluated defines the names from C_1. We are also free to declare x as being of empty support in the typing of f, because x will always be bound to a value.

Example 34 We will use some further syntactic abbreviations as well. Recall that in the calculus of benign effects, we abbreviated:

$$\begin{array}{rcl}\mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 & = & \mathbf{unbox}\ ((\lambda x.\ \mathbf{box}\ e_2)\ e_1) \\ \mathbf{let\ name}\ X{:}A\ \mathbf{in}\ e & = & \mathbf{unbox}\ (\mathbf{choose}\ (\nu X{:}A.\ \mathbf{box}\ e))\end{array}$$

We need similar constructs in the syntactic category of phrases; we define them in terms of **let val** and **let name** for expressions.

$$\begin{array}{rcl}\mathbf{let\ val}\ x = e\ \mathbf{in}\ f & = & \mathbf{let\ dia}\ y = (\mathbf{let\ val}\ x = e\ \mathbf{in\ dia}\ f)\ \mathbf{in}\ y \\ \mathbf{let\ name}\ X{:}A\ \mathbf{in}\ f & = & \mathbf{let\ dia}\ y = (\mathbf{let\ name}\ X{:}A\ \mathbf{in\ dia}\ f)\ \mathbf{in}\ y\end{array}$$

In contrast to the **let box** construct for phrases, which is primitive in the calculus, and must be present in order to ensure the subformula property, **let val** and **let name** do not eliminate any type and hence do not have any proof theoretic significance. The typing rules for the two are easily derived as

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

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

■

Example 35 If C and D are well-formed supports, then the following are derivable judgments in the calculus of state.

1. $\vdash \lambda x.\, \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in}\ [\cdot, y]) : \Diamond_D A \to \Diamond_C A$, where $C \subseteq D$
2. $\vdash \lambda x.\, \textbf{dia}\ [\cdot, x] : A \to \Diamond A$
3. $\vdash \lambda x.\, \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in let dia}\ z = y\ \textbf{in}\ [\cdot, z]) : \Diamond_C \Diamond_D A \to \Diamond_D A$
4. $\vdash \lambda x.\, \lambda y.\, \textbf{let box}\ u = x\ \textbf{in dia}\ (\textbf{let dia}\ z = y\ \textbf{in}\ [\cdot, u\ z])$ $: \Box_C(A \to B) \to \Diamond_D A \to \Diamond_D B$, where $C \subseteq D$

As an illustration, we present the derivation of the judgment (1).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\Diamond_D A \vdash x : \Diamond_D A} \quad \dfrac{\dfrac{C \subseteq D}{x{:}\Diamond_D A, y{:}A \vdash \langle\cdot\rangle : [C] \Rightarrow [D]} \quad \dfrac{\emptyset \subseteq C}{x{:}\Diamond_D A, y{:}A \vdash y : A\,[C]}}{x{:}\Diamond_D A, y{:}A \vdash [\cdot, y] \div_C A\,[D]}}{x{:}\Diamond_D A \vdash \textbf{let dia}\ y = x\ \textbf{in}\ [\cdot, y] \div_C A}}{x{:}\Diamond_D A \vdash \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in}\ [\cdot, y]) : \Diamond_C A}}{\vdash \lambda x.\, \textbf{dia}\ (\textbf{let dia}\ y = x\ \textbf{in}\ [\cdot, y]) : \Diamond_D A \to \Diamond_C A}$$

As can be noticed, the function (1) simply η-expands its argument x. It illustrates that strengthening at the index supports of $\Diamond$ types is derivable. This is not surprising, as strenghtenning only involves forgetting some entries from the substitution associated with the phrase x. The rest of the expressions generalize the characteristic axioms of the constructive S4 modal possibility introduced in Section 1.1.4. For example, function (2) is a coercion from expressions into phrases 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. In other words,

$$\not\vdash \lambda x.\, \textbf{dia}\ [\cdot, x] : A \to \Diamond_C A$$

However, the following hypothetical judgment is derivable:

$$x{:}A \vdash \textbf{dia}\ [\cdot, x] : \Diamond_C A\,[D] \qquad \text{if } C \subseteq D,$$

as witnessed by the derivation below.

$$\dfrac{\dfrac{\dfrac{C \subseteq D}{x{:}A \vdash \langle\cdot\rangle : [C] \Rightarrow [D]} \quad \dfrac{\emptyset \subseteq C}{x{:}A \vdash x : C}}{x{:}A \vdash [\cdot, x] \div_C A\,[D]}}{x{:}A \vdash \textbf{dia}\ [\cdot, x] : \Diamond_C A\,[D]}$$

Function (3) illustrates that it is only the last layer of $\Diamond$'s that matter; all the additional ones can be ignored. Function (4) takes $x{:}\Box_C(A \to B)$ and $y{:}\Diamond_C A$ as arguments. The argument x embodies a computation $u{:}A \to B[C]$ which depends on names C in order to generate a function of type $A \to B$. The argument y is a computation that provides a term $v{:}A$ and definitions for names in C (and possibly some more, since its index support is $D \supseteq C$). The definitions from y are then placed into the global store and used as an environment for evaluating $u\ v$. ■

Example 36 We can use the new type and term constructors for possibility to serialize the example given at the beginning of the section.

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

In the last line of this program, we abbreviated, and instead of `[<>, (w, z)]`, simply wrote `(w, z)`. The program is well-typed in the judgment of phrases, and has the type **int** × **int**.

We next informally describe the evaluation of this program, with the goal of suplying the intutition for the next section, where we present the operational semantics of the calculus. The evaluation starts by allocating an integer name X, which promply becomes part of the global store, initialized to 0. Then the function f is defined. Notice that we assume recursive function definitions, which are easily added to the language without any technical problems. The evaluation proceeds by computing $f\ 1$, which evaluates to **box** $(X+1)$, so that u is bound to $X+1$. Because global store declares that $X \to 0$, the variable z is bound to 2, which is the value of $u+1$ relative to the current global store. Subsequently, however, the global store is changed into $X \to 1$, and the variable w is bound to the value of the expression u, as computed in this new version of the store. As u is bound to $X+1$, w is assigned the value 2. Thus, the final outcome of the evaluation is the pair $(2, 2)$. Observe that the final result does not depend on the name X; this is enforced by the typing rules for **let name**. As a consequence, X can silently be ommitted from the store at the end of the evaluation. ■

Operational semantics

In this section we develop a call-by-value left-to-right operational semantics for the calculus of state with both the modal constructors □ and ◇. We ignore the phrase constructors **let val** and **let name** 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.

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

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 phrases. We adopt a particular formulation of these judgments which emphasizes the relationship between the simultaneous modal possibility and global state. The expression evaluation judgment has the form

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

and reads: in a context of declared locations Σ and a store σ assigning values to these locations (and some locations may remain uninitialized), the term e steps into e' and possibly introduces new locations Σ'. The evaluation steps cannot change the store σ, as expressions can only read from the store but not write into it. The definition is a straightforward extension of the operational semantics of dynamic binding (Section 4.4).

The judgment for evaluating phrases prescribes evaluation of stateful constructs. It has the form

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

where f steps into f', changing in the process the set of allocated locations from Σ into Σ' and the global store from σ into σ'. The evaluation strategy that we consider will evaluate under the constructor **dia** only if it is found in a let-branch of a **let dia**. This way, the changes to the global store prescribed under **dia** will take place only when they are serialized by a **let dia**. Note that this is not the only possible evaluation strategy, but it is the one that relates simultaneous possibility to global state and destructive update. Following this idea, we extend the categories of values, evaluation contexts and redexes from Section 4.3 as summarized below.

$$\begin{array}{llll}
\textit{Values} & v & ::= & \lambda x{:}A.\ e \mid \mathbf{box}\ e \mid \nu X{:}A.\ e \mid \mathbf{dia}\ f \\
\textit{Value substitutions} & \sigma & ::= & \cdot \mid X \to v, \sigma \\
\textit{Evaluation contexts} & E & ::= & [\ ] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E \mid \langle \sigma, X \to E, \Theta \rangle e \\
\textit{Redexes} & r & ::= & (\lambda x.\ e)\ v \mid \mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ e \mid \mathbf{choose}\ (\nu X.\ e) \mid \langle \sigma \rangle e \mid X \\
\textit{Phrase contexts} & F & ::= & [\ ] \mid \mathbf{let\ dia}\ x = E\ \mathbf{in}\ f \mid \mathbf{let\ dia}\ x = \mathbf{dia}\ F\ \mathbf{in}\ f \mid \\
 & & & \mathbf{let\ dia}\ x = \mathbf{dia}\ [\langle \sigma, X \to E, \Theta \rangle, e]\ \mathbf{in}\ f \mid \\
 & & & \mathbf{let\ dia}\ x = \mathbf{dia}\ [\cdot, E]\ \mathbf{in}\ f \mid \\
 & & & \mathbf{let\ box}\ u = E\ \mathbf{in}\ f \\
\textit{Phrase redexes} & c & ::= & \mathbf{let\ dia}\ x = \mathbf{dia}\ [\sigma, e]\ \mathbf{in}\ f \mid \mathbf{let\ dia}\ x = \mathbf{dia}\ [\cdot, v]\ \mathbf{in}\ f \mid \\
 & & & \mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ f
\end{array}$$

The two evaluation judgments require two primitive reduction relations: a primitive reduction for expressions $\overset{\sigma}{\longrightarrow}$, and a primitive reduction for phrases $\longrightarrow$.

Primitive reductions for expressions.

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

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

Primitive reductions for phrases.

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

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

$$\frac{}{(\Sigma, \sigma), \mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ f \longrightarrow (\Sigma, \sigma), [e/u]f}$$

Evaluation for expressions.

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

Evaluation for phrases.

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

All the rules are fairly straightforward, except the one for primitive reduction of phrases with nonempty substitution. The meaning of this rule is to change the global store according to the phrase substitution and continue evaluating in the new store. Thus, the substitution σ' is moved out of the phrase and composed with σ which is the current global store. Observe that this rule is required in order to preserve the soundness of the operational semantics. In the phrase **let dia** $x =$ **dia** $[\sigma, e]$ **in** f, the type system assumes that the variable x has empty support. Thus, the expression e has to be reduced to a value (as values have empty support), before it can be bound to x.

Structural properties and type soundness

The calculus of state is an extension of the calculus of dynamic binding from Section 4.4 with the possibility judgment and the language constructs corresponding to possibility. It's structural properties and substitution principles, thus, extend the properties of the calculus of dynamic binding, and are also straighforward adaptations of the properties of the modal ν-calculus from 2.2.3. We list the main properties below, and comment on their proofs.

The support weakening lemma is standard, and will be used further in this section in the proof of the Replacement lemma (Lemma 50).

Lemma 46 (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* $\Sigma; \Delta \vdash f \div_{C_1} A\,[D]$

Proof: By a simultaneous induction on the stricture of the three main derivations. ■

The expression substitution principle corresponds to the modal substitution principle from Section 2.2.3.

Lemma 47 (Expression substitution principle)
Let $\Sigma; \Delta \vdash e_1 : A\,[C]$. *Then the following holds:*

1. *if* $\Sigma; (\Delta, u{:}A[C]) \vdash e_2 : B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]e_2 : B\,[D]$
2. *if* $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : [D'] \Rightarrow [D]$, *then* $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : [D'] \Rightarrow [D]$
3. *if* $\Sigma; (\Delta, u{:}A[C]) \vdash f \div_{C_1} B\,[D]$, *then* $\Sigma; \Delta \vdash [e_1/u]f \div_{C_1} B\,[D]$

Proof: By simultaneous induction on the structure of the three derivations. We present the case $f = \textbf{let dia}\ x = e\ \textbf{in}\ f'$ in the proof of the third statement. In thise case, by derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e : \Diamond_{C'}A'\,[D]$, and $\Sigma; (\Delta, u{:}A[C], x{:}A') \vdash f' \div_{C_1} B\,[C']$, for some support C' and type A'. By the first first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e : \Diamond_{C'}A'\,[D]$. By the third induction hypothesis, $\Sigma; (\Delta, x{:}A') \vdash [e_1/u]f' \div_{C_1} B\,[C']$. Now the result follows by the typing rule for **let dia**. ■

The explicit substitution principle is also a straightforward adaptation.

Lemma 48 (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: The proof is by simultaneous induction on the three judgments. It is analogous to the proof of the explicit substitution principle for the modal ν-calculus from Section 2.2.3. We present the case when $f = [\Theta', e]$, in the proof of the third statement.

In this case, by derivation, $\Sigma; \Delta \vdash e : A\,[C_1]$ and $\Sigma; \Delta \vdash \langle\Theta'\rangle : [C_1] \Rightarrow [C]$. By the second induction hypothesis, $\Sigma; \Delta \vdash \langle\Theta \circ \Theta'\rangle : [C_1] \Rightarrow [D]$. Now, result follows by typing rule for phrases. ■

Lemma 49 (Canonical forms)
Let v *be a 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* $v = \textbf{box}\ e$ *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* $v = \textbf{dia}\ f$ *and* $\Sigma; \cdot \vdash f \div_D B\,[C]$

As a consequence, the support of v *is empty, and can be weakened arbitrarily.*

Proof: By a straightforward case analysis. ■

The next Replacement lemma allows expressions and phrases to be exchanged in an expression and phrase contexts respectively. Of course, the replacement expressions and phrases have to match the type of the expression or the phrase that is being replaced. Notice that the Replacement lemma in this section, unlike the Replacement lemma of the calculus for dynamic binding, considers non-empty supports in the typing judgments. The reason is that, unlike in dynamic binding, the calculus of state allows evaluation of expressions and phrases with non-empty support C, as long as the names from C are initialized by the global store.

Lemma 50 (Replacement)

1. *If $\Sigma; \cdot \vdash E[e] : A\,[C]$, then there exists a type B such that*
   - *(a) $\Sigma; \cdot \vdash e : B\,[C]$, and*
   - *(b) if Σ' extends Σ, and $\Sigma'; \cdot \vdash e' : B\,[C]$, then $\Sigma'; \cdot \vdash E[e'] : A\,[C]$*
2. *If $\Sigma; \cdot \vdash F[e] \div_C A\,[D]$, then there exists a type B such that*
   - *(a) $\Sigma; \cdot \vdash e : B\,[D]$, and*
   - *(b) if Σ' extends Σ and $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash F[e'] \div_C A\,[D]$*
3. *If $\Sigma; \cdot \vdash F[f] \div_C A\,[D]$, then there exists a type B and support C_1 such that*
   - *(a) $\Sigma; \cdot \vdash f \div_{C_1} B\,[D]$, and*
   - *(b) if Σ' extends Σ and D_1 is a support set such that $\Sigma'; \cdot \vdash f' \div_{C_1} B\,[D_1]$, then $\Sigma'; \cdot \vdash F[f'] \div_C A\,[D_1]$*

Proof: By simultaneous induction on the structure of the contexts E and F. We present the proofs for induction hypotheses (2) and (3), as the case (1) is similar to the proof of Replacement for dynamic binding (Lemma 39).

For the induction hypothesis (2), the following cases may appear.

case $F = \textbf{let dia}\ x = E_1\ \textbf{in}\ f$. By derivation, $\Sigma; \cdot \vdash E_1[e] : \Diamond_{C_1} A_1\,[D]$, and $\Sigma; x{:}A_1 \vdash f \div_C A\,[C_1]$. By first induction hypothesis, there exists B such that $\Sigma; \cdot \vdash e : B\,[D]$. Also, if $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash E_1[e'] : \Diamond_{C_1} A_1\,[D]$. Conclusion now follows by typing rule for **let dia**.

case $F = \textbf{let dia}\ x = \textbf{dia}\ F_1\ \textbf{in}\ f$. By derivation, $\Sigma; \cdot \vdash F_1[e] \div_{C_1} A_1\,[D]$, and $\Sigma; x{:}A_1 \vdash f \div_C A\,[C_1]$. By second induction hypothesis, there exists B such that $\Sigma; \cdot \vdash e : B\,[D]$. Also, if $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash F_1[e'] \div_{C_1} A_1\,[D]$. The result again follows by typing for **let dia**.

case $F = \textbf{let dia}\ x = \textbf{dia}\ [\langle \sigma, X \rightarrow E_1, \Theta \rangle, e]\ \textbf{in}\ f$, where $X{:}B_1 \in \Sigma$. By derivation, $\Sigma; \cdot \vdash E_1[e] : B_1\,[D]$, and $\Sigma; x{:}A_1 \vdash f \div_C A\,[C_1]$. By first induction hypothesis, there exists B such that $\Sigma; \cdot \vdash e : B\,[D]$. Also, if $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash E_1[e'] : B_1\,[D]$. Once again, the typing for **let dia** lead to the required conclusion.

case $F = \textbf{let dia}\ x = \textbf{dia}\ [\cdot, E_1]\ \textbf{in}\ f$. By derivation, $\Sigma; \cdot \vdash E_1[e] : A_1\,[C_1]$, where $C_1 \subseteq D$, and $\Sigma; x{:}A_1 \vdash f \div_C A\,[C_1]$. By support weakening, $\Sigma; \cdot \vdash E_1[e] : A_1\,[D]$ and $\Sigma; x{:}A_1 \vdash f \div_C A\,[D]$. By first induction hypothesis, there exists B such that $\Sigma; \cdot \vdash e : B\,[D]$. Also, if $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash E_1[e'] : A_1\,[D]$. Finally, use the typing rule for **let dia** again to conclude the proof.

For the induction hypothesis (3), the following cases may appear.

case $F = [\,]$. In this case, obviously, pick $B = A$, and $C_1 = C$ to finish the proof.

case $F = \textbf{let dia } x = \textbf{dia } F_1 \textbf{ in } f_1$. By derivation, $\Sigma; \cdot \vdash F_1[f] \div_{C'} A'\,[D]$, and $\Sigma; x{:}A' \vdash f_1 \div_C A\,[C']$. By third induction hypothesis, there exist B and C_1 such that $\Sigma; \cdot \vdash f \div_{C_1} B\,[D]$. Also, if $\Sigma'; \cdot \vdash f' \div_{C_1} B\,[D_1]$, then $\Sigma'; \cdot \vdash F_1[f'] \div_{C'} A'\,[D_1]$. The result again follows by typing rules for **let dia**.

■

The Subject reduction lemma establishes that primitive reductions preserve types nad supports. Notice that in the calculus of state, the evaluation is always undertaken relative to a global store $\sigma : [C] \Rightarrow [\,]$, which provides definitions for a certain set of names C that the evaluated expressions and phrases are allowed to dereference. Notice that the evaluation of expressions may only depend on the global store σ, but the evaluation of phrases may change σ into some new $\sigma' : [C'] \Rightarrow [\,]$. Of course, in the typing of the new global store, C' will always be a well-formed support set, as the lemma below postulates.

Lemma 51 (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 $C' \subseteq \mathrm{dom}(\Sigma')$*

Proof: By case analysis of the possible reductions. We present the selected cases.

case $e = \langle\sigma'\rangle e_1$. By derivation, $\Sigma; \cdot \vdash e_1 : A\,[C']$, and $\Sigma; \cdot \vdash \langle\sigma'\rangle : [C'] \Rightarrow [\,]$. By explicit substitution principle, $\Sigma; \cdot \vdash \{\sigma'\}e_1 : A\,[\,]$. By definition, $e' = \{\sigma\}e_1$, which finishes the proof.

case $e = X$, where $X{:}A \in \Sigma$. By derivation, $X \in C$, and thus by typing for substitutions $\Sigma; \cdot \vdash \sigma(X) : A\,[\,]$. 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 $\Sigma; \cdot \vdash \sigma(X) : A\,[C]$.

case $f = \textbf{let dia } x = \textbf{dia } [\sigma_1, e] \textbf{ in } f_1$. By definition, $\Sigma' = \Sigma$ and $\sigma' = \sigma \circ \sigma_1$. By derivation, $\Sigma; \cdot \vdash e : B\,[C']$, and $\Sigma; \cdot \vdash \langle\sigma_1\rangle : [C'] \Rightarrow [C]$, and $\Sigma; x{:}B \vdash f_1 \div_D A\,[C']$. By explicit substitution principle, $\Sigma; \cdot \vdash \langle\sigma \circ \sigma_1\rangle : [C'] \Rightarrow [\,]$. Result follows by typing rule for **let dia**.

case $f = \textbf{let dia } x = \textbf{dia } [\cdot, v] \textbf{ in } f_1$. By definition, $\Sigma' = \Sigma$ and $\sigma' = \sigma$ and $C_1 = C$. By derivation, $\Sigma; \cdot \vdash v : B\,[C_1]$ for some $C_1 \sqsubseteq C$, and $\Sigma; x{:}B \vdash f_1 \div_D A\,[C']$. By cannonical forms lemma, $\Sigma; \cdot \vdash v \div_B [\,]$. By support weakening, $\Sigma; x{:}B \vdash f_1 \div_D A\,[C]$. Finally, by the expression substitution principle, $\Sigma; \cdot \vdash [v/x]f_1 \div_D A\,[C]$.

■

The Preservation lemma extends the result of Subject reduction, which was valid only on primitive reductions, to the evaluation relation.

Lemma 52 (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* $C' \subseteq \mathrm{dom}(\Sigma')$

Proof: The proof of statement (1), proceeds as follows. By evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \overset{\sigma}{\longrightarrow} \Sigma', r'$ and $e' = E[r']$. By the 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 again, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$. Since $e' = E[r']$ this proves statement (1).

To prove the statement (2), observe that by the 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 $\Sigma, r \overset{\sigma}{\longrightarrow} \Sigma', e'$ and $f' = F[e']$, and $\sigma' = \sigma$ and $C_1 = C$. By the 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, $\Sigma'; \cdot \vdash F[e'] : A\,[C]$.

On the other hand, 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 again, $\Sigma'; \cdot \vdash F[c'] \div_D A\,[C']$. ■

Lemma 53 (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* $\Sigma, r \overset{\sigma}{\longrightarrow} \Sigma', e'$.

2. *if* $\Sigma; \cdot \vdash c \div_D A\,[C]$, *then there exist a phrase* f', *a value substitution* σ' *and a context* Σ', *such that* $(\Sigma, \sigma), c \longrightarrow (\Sigma', \sigma'), f'$.

Proof: By case analysis over possible redexes. For example, in the statement (1), when $r = X$, for some name X, we can pick $\Sigma' = \Sigma$ and $e = \sigma(X)$. The other cases of statement (1), as well as the statement (2) are also easy to establish. ■

Lemma 54 (Unique decomposition)

1. *If* e *is a closed expression (i.e.,* e *does not contain any free variables, but it may contain free names), then either:*

   (a) e *is a value, or*

   (b) $e = E[r]$ *for a unique evaluation context* E *and a redex* r.

2. *If* f *is a closed phrase, then either:*

   (a) $f = [\Theta, e]$ *for some substitution* Θ *and expression* e, *or*

   (b) $f = F[r]$ *for a unique phrase context* F *and term redex* r, *or*

   (c) $f = F[c]$ *for a unique phrase context* F *and phrase redex* c.

Proof: Straightforwaed, by induction on the structure of e and f. ■

As customary by now, we proceed to prove that in the calculus of state, well-typed closed expressions and phrases do not get stuck, and that reductions from one and the same expression or a phrase differ only in the choice of new names. These claims are formalized by the Progress and Determinacy lemmas below.

Lemma 55 (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 phrase f', a context Σ', and a value substitution σ', such that $(\Sigma, \sigma), f \longmapsto (\Sigma', \sigma'), f'$*

Proof: The proof of statement (1) proceeds as follows. By unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$. If e is not a value, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$. By progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \overset{\sigma}{\longrightarrow} \Sigma', e_1$. By evaluation rules, $\Sigma, E[r] \overset{\sigma}{\longmapsto} \Sigma', E[e_1]$. Now, we can pick $e' = E[e_1]$, to finish the proof.

To prove statement (2), notice that, by the 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 $f = F[c]$. In the second case, by replacement lemma, there exists B such that $\Sigma; \cdot \vdash r : B\,[C]$. By progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \overset{\sigma}{\longrightarrow} \Sigma', e_1$. Then we can pick $f' = F[e_1]$. In the third case, by replacement lemma, there exists a type B and support C_1 such that $\Sigma; \cdot \vdash c \div_{C_1} B\,[C]$. By progress for $\longrightarrow$, there exists a phrase f_1, a context Σ' and a substitution σ', such that $(\Sigma, \sigma), c \longmapsto (\Sigma', \sigma'), f_1$. In this case, we can pick $f' = F[f_1]$. ■

Lemma 56 (Determinacy)

1. *If e, e_1, e_2 are terms such that $\Sigma, e \overset{\sigma}{\longrightarrow}{}^n \Sigma_1, e_1$ and $\Sigma, e \overset{\sigma}{\longrightarrow}{}^n \Sigma_2, e_2$, 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 phrases 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 proof of the first statement is analogous to the proof of determinacy for dynamic binding, so we omit it here. The second lemma statement is trivial, because there are no primitive phrase constructors that introduce fresh names. ■

4.6 Exceptions

Syntax and typing

Raising an exception is a control-flow effect – it causes the execution of the program to make a jump and continue from another point. Along the jump, the exception passes a value, to be used by the program at the destination point of the jump. Exactly where and how the execution of the program resumes, is determined by the *exception handler*. The handler takes as argument the value that is passed by the exception, and then proceeds with execution. Thus, a computation that may raise an exception is, in a sense, *partial.* It must be executed in an environment in which a handler for the exception is specified, or else it may not produce a result. Notice, however, that exceptions are *benign effects.* Unlike writing into memory, raising an exception does not cause a permanent change in the environemnt.

In this section we develop a calculus of exceptions, based on the core fragment of the calculus for benign effects from Section 4.3. The idea is to assign a name to each exception, which could then be propagated and tracked by the type system. To be able to raise and handle exceptions, we need further constructs specific only to exceptions, so we extend the syntax of our language as follows.

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

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

An exception handler is defined as a finite set of *exception patterns.* A pattern $Xz \to e$ associates the exception X with the expression e; the variable z is bound in the pattern. Whenever X is raised with some value v, it will be handled by evaluating the expression $[v/z]e$. Given a handler Θ, its domain $\mathrm{dom}(\Theta)$ is defined as the set

$$\mathrm{dom}(\Theta) = \{X \in \mathcal{N} \mid Xz \to e \in \Theta\}$$

Every exception $X \in \mathrm{dom}(\Theta)$ must be associated with a unique pattern of Θ.

An exception handler Θ defines a unique map $[\![\Theta]\!] : \mathcal{N} \to \textit{Values} \to \textit{Expressions}$ as follows.

$$[\![\Theta]\!](X)(v) = \begin{cases} [v/z]e & \text{if } Xz \to e \in \Theta \\ \mathbf{raise}_X\ v & \text{otherwise} \end{cases}$$

We will frequently identify the handler Θ with the function $[\![\Theta]\!]$, and write $\Theta(X)(v)$ instead of $[\![\Theta]\!](X)(v)$. According to the above definition, if X is an exception such that $X \notin \mathrm{dom}(\Theta)$, then Θ simply propagates X further.

Example 37 Assuming X and Y are integer names, the following are well-formed expressions.

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

The expressions evaluate to 13, 0 and 1, respectively. Expression (1) raises the exception Y, passing 10 along. This is handled by the pattern $Yy \to y + 3$, to produce 13. Expression (2) raises X with value 0, but while handling X it raises Y with value 0, which is finally handled by the outside handler $\langle Yy \to y\rangle$, to produce 0. Expression (3) raises X with 0, which is propagated by the inside handler, and then handled by the outside handler $\langle Xx \to x + 1\rangle$, to return 1. ■

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

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

and it simply extends the judgment from the core fragment presented in Section 4.3 with the new rules for **raise** and **handle**. The specific characteristic 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

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

and the handler Θ will be given the type $[C] \overset{A}{\Rightarrow} [D]$ 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 below, and we briefly comment on them.

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

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

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

Definition of $\Sigma; \Delta \vdash \langle \Theta \rangle : [C] \overset{A}{\Rightarrow} [D]$.

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

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

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 context 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 empty exception handler $\langle\ \rangle$ only propagates whichever exceptions it encounters. 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 inductively checks 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 38 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 $\Box_{\texttt{EMPTY}}$`intlist`. This is required because the body of `tail` raises an exception, and, as explained in Section 4.3, all the effects in function bodies must be boxed.

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

■

Before we proceed to describe the operational semantics of the exception calculus, let us outline some of its properties and how they relate to other treatments of exceptions in functional languages.

First of all, exceptions in our calculus are second class. They are not values and cannot be bound to variables. Correspondingly, exceptions 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.

It is also instructive to compare our calculus with the monadic formulation of exceptions from Section 4.1.3. To that end, we recall Example 26, where the exception monad $\bigcirc$ provides for a unique exception of type E. The definition of the monad $\bigcirc$ and its related term constructors is given as follows.

$$\begin{array}{rcl}
\bigcirc A & = & A + E \\
\mathbf{comp}\ e & = & \mathbf{inl}\ e \\
\mathbf{let\ comp}\ x = e_1\ \mathbf{in}\ e_2 & = & \mathbf{case}\ \ e_1\ \mathbf{of\ inl}\ x \Rightarrow e_2 \mid \mathbf{inr}\ y \Rightarrow \mathbf{inr}\ y \\
\mathbf{raise} & : & E \Rightarrow \bigcirc A \\
\mathbf{raise}\ e & = & \mathbf{inr}\ e \\
\mathbf{handle} & : & \bigcirc A \Rightarrow (E \Rightarrow B) \Rightarrow B \\
\mathbf{handle}\ e\ h & = & \mathbf{case}\ \ e\ \mathbf{of\ inl}\ v \Rightarrow v \mid \mathbf{inr}\ exn \Rightarrow h\ exn
\end{array}$$

In this definition, the operational semantics given to all the constructs relies on the standard operational semantics associated with disjoint sums. For example, is we assume that $f : \mathbf{int} \Rightarrow$

$\bigcirc$**int**, then the following program adds the results of f 1 and f 2. If the evaluation of any of the two function applications raises an exception, the overall computed result is zero.

```
handle (let comp x1 = f 1
            comp x2 = f 2
        in
            comp (x1 + x2)
        end) (λexn. 0)
```

In our calculus of exceptions, the equivalent of the above program may be written in several ways, depending on the evaluation order that the programmer may wish to specify. For example, let us assume that $X{:}E$ is an exception name, and that $f : \mathbf{int} \to \Box_X\mathbf{int}$. Then the operational behavior of the previous monadic program is exhibited by the following program in the calculus of exceptions.

```
(let val x1 = unbox (f 1)
     val x2 = unbox (f 2)
 in
     x1 + x2
 end) handle <X exn -> 0>
```

However, because exceptions are benign effects, the computations internalized by f 1 and f 2 are independent of each other. There is no need to first evaluate and unbox f 1 and then evaluate and unbox f 2. For example, we could write the following program that computes the same results.

```
let box u1 = f 1
    box u2 = f 2
in
    (u1 + u2) handle <X exn -> 0>
end
```

The first two **let box** branches of this program evaluate the expressions f 1 and f 2 in that order to obtain boxed computations **box** e_1 and **box** e_2, but they *do not evaluate* e_1 and e_2. The computations e_1 and e_2 are substituted for u_1 and u_2, and only then is the execution of $(e_1 + e_2)$ attempted, in the order specified by the operational semantics of addition. Following a similar idea, an even more compact way to compute the sum of f 1 and f 2 is given simply as

```
(unbox (f 1) + unbox (f 2)) handle <X exn -> 0>
```

As a conclusion, the calculus of exceptions – and more generally, the calculus of benign effects based on modal necessity – allows programs that are uncommitted about the evaluation order of its effects. The evaluation order is eventually determined by the operational semantics, but it is not necessary to make this order explicit in the program. This is the major difference between the treatment of benign effects and persistent effects. It is also the major difference between the modal operator $\Box$ on one hand, and the monad $\bigcirc$ and the modal operator $\Diamond$ on the other hand.

Note that this distinction may potentially have consequences for the efficiency of exceptional programs. In the monadic case, an expression $e : \bigcirc A$ either evaluates to a value, or raises an exception. The outcome of the evaluation of e has to be *tagged* (with **inl** or **inr**) in order to distinguish between the two cases, and this tag has to be *checked* at run time whenever e is used[2]. In the modal case, the effectful computation boxed in the expression $e : \Box_X A$ will only be evaluated within the scope of some handler for X. This evaluation can only produce a value, and cannot result with an unhandled exception. In the modal case, there cannot exists a raised exceptions that is not handled, so there is no need for tagging and tag checking.

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, or inner-most handler enclosing this **raise**. For that purpose, we define a special subclass of evaluation contexts, called *pure evaluation contexts.*

Definition 57 (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 identify, within each evaluation context E, the handling construct (if any) that is closest to the hole of E, as stated by the following lemma.

Lemma 58 (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*
   $E = E'[P'\ \mathbf{handle}\ \langle\Theta\rangle]$.

Proof: By induction on the structure of E. We present selected cases.

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

[2] Strictly speaking, if the call-by-name evaluation strategy is used, then exceptional computations are only evaluated when explicitly forced by **let comp**. In other words, the tagged values never appear as intermediate results, except immediately before handling. However, if any other evaluation strategy is used, the definition of the exception monad will insist on tagging and tag checking.

case $E = E_1$ **handle** Θ. By induction hypothesis, E_1 is either pure, in which case pick $E' = []$ and $P' = E_1$, or $E_1 = E'_2[P'_2$ **handle** $\Theta_2]$, in which case pick $E' = E'_2$ **handle** Θ and $P' = P'_2$.

■

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

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

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

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 37, where it is the exception Y which is raised and eventually handled.

Structural properties and type soundness

Before proceeding to prove the basic properties of the calculus of exceptions, we first summarize its basic syntactic constructs.

$$\begin{array}{llcl}
\textit{Expressions} & e & ::= & u \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \textbf{box}\ e \mid \textbf{let box}\ u = e_1\ \textbf{in}\ e_2 \mid \\
 & & & \nu X{:}A.\ e \mid \textbf{choose}\ e \mid \\
 & & & \textbf{raise}_X\ e \mid e\ \textbf{handle}\ \langle\Theta\rangle \\
\textit{Exception handlers} & \Theta & ::= & \cdot \mid Xz \to e, \Theta \\
\textit{Values} & v & ::= & \lambda x{:}A.\ e \mid \textbf{box}\ e \mid \nu X{:}A.\ e \\
\textit{Evaluation contexts} & E & ::= & [\,] \mid E\ e_1 \mid v_1\ E \mid \textbf{let box}\ u = E\ \textbf{in}\ e \mid \textbf{choose}\ E \mid \\
 & & & \textbf{raise}_X\ E \mid E\ \textbf{handle}\ \langle\Theta\rangle \\
\textit{Pure contexts} & P & ::= & [\,] \mid P\ e_1 \mid v_1\ P \mid \textbf{let box}\ u = P\ \textbf{in}\ e \mid \textbf{choose}\ P \mid \\
 & & & \textbf{raise}_X\ P \\
\textit{Redexes} & r & ::= & (\lambda x.\ e)\ v \mid \textbf{let box}\ u = \textbf{box}\ e\ \textbf{in}\ e \mid \textbf{choose}\ (\nu X.\ e) \mid \\
 & & & v\ \textbf{handle}\ \langle\Theta\rangle \mid P[\textbf{raise}_X\ v]\ \textbf{handle}\ \langle\Theta\rangle
\end{array}$$

The Expression substitution principle for the exception calculus is similar to the Expression substitution principle from the calculus of dynamic binding and state, except that it now includes a statement about exception handlers, rather than a statement about explicit substitutions.

Lemma 59 (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* $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : [D'] \overset{B}{\Rightarrow} [D]$, *then* $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : [D'] \overset{B}{\Rightarrow} [D]$

Proof: By simultaneous induction on the structure of e and Θ. We just present the cases that are specific to exceptions.

case $e_2 = \textbf{raise}_X\ e'$, where $X{:}B' \in \Sigma$, and $X \in D$. By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D]$. By induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D]$. The result follows by the typing for **raise**.

case $e_2 = e'$ **handle** Θ. By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B\,[D']$, and $\Sigma; (\Delta, u{:}A[C]) \vdash \langle\Theta\rangle : [D'] \overset{B}{\Rightarrow} [D]$. By first induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D']$. By second induction hypothesis, $\Sigma; \Delta \vdash \langle[e_1/u]\Theta\rangle : [D'] \overset{B}{\Rightarrow} [D]$. The case is now proved, by using the typing rules for **handle**.

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

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

■

The replacement lemma now has to account for both pure and impure contexts. Because pure contexts do not allow a handler acting on the hole of the context, placing an expression within a pure context preserve the expression's support. That is not necessarily the case with ordinary evaluation contexts.

Lemma 60 (Replacement)

1. *If* $\Sigma; \cdot \vdash P[e] : A\,[C]$, *then there exist a type* B *such that*

   (a) $\Sigma; \cdot \vdash e : B\,[C]$, *and*

   (b) *if* Σ' *extends* Σ, *and* $\Sigma'; \cdot \vdash e' : B\,[C]$, *then* $\Sigma'; \cdot \vdash P[e'] : A\,[C]$

2. *If* $\Sigma; \cdot \vdash E[e] : A\,[C]$, *then there exist a type* B *and a support* D *such that*

   (a) $\Sigma; \cdot \vdash e : B\,[D]$, *and*

   (b) *if* Σ' *extends* Σ *and* $\Sigma'; \cdot \vdash e' : B\,[D]$, *then* $\Sigma'; \cdot \vdash E[e'] : A\,[C]$

Proof: The first statement is proved by induction on the structure of the pure context P. For an example, consider the case when $P = \textbf{raise}_X\ P_1$, for $X{:}B' \in \Sigma$, and $X \in C$. In this case, by derivation, $\Sigma; \cdot \vdash P_1[e] : B'\,[C]$. By induction hypothesis, there exist B such that $\Sigma; \cdot \vdash e : B\,[C]$. Again by induction hypothesis, for every e' such that $\Sigma'; \cdot \vdash e' : B\,[C]$, we have $\Sigma'; \cdot \vdash P_1[e'] : B'\,[C]$. Now the conclusion follows by the typing rule for **raise**.

To prove the second statement, by Evaluation context decomposition lemma (Lemma 58, we need only consider two cases.

case $E = P$. This case follows from the already proved replacement for pure contexts.

case $E = E_1[P\ \mathbf{handle}\ \Theta]$. In this case, by induction hypothesis, there exist B' and D' such that $\Sigma; \cdot \vdash P[e]\ \mathbf{handle}\ \Theta : B'\,[D']$. By typing, $\Sigma; \cdot \vdash P[e] : B'\,[D'']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : [D''] \overset{B'}{\Rightarrow} [D']$. By replacement for pure contexts, there exists B such that $\Sigma; \cdot \vdash e : B\,[D'']$. Also, for every e' such that $\Sigma'; \cdot \vdash e' : B\,[D]$, we have $\Sigma'; \cdot \vdash P[e'] : B'\,[D'']$. The result now follows by typing for **handle**.

■

Lemma 61 (Canonical forms)
Let v be a 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 $A = A_1 \not\twoheadrightarrow A_2$, then $v = \nu X{:}A_1.\ e$ and $(\Sigma, X{:}A_1); \cdot \vdash e : A_2\,[\,]$*

As a consequence, the support of v is empty, and can be weakened arbitrarily.

Proof: By case analysis on the structure of values. ■

The next step of the development is the Subject reduction lemma. Notice that the subject reduction for exceptions differs from the subject reduction of dynamic binding. The semantics of dynamic binding only reduces expressions of empty support, while with exceptions we need to reduce under an exception handler. This is reflected in the subject reduction lemma, where we now allow arbitrary supports C.

Lemma 62 (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: By simple case analysis over possible redexes. We consider two cases in detail.

case $e = v\ \mathbf{handle}\ \Theta$. By derivation, $\Sigma; \cdot \vdash v : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : [C'] \overset{A}{\Rightarrow} [C]$. 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 $X{:}B' \in \Sigma$. By derivation, $\Sigma; \cdot \vdash P[\mathbf{raise}_X\ v] : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : [C'] \overset{A}{\Rightarrow} [C]$. By replacement lemma, there exists a type B such that $\Sigma; \cdot \vdash \mathbf{raise}_X\ v : B\,[C']$. By typing rules, there must be $X \in C'$, and $\Sigma; \cdot \vdash v : B'\,[C']$. By canonical forms lemma, support of a value is empty, i.e., $\Sigma; \cdot \vdash v : B'\,[-]$. Now, by the well-typing of the handler Θ, $\Sigma; \cdot \vdash \Theta(X)(v) : A\,[C]$. Since $\Sigma, e \longrightarrow \Sigma, \Theta(X)(v)$, this finishes the proof.

■

The Preservation lemma now generalizes Subject reduction to the evaluation judgment. For purposes of generality, we follow the statement of the Subject reduction, and allow arbitrary supports C in the statement of Preservation.

Lemma 63 (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: By evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$. By replacement lemma, there exist B and D such that $\Sigma; \cdot; \cdot \vdash r : B\,[D]$. By subject reduction, Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash r' : B\,[D]$. By relacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$. Since $e' = E[r']$, this proves the lemma. Notice how the proof appeals in an essential way to the subject reduction lemma with non-empty supports. ■

The following lemma shows that a closed well typed redex can always be reduced. Again, as in the case of Subject reduction and Preservation, we consider redexes with a general (not necessarily empty) support C. This will be used in an essential way in the proof of the Progress lemma below (Lemma 66).

Lemma 64 (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 straightforward case analysis. We only present two cases.

case $r = v$ **handle** Θ. By reduction rules, Σ, v **handle** $\Theta \longrightarrow \Sigma, v$. Pick $\Sigma' = \Sigma$ and $e' = v$.

case $r = P[\mathbf{raise}_X\ v]$ **handle** Θ, where $X{:}B \in \Sigma$. By derivation, $\Sigma; \cdot \vdash P[\mathbf{raise}_X\ v] : A\,[C']$, and $\Sigma; \cdot \vdash \langle\Theta\rangle : [C'] \overset{A}{\Rightarrow} [C]$. By replacement lemma, there exists B' such that $\Sigma; \cdot \vdash \mathbf{raise}_X\ v : B'\,[C']$. By typing rules, it must be $X \in C'$, and thus $\Theta(X)(v)$ is well-defined. Now pick $\Sigma' = \Sigma$ and $e' = \Theta(X)(v)$.

■

The unique decomposition lemma is standard.

Lemma 65 (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.*

Proof: By induction on the structure of the expression e. ■

Finally, we can establish the Progress and Determinacy lemmas below.

Lemma 66 (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: Because e has empty support, by unique decomposition lemma, e is either a value, or there exists unique E and r such that $e = E[r]$. If e is not a value, by replacement lemma, there exists B and C such that $\Sigma; \cdot \vdash r : B\,[C]$. By progress for $\longrightarrow$, there exists Σ' and e_1 such that $\Sigma, r \longrightarrow \Sigma', e_1$. By evaluation rules, $\Sigma, E[r] \longmapsto \Sigma', E[e_1]$. Now, we can pick $e' = E[e_1]$, to complete the proof. ■

Lemma 67 (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 rest follows by induction on n, using the property that if $\Sigma, e \longmapsto^n \Sigma', e'$, then $\pi(\Sigma), \pi(e) \longmapsto^n \pi(\Sigma'), \pi(e')$. In case $n = 1$, we analyse the possible reduction cases.

1. 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.

2. 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.

■

4.7 Catch and throw

Syntax and typing

The catch-and-throw calculus is a simplification of the calculus of exceptions. We consider it here in its own right in order to illustrate a different notion of handling. It will also provide some intuition for the calculus of composable continuation in Section 4.8. 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.

$$\textit{Expressions} \quad e \quad ::= \quad \ldots \mid \textbf{throw}_X\ e \mid \textbf{catch}_X\ e$$

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.

Because the notion of handling in the catch-and-throw calculus is so simple when compared to exceptions, we only need the typing judgment for expressions $\Sigma; \Delta \vdash e : A\,[C]$. It is not necessary to define the judgment for handlers $\Sigma; \Delta \vdash \langle\Theta\rangle : [C] \overset{A}{\Rightarrow} [D]$. The meaning of $\Sigma; \Delta \vdash e : A\,[C]$ is that e has type A and may throw to destination points whose names are listed in the support C. The supports are sets, just like in the calculus of exceptions. The typing rules of the calculus are presented below.

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

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

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 against which the argument expression is checked.

Example 39 The following terms (adapted from [Kam00a]) are well-typed in our catch-and-throw calculus.

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

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

The first term evaluates to 0, because the addition with 1 is skipped over by a **throw**. In the second term, the **catch** is pushed further inside, to preserve this addition, and so the term evaluates to 1.

■

Example 40 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.

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

val it = 0 : int
```

■

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 4.3. We first need to define the new redexes, corresponding to the new **catch** and **throw** constructs, and extend the syntactic category of evaluation contexts of the core calculus of benign effects from Section 4.3.

$$\begin{array}{llcl}
\textit{Redexes} & r & ::= & (\lambda x.\ e)\ v \mid \mathbf{let\ box}\ u = \mathbf{box}\ e\ \mathbf{in}\ e \mid \mathbf{choose}\ (\nu X.\ e) \mid \\
 & & & \mid \mathbf{catch}_X\ v \mid \mathbf{catch}_X\ E[\mathbf{throw}_X\ v] \\
\textit{Evaluation contexts} & E & ::= & [\ ] \mid E\ e_1 \mid v_1\ E \mid \mathbf{let\ box}\ u = E\ \mathbf{in}\ e \mid \mathbf{choose}\ E \mid \\
 & & & \mathbf{throw}_X\ E \mid \mathbf{catch}_X\ E
\end{array}$$

In the redex $\mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$ it is assumed that the context E is X-pure, i.e., E does not contain a $\mathbf{catch}_X$ construct acting on the hole of E, altough E is allowed to catch names other than X. The relation of primitive reductions from Section 4.3 is extended with the following new cases.

$$\begin{array}{rcl}
\Sigma, \mathbf{catch}_X\ v & \longrightarrow & \Sigma, v \\
\Sigma, (\mathbf{catch}_X\ E[\mathbf{throw}_X\ v]) & \longrightarrow & \Sigma, v, \quad E \text{ is } X\text{-pure}
\end{array}$$

Similar 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.

Structural properties and type soundness

We start the exploration of the basic structural properties of the catch and throw calculus by considering the appropriate expression substitution principle. The principle is standard, and analogous to the expression substitution principles already proved for the calculi of dynamic binding and exceptions.

Lemma 68 (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 $e_2 = \mathbf{throw}_X\ e'$, where $X{:}B' \in \Sigma$, and $X \in D$. By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B'\,[D]$. By induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B'\,[D]$. The conclusion now follows by the typing rule for **throw**.

case $e_2 = \mathbf{catch}_X\ e'$, where $X{:}B \in \Sigma$. By derivation, $\Sigma; (\Delta, u{:}A[C]) \vdash e' : B\,[D, X]$. By induction hypothesis, $\Sigma; \Delta \vdash [e_1/u]e' : B\,[D, X]$. The last step of the proof now applies the typing rule for **catch**.

■

The replacement lemma needs to take into account that **catch** expressions may be acting on the hole of the context E, thus changing the support of enclosed expression.

Lemma 69 (Replacement)
If $\Sigma; \cdot \vdash E[e] : A\,[C]$, then there exist a type B and a support D such that

1. *$\Sigma; \cdot \vdash e : B\,[D]$, and*

2. *if Σ' extends Σ and $\Sigma'; \cdot \vdash e' : B\,[D]$, then $\Sigma'; \cdot \vdash E[e'] : A\,[C]$*

Proof: By induction on the structure of E.

case $E = \mathbf{throw}_X\ E_1$, where $X{:}B' \in \Sigma$, and $X \in C$. By derivation, $\Sigma; \cdot \vdash E_1[e] : B'\,[C]$. By induction hypothesis, there exist B and D such that $\Sigma; \cdot \vdash e : B\,[D]$. Again by induction hypothesis, for every e' such that $\Sigma'; \cdot \vdash e' : B\,[D]$, we have $\Sigma'; \cdot \vdash E_1[e'] : B'\,[C]$. Now the conclusion follows by the typing rules.

case $E = \mathbf{catch}_X\ E_1$, and $X{:}A \in \Sigma$. By derivation, $\Sigma; \cdot \vdash E_1[e] : A\,[C, X]$. By induction hypothesis, there exist B and D such that $\Sigma; \cdot \vdash e : B\,[D]$. Again by induction hypothesis, for every e' such that $\Sigma'; \cdot \vdash e' : B\,[D]$, we have $\Sigma'; \cdot \vdash E_1[e'] : A\,[C, X]$. Conclude the proof by using the typing rule for **catch**.

■

Lemma 70 (Canonical forms)
Let v be a 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 is empty and can be weakened arbitrarily.

Proof: By a straghtforward analysis of the structure of values. ■

Similar to the calculus of exceptions, the catch and throw calculus considers for evaluation expressions that may appear within the scope of a number of **catch** constructs. Since **catch** shrinks the support set of an expression, the subject reduction lemma for catch and throw has to consider primitive reductions over expressions with arbitrary, non-empty, support C.

Lemma 71 (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: By case analysis over possible redexes. We present below some representative cases.

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

■

The Preservation lemma follows the same patter as Subject reduction, and considers expressions with arbitrary support C.

Lemma 72 (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: By evaluation rules, there exists an evaluation context E such that $e = E[r]$, $\Sigma, r \longrightarrow \Sigma', r'$ and $e' = E[r']$. By replacement lemma, there exist B and D such that $\Sigma; \cdot; \cdot \vdash r : B\,[D]$. By subject reduction, Σ' extends Σ, and $\Sigma'; \cdot; \cdot \vdash r' : B\,[D]$. By replacement lemma, $\Sigma'; \cdot \vdash E[r'] : A\,[C]$. Since $e' = E[r']$ this proves the lemma. ■

Lemma 73 (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 case analysis on the structure of redexes.

case $r = \mathbf{catch}_X\ v$, where $X{:}A \in \Sigma$. By reduction rules, $\Sigma, \mathbf{catch}_X\ v \longrightarrow \Sigma, v$. Then we can pick, $\Sigma' = \Sigma$ and $e' = v$.

case $r = \mathbf{catch}_X\ E[\mathbf{throw}_X\ v]$, where $X{:}A \in \Sigma$. By derivation, $\Sigma; \cdot \vdash E[\mathbf{throw}_X\ ke_1] : A\,[C, X]$. By replacement lemma, there exist B and D such that $\Sigma; \cdot \vdash \mathbf{throw}_X\ v : B\,[D]$. By typing rules, it must be $B = A$ and $X \in D$ and $\Sigma; \cdot \vdash v : A\,[D]$. By cannonical forms lemma, v has empty support, and can be arbitrary weakened; in particular $\Sigma; \cdot \vdash v : A\,[C]$. We can thus pick $\Sigma' = \Sigma$ and $e' = v$.

■

The Unique decomposition lemma takes the usual form, as do the Progress and Determinacy lemmas.

Lemma 74 (Unique decomposition)
For every closed 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: Straighforward, by induction on the structure of e. ■

Lemma 75 (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:

■

Lemma 76 (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 $e_2 = \pi(e_1)$.

4.8 Composable continuations

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 **mark** which also assigns a name to it; thus, it is similar to **catch** from the previous section. The jump itself is performed by **recall**, which corresponds to **throw** from the catch-and-throw calculus. The exact syntax of the calculus is defined as follows.

$$\textit{Expressions} \quad e \quad ::= \quad \ldots \mid \textbf{recall}_X\ k.\ e \mid \textbf{mark}_X\ e$$

The differences from the catch-and-throw calculus, however, arise from the following property, which is characteristic for continuation calculi: unlike **throw**, when the construct $\textbf{recall}_X\ k.\ e$ is evaluated, it captures into the variable k the part of the surrounding environment between this **recall** and corresponding **mark** which precedes it; 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 recall to mark points which were defined in the captured and removed part.

The explained operational intuition is formalized by the following definitions of evaluation contexts, redexes and primitive reductions. Because each **recall** is handled by the nearest **mark**, we need to identify within each evaluation context E that **mark** (if any) that is closest to the hole of E. Thus, we define a specific subclass of evaluation contexts that are pure, in the sense that they do not contain a **mark** acting on their hole.

$$\begin{array}{lcll}
\textit{Evaluation contexts} & E & ::= & \ldots \mid \textbf{mark}_X\ E \\
\textit{Pure contexts} & P & ::= & [\,] \mid P\ e_1 \mid v_1\ P \mid \textbf{let box}\ u = P\ \textbf{in}\ e \mid \textbf{choose}\ P \\
\textit{Redexes} & r & ::= & \ldots \mid \textbf{mark}_X\ v \mid \textbf{mark}_X\ P[\textbf{recall}_Y\ k.\ e]
\end{array}$$

$$\begin{array}{rcl}
\Sigma, \textbf{mark}_X\ v & \longrightarrow & \Sigma, v \\
\Sigma, (\textbf{mark}_X\ P[\textbf{recall}_X\ k.\ e]) & \longrightarrow & \Sigma, [K/k]e, \\
 & & \text{where } K = \lambda x.\ \textbf{let box}\ u = x\ \textbf{in box}\ P[u]
\end{array}$$

Example 41 The following expressions (adapted from [DF89, Wad94]), are well formed examples from the calculus of composable continuations.

```
$e_1$ =  1 + mark$_X$ (10 + recall$_X$ f:□$_X$int->□$_X$int.
                let box u = f (f (box 100))
                in
                    mark$_X$ u
                end)

$e_2$ = 1 + mark$_X$ (10 + recall$_X$ f. 100)
```

```
$e_3$ = 1 + mark$_X$ (10 + recall$_X$ f.
            let box u1 = f (box 100)
                box u2 = f (box 1000)
            in
                mark$_X$ (u1 + u2)
            end)
```

The expressions evaluate to 121, 101 and 1121, respectively. In each of these examples, the continuation variable $f : \Box_X\mathbf{int} \rightarrow \Box_X\mathbf{int}$ is bound to $\lambda x.$ **let box** $v = x$ **in box** $(10 + v)$. It captures and internalizes the evaluation environment $(10 + -)$, which is enclosed between **mark** and **recall**. Notice that upon capturing of the environment into f, the delimiting **mark** is *removed* from the reduct. As an illustration of the operational semantics, we show in full the evaluation of e_1.

```
    1 + mark$_X$ (10 + recall$_X$ f.
                        let box u = f (f (box 100))
                        in
                            mark$_X$ u
                        end)

⟼   1 + (let box u = f (f (box 100))
         in
             mark$_X$ u
         end), where f = $\lambda$x. let box v = x in box (10 + v)

⟼   1 + (let box u = f (box (10 + 100))
         in
             mark$_X$ u
         end)

⟼   1 + (let box u = box (10 + (10 + 100))
         in
              mark$_X$ u
         end)

⟼   1 + mark$_X$ (10 + (10 + 100))
⟼   1 + mark$_X$ (10 + 110)
⟼   1 + mark$_X$ 120
⟼   1 + 120
⟼   121
```

■

It is the expression 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 $\perp$, and that is why ordinary continuations cannot be composed in any non-trivial way.

The typing judgment of the calculus for composable continuations is again

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

It establishes that the expression e has type A and may recall the destination points whose names are listed in the support C. The typing rules for composable continuations are presented below.

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

$$\frac{\Sigma; (\Delta, k{:}\Box_{C,X} B \to \Box_{C,X} A) \vdash e : A\,[C] \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{recall}_X\ k.\ e : B\,[C, X]}$$

$$\frac{\Sigma; \Delta \vdash e : A\,[C, X] \quad C \sqsubseteq D \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{mark}_X\ e : A\,[D]}$$

In the case of composable continuations, it is a **recall** to a name that is the notion of effect, and **mark**-ing a name as a destination point is the notion of handling. Therefore, the type system should enable a **recall** to X only if X appears at the support C, placed there by a corresponding **mark**. The situation, however, is a bit more involved. As already mentioned, $\mathbf{recall}_X\ k.\ e$ evaluates e in a changed environment from which the part enclosed between $\mathbf{mark}_X$ and $\mathbf{recall}_X$ has been removed. Correspondingly, e has to be checked against a support from which X has been removed.

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 endow supports with a list-like structure. For example, we allow a **recall** to a certain name only if that name is at the end of the support. This is illustrated in the typing rule for $\mathbf{recall}_X\ k.\ e$, where we demand that X is the *rightmost* name in the support (C, X). If a recall is required to a name which is deeper to the left in C, it can still be done by performing a sequence of nested recalls in a last-in-first-out manner to all the names in between. In this sense, the supports of the calculus of composable continuations may be seen as *stacks*, where the top of the stack is at the rightmost end of the support.

There are yet further important aspects of the typing rule for **recall** 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 **recall** does not matter; we can type **recall** by an arbitrary type B. The domain and the range of the continuation k must match the source and the destination points of the jump, which in this rule have types B and A, respectively. The **recall** appears in the context of a support (C, X) and that is why the domain type of k is $\Box_{C,X} B$. The range type of k is $\Box_{C,X} A$, meaning that the environment captured in k *will not include* the delimiting $\mathbf{mark}_X$.

The typing rule for **mark** is much simpler. The construct $\mathbf{mark}_X\ e$ establishes a destination point X and allows the expression e to recall to X by placing X in 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 42 The program below is a particularly convoluted way of reversing a list, adapted from [DF89].

```
fun reverse (l : intlist) : intlist =
  let name X : intlist
    fun rev' (l : intlist) : □_Xintlist =
      case l
       of nil => box nil
        | (x::xs) =>
         let val y = rev' xs
         in
           box (recall_X c:□_Xintlint -> □_Xintlist.
                mark_X x :: unbox (c y))
         end
    box v = rev' l
  in
    mark_X 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 (recall_X c3.
   mark_X 2 :: unbox c3 (box recall_X c2.
    mark_X 1 :: unbox c2 (box recall_X c1.
     mark_X 0 :: unbox c1 (box nil))))
```

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

There is actually a bit of a leeway in defining the static and dynamic semantics for composable continuations, which has to do with whether the continuation captured by **recall** should include the delimiting **mark** and/or remove it from the environment. The primitive reduction that we have used in our formulation is

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, [K/k]e,$$
$$\text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u]$$

As can be seen, this reduction removes **mark** both from the captured continuation K, and from the evaluation context of the reduced term. But either of the following rules is a possible choice, and we discuss them informally below.

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, [K/k]e, \qquad \text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ (\mathbf{mark}_X\ P[u]) \tag{4.1}$$

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, \mathbf{mark}_X\ [K/k]e, \qquad \text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ P[u] \tag{4.2}$$

$$\Sigma, (\mathbf{mark}_X\ P[\mathbf{recall}_X\ k.\ e]) \longrightarrow \Sigma, \mathbf{mark}_X\ [K/k]e, \qquad \text{where } K = \lambda x.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box}\ (\mathbf{mark}_X\ P[u]) \tag{4.3}$$

The rule (4.1) captures $\mathbf{mark}_X$ into K, but removes it from the evaluation environment of e. The typing rule matching this operational semantics is

$$\frac{\Sigma; (\Delta, k{:}\Box_{C,X} B \to \Box_C A) \vdash e : A\,[C] \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{recall}_X\ k.\ e : B\,[C, X]}$$

Because the mark X is removed from the environment, it becomes impossible for e to recall to X. This is why X does not appear in the support of the premise of this typing rule. Because the mark X is captured into the continuation, the result of applying the continuation does not require a mark for X in its evaluation environment, and so X is also dropped from the range type of k.

The rule (4.2) omits the mark from the continuation K, but leaves it in the evaluation environment of e. The corresponding typing rule leaves X in the support of the premise and in the range type of k.

$$\frac{\Sigma; (\Delta, k{:}\Box_{C,X} B \to \Box_{C,X} A) \vdash e : A\,[C, X] \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{recall}_X\ k.\ e : B\,[C, X]}$$

Because the mark is left in the evaluation environment, it becomes impossible to jump in sequence to names that are further down in the support stack. In this setting, it becomes necessary to consider semantics that allow jumps arbitrarily deep into the support stack. This is very related to the behavior of Felleisen's $\mathcal{F}$ operator [Fel88]. If we label by D the top of the support stack, up to but not including the target mark, then a recall which would jump over the names in D will be typed as follows.

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

Indeed, because the names from D are captured into the continuation, they must be removed from the range type of k. Support D is also removed from the evaluation environment, and hence must be omitted from the support of the premise.

The rule (4.3) leaves the mark into both the continuation K and the evaluation environment of e, and the typing rule for it is thus

$$\frac{\Sigma; (\Delta, k{:}\Box_{C,X} B \to \Box_C A) \vdash e : A\,[C, X] \quad X{:}A \in \Sigma}{\Sigma; \Delta \vdash \mathbf{recall}_X\ k.\ e : B\,[C, X]}$$

This choice of semantics corresponds to Danvy and Filinski's **shift** operator [DF89, DF90]. If only jumps to the last established mark are allowed (as is the case in [DF89, DF90]), then it may be possible to simplify the typing rules so that the modal types only record the top-most name in the support.

Our choice of operational semantics for composable continuations is similar to the one for the **set/cupto** operators of Gunter, Rémy and Riecke [GRR95]. We have decided on this choice of operational semantics for composable continuations because all the other choices can be encoded within it. Obviously, if the mark is discarded during reduction, it can always be placed back; if it is retained, it can never be eliminated. We do not know if the other operational semantics can match this expressiveness.

Example 43 Composable continuations have been used to conveniently express "nondeterministic computation"; that is, computation which can return many results [DF89, DF90]. The following example is a program for finding all the partitions of a natural number n, i.e. all the lists of natural numbers that add up to n. The main function `partition` is very effectively phrased in terms of a primitive function `choice`. The idea is to use `choice` to non-deterministically pick a number between 1 and n, and not worry about backtracking and exploring other options. Backtracking is automatically handled by `choice`.

```
fun partition n =
  if n = 0 then box (nil)
  else
      box (let val i = unbox (choice n)
               box l = partition (n - i)
           in
               (i::l)
           end)
```

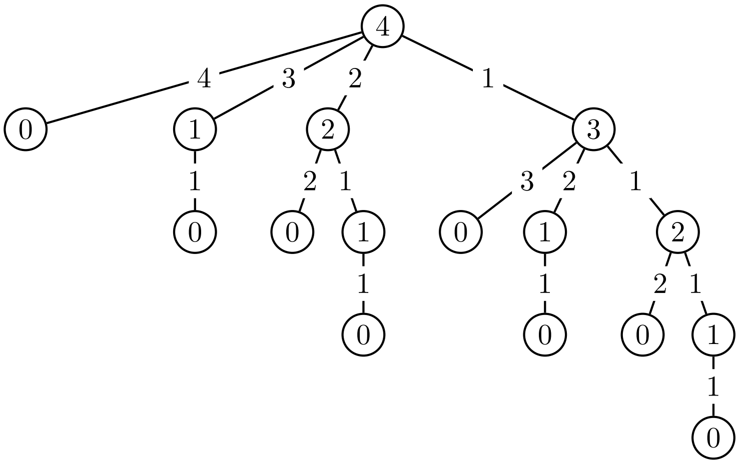
The important point is that `choice` itself can be implemented using composable continuations. The way `choice` is implemented will determine the ordering in which `partition` considers the candidate lists for partitioning n.

The process of generating partitions for n may be seen as a traversal of a tree with labeled nodes and edges – a partition tree. Paths in the partition tree emanating from a node labeled by n represent the partitions of n. An inductive definition of the partition tree for n is given as follows:

(i) if $n = 0$, then the tree consists of a single node labeled 0.

(ii) if $n > 0$, then the root of the tree is labeled with n, and edges labeled with $n, n-1, \ldots, 1$ connect the root to partition trees for $0, 1, \ldots, n-1$, respectively.

An example partition tree for $n = 4$ is presented below.



Of course, just as with any tree, various traversal strategies may be employed to generate the partitions for n. For example, a *depth-first strategy* may employ a *stack* k to store the nodes that remain to be traversed. After putting the root node on the stack, the depth-first strategy repeats the following algorithm: remove the top node t from k, and *expand* it, i.e. determine all the children of t (if any), and put them onto the *top* of k; if k is empty, then exit.

On the other hand, a *breadth-first strategy* may employ a *queue* k to store the nodes that remain to be traversed. After putting the root node on the queue, the breadth-first strategy repeats the following: remove the top node t from k, and *expand* it, i.e. determine all the children of t (if any), and put them at the *bottom* of k; if k is empty, then exit.

In our implementation of the partition algorithm, the partition tree for n is never explicitly built, but is implicitly described by the execution of the `partition` function. For example, we present below a version of `choice` which facilitates a depth-first traversal of the tree. It assumes a name X of unit type.

```
(* choice : int -> □_Xint *)
fun choice n =
    box (recall_X t : □_Xint -> □_Xunit.
        let fun loop (s : int) : unit =
           if s = 0 then ()
           else
               let box u = t (box s)
               in
                   (mark_X u);
                   loop (s - 1)
               end
        in
           loop (n)
        end)
```

The program works by viewing the current global program continuation as an implicit stack k of nodes to be expanded in order. Each node has its own composable continuation, all of which *compose* to create k. The function `choice` simply captures into t the composable continuation for the first node in the sequence. The captured node is *removed*, and t is applied to generate all of its children – one child for each possible value of the variable s. The children nodes are added in place of the parent node at the top of the global program continuation k. Because the new nodes are added to the beginning, they will be the the first to expand in the subsequent execution. As a consequence, this implementation of choice uses *depth-first traversal strategy.*

With this version of choice, partition has the type int -> $\Box_X$intlist. To compute the partitions for 4, we run mark$_X$ print (unbox partition 4). The result consists of the lists [4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]. Because depth-first traversal is employed, the lists are sorted in lexicographical order.

In our calculus, it is also possible to implement choice so that it facilitates breadth-first strategy. When generating the children of some node, we only need to attach them at the end, rather than at the beginning of the queue k that the global continuation represents. One possible breadth-first implementation of choice is given below.

```
(* choice : int -> $\Box_{Y,X}$int *)
fun choice n =
    box (recall$_X$ t : $\Box_{Y,X}$int -> $\Box_{Y,X}$unit.
              recall$_Y$ k : $\Box_Y$unit -> $\Box_Y$unit.
           mark$_Y$
              let fun loop (s : int) : $\Box_Y$unit =
                      if s = 0 then box ()
                      else
                          let box u = t (box s)
                              box u' = loop (s - 1)
                          in
                              box (mark$_X$ u; u')
                          end
                  box v = k (box mark$_X$ ())
                  box v' = loop n
              in
                  v; v'
              end)
```

How does this function work? First, we must assume that the queue is marked by a new name Y of unit type, so that it can be captured into a continuation itself. The function choice captures the topmost node into t, and then captures the rest of the queue into k. It is important that the continuation k will not contain the delimiting **mark**$_Y$. Then choice expands the topmost node t, adds the obtained children nodes to the bottom of k, and puts **mark**$_Y$ back, so that its scope includes the children nodes. Again, it is crucial for this application that the captured continuations omit the target mark (unlike, for example, in the calculi from [DF89, DF90]), as this mark will get in the way of adding new nodes at the bottom of k.

With this implementation of choice, the appropriate type for the function partition is int->$\Box_{Y,X}$intlist. To compute the partitions for 4, we run

```
mark$_Y$ mark$_X$ print (unbox partition 4)
```

to obtain the lists [4], [3, 1], [2, 2], [1, 3], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]. Because we used breadth-first traversal strategy, we first explored all the partitions of size 1, then all the partitions of size 2, etc. Thus, the lists will be sorted first by size, rather than lexicographically, as was the case with depth-first traversal.

■

Structural properties and type soundness

Lemma 77 (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 78 (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 79 (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 $v =$ **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 80 (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 81 (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 82 (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 83 (Unique decomposition)
For every expression e, either:

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

Lemma 84 (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 85 (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 $e_2 = \pi(e_1)$.

4.9 Notes

Related work on type-and-effect systems

Integrating effects into typed 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 a representative example of these *type-and-effect systems*, we simply list the works of Gifford, Lucassen, Jouvelot, Talpin and Tofte [GL86, LG88, JG89, JG91, TJ92, TJ94, TT97]. The approach usually taken by type-and-effect systems is to extend the language with a type of *effectful functions* $A \overset{C}{\to} B$. Here, C is a set of effects that the evaluation of the function body may cause.

Coming from the side of logic and type theory, type-and-effect systems are directly related to monads. A monad $\bigcirc$ is a type constructor which is used to differentiate between values and effectful computations. In monadic calculi, the type $\bigcirc A$ is ascribed to expressions which may evaluate to a value of type A, but may cause some effect in the course of evaluation. Monads were invented for use in denotational semantics by Moggi [Mog89, Mog91], and were later adopted for functional programming by Wadler [Wad92, Wad95].

The connection between monads and effect systems is described by Wadler in [Wad98]. Briefly, the effectful function type $A \overset{C}{\to} B$ in the type-and-effect systems corresponds to the monadic type $A \to \bigcirc_C B$. One advantage of monads over type-and-effect systems is that monads *encapsulate* effects, so that effects can be added to the language in a *modular* way, without changing the already existing language constructs. This is opposite to the type-and-effect systems, which require that the function types $A \to B$ be extended into effectful function types $A \overset{C}{\to} B$.

The modal effect calculi described in this dissertation also encapsulate effects and add them to the language in a modular way, without changing the underlying function types. However, the modal framework allows more than one type operator for effects, and thus allows more precise distinctions between different effectful computations.

Related work on dynamic binding

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.

Related work on exceptions

A treatment of exceptions in Haskell is considered by Peyton Jones at al. in [PRH$^+$99]. It is interesting that this paper does not the exception monad to add exceptions to the underlying language, but rather implements *imprecise* exceptions. The idea is to keep track of all the possible exceptions that a given expression may raise, and then for handling purposes return only one of the possibilities, non-deterministically.

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 we refer to the following theoretical works on catch-and-throw [Nak92, Kam00a, KS02]. These calculi also lack the type constructor for exceptional computations, and thus have to restrict the way exceptions are introduced, propagated and handled.

Related work on composable continuations

Composable continuations were probably first considered by Felleisen in [Fel88], in an untyped setting and with recalling (or shifting) to only the nearest mark (or reset, or prompt). A generalization to a whole family of control operators for recalling, each of which is indexed by a numeral proscribing how many closest marks 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 recalls.

In a typed setting, Danvy and Filinski in [DF89] develop a calculus for composable continuations with a single recall operator. The marks are not labeled. In the Appendix C, they also briefly discuss the idea which we have employed here: upon capturing, remove the marks from the environment, so that jumps can be made to the marks further down in the context stack. Danvy and Filinski further relate composable continuations to the CPS transformation in [DF90, DF92]. These papers also contain extensive commentary on the related work regarding composable continuations. Gunther, Rémi and Riecke in [GRR95] develop a calculus whose operational semantics is very similar to the one used for the calculus of composable continuation in this dissertation. In particular, this calculus removes the delimiting mark upon capture, from both the environment and the reduct. Most recently, Kameyama in [Kam00a, Kam00b] works with labels instead of numerals to provide a hierarchy of recall operators. The mentioned typed calculi lack a type constructor for effectful computations, so they must impose restrictions on expressiveness and type safety in order to avoid the extrusion of effect scope.

Logical content of composable continuations is studied by Murthy in [Mur92]. This paper develops a type system for composable continuation with a hierarchy of recall operators, which is based on monads indexed by sets of types, but has to restrict the marks 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 recall operator, and with a hierarchy of recall operators, and presents them in terms of indexed monads. All these calculi are characterized by the serialization of effects inherent in the monadic programming.

Monadic reflection and reification

One of the main features of the monadic calculi is the programming style in which the program itself must specify a total ordering on the compitational effects. But sometimes, most notably in the case of benign effects, effectful computations may be independent and therefore may be evaluated out of order.

This problem with excessive serialization of monadic programs has been addressed previously by Filinski, using monadic reflection and reification [Fil94, Fil96, Fil99]. Reflection and reification are translations between an effectful source language and a monadic meta language. The effectful source language provides the syntax for programming (which avoids the burden of excessive serialization), while the monadic meta language defines the semantics for the program. The modal approach to effects addresses the same problem of excessive serialization, but it does so directly, using only natural deduction, and without any translations.

A futher advantage of modal calculi over monads concerns efficiency. Monads may sometimes impose efficiency penalties (depending on the evaluation strategy), which come in the form of tagging and run-time tag checking of monadic values. This was particularly evident in the case of exceptions (Section 4.6). Monadic reflection and reification do not resolve this issue; as concluded in [Fil94], reflection and reification still incur the operational penalties of tag checking. In contrast, tagging is *not* required in the modal calculus for exceptions.

Kripke semantics for the lax logic

As described in Section 4.1, the identification of truth and necessity in CS4 leads to the formulation of lax logic, in the sense that the modal operator $\Diamond$ translates into the lax operator $\bigcirc$. This identification is achieved by extending the CS4 logic with the axiom $A \rightarrow \Box A$.

In the Kripke semantics of CS4, truth and necessity are identified if the Kripke model satisfies the following *monotonicity* property:

for every world w and proposition A, if $w \models A$ and $w \rightarrow w'$ then $w' \models A$.

Indeed, in this class of models, if A is true at the current world, then A is true at all accessible worlds, and is therefore necessary. Then, as established by Alechina at al. in [AMdPR01], a Kripke model for propositional lax logic consists of a Kripke model for CS4 that satisfies the above monotonicity property.

Logical meaning of dynamic binding and exceptions

In this note we describe a possible logical interpretation for the calculi of dynamic binding and exceptions (Sections 4.4 and 4.6). The main idea is to involve two levels of interpretation. The judgment from the calculi of dynamic binding and for exceptions form the *object level.* The *meta* level, or the meta logic, defines the reasoning *about* the derivability in the calculi from the object level. The modal operators may be seen as internalizing properties of the meta logic for reasoning about *categorical* derivations from the object level. This note will necessarily be very informal, and making the presented intuition precise is left for future work.

The types from the object level should be contrasted to *meta propositions*, which belong to the meta logic. For example, the atomic propositions of this meta logic are of the form A *true* where A is an arbitrary the object types. For each object type operator, the meta logic ought to contain a corresponding connective on propositions, and appropriately relate the two. For example, in the meta logic we have

$$A\ \mathit{true} \supset B\ \mathit{true}$$

if and only if we may also derive $A \rightarrow B$ *true*.

Memory locations $X_1{:}A_1, \ldots, X_n{:}A_n$ may be treated as labels for the meta propositions A_1 *true*, … , A_n *true*. Then, we require that

$$\Box_{X_1,\ldots,X_n} A\ \mathit{true}$$

if and only if the conclusion A *true* may be derived in the meta logic from the hypotheses $X_1, \ldots,$ X_n. In the calculus of dynamic binding, the reflection principle is realized by means of explicit substitutions, and it simply allows that meta logic derivations to be translated into the object logic.

The meta logic for dynamic binding pretty closely follows the object calculus, in the sense that the meta logic only contains the connectives corresponding to the object type operators. But this need not be the case. For example, the meta logic for exceptions should contain a propositional operator $\neg$ for negation, while negation is not a type operator on the object level.

Exceptions $X_1{:}A_1, \ldots, X_n{:}A_n$ may be considered as labels for the meta logical propositions $\neg(A_1\ \mathit{true}), \ldots, \neg(A_n\ \mathit{true})$. Then in the meta logic, we require that

$$\Box_{X_1,\ldots,X_n} A\ \mathit{true}$$

if and only if the conclusion $A\ \mathit{true}$ may be derived in the meta logic from the hypotheses $X_1, \ldots, X_n$.

For example, let us assume that $A\ \mathit{true}$ can be proved in the meta logic, and let the name X be a label for the proposition $\neg(A\ \mathit{true})$. Then we can use X to reason by contradiction and prove $B\ \mathit{true}$, where B is an arbirary object type. In other words, given $A\ \mathit{true}$, we can derive $\Box_X B\ \mathit{true}$. This reasoning directly corresponds to the following derivation in the calculus of exceptions:

if $\vdash e : A$ then $\vdash \mathbf{box}\ (\mathbf{raise}_X\ e) : \Box_X B$.

In the calculus of exceptions, the reflection principle corresponds to exception handling, and it allows the meta logical derivations to be coerced into object logic. Let us assume that we are given the meta logic propositions $\Box_X B\ \mathit{true}$ and $A\ \mathit{true} \supset B\ \mathit{true}$. Because $\Box_X B\ \mathit{true}$ corresponds to $\neg(A\ \mathit{true}) \supset B\ \mathit{true}$, we can employ the law of excluded middle and derive $B\ \mathit{true}$. This reasoning directly corresponds to the following derivation in the calculus of excluded middle.

if $\vdash e : \Box_X B$ and $\vdash \langle\Theta\rangle : [X] \overset{B}{\Rightarrow} [\,]$, then $(\mathbf{unbox}\ e)\ \mathbf{handle}\ \Theta : B$.

From the standpoint of Kripke semantics, it seems a plausible that the indexed modalities may be introduced by the following redefinition of the $\models$ relation.

1. $w \models \Box_C A$ iff for all $w' \sqsupseteq w$ and $u' \leftarrow w'$, $u' \models C$ implies $u' \models A$.
2. $w \models \Diamond_C A$ iff for all $w' \sqsupseteq w$ there exists $u' \leftarrow w'$ such that $u' \models C$ and $u' \models A$

In this definition, C is the set of names $C = \{X_1, \ldots, X_n\}$, where the name X_i has the type A_i. In the case of dynamic binding, we set $w \models C$ if and only $w \models A_1, \ldots, w \models A_n$. On the other hand, in the case of exceptions we set $w \models C$ if and only if $w \not\models A_1, \ldots, w \not\models A_n$.

Recursive names and future work on dynamic binding and state

It is a well known property of functional languages, that in the presence of state and higher-order functions, recursion becomes admissible. For example, we can define a recursive function `fact:int->int` for computing factorials, without explicitly using the constructs for recursion. Below is an example in ML-like notation.

```
let val fact : int -> int =
    let val F = ref  (λx. x) (* a dummy value *)
        val g = λx. if x = 0 then 1
                    else x * (!F)(x - 1)
    in
        (F := g); g
    end
```

The admissibility of recursion is a slightly concerning property of stateful computations, because it shows that state destroys the connection with logic which is otherwise enjoyed by the pure λ-calculus.

We may attempt to translate the above program into the calculus of dynamic binding from Sections 4.4, by declaring `F` as a name of type `int -> int`. This translation, however, will *not* result in a well-typed program. Indeed, the function `g` must be typed as `int ->` $\Box_F$`int`, because `g` references `F` in its body. But then, it is not possible to assign `g` to `F` because of a type mismatch. The type of `F` cannot simply be `int -> int`, but rather must be `int ->` $\Box_F$`int`. When the type of `F` depends on `F` itself, as it is the case here, we say that F is a *recursive* name. With an explicit construct for recursive names, the recursive factorial function can be defined in the calculus of dynamic binding.

```
let val fact : int -> int
    let recname F : int -> □_F int (* no need for a dummy value *)
        val g = λx. if x = 0 then box 1
                    else box (x * unbox (F (x - 1)))
    in
        λn. <F -> g> unbox (g n)
    end
```

Incidentally, the fact that recursion does not seem possible unless enabled by a separate language construct, provides a compeling evidence that the modal calculi for dynamic binding and state from Sections 4.4 and 4.5 are actually *strongly normalizing.* This is another conjecture left for future work.

Many other features, in addition to recursive names, need to be considered if the modal calculus is to be extended into a full-fledged language with state. It seems important, for example, to consier first-class names (as suggested in Section 2.3), support polymorphism (Section 3.3), explicit substitutions of variable names, etc. The design space is rather large, and each of these extensions may be interesting in its own right.

Related work on the comonadic formulation of effects

In category theory, the operator $\Box$ of CS4 modal logic is usually modeled by a comonad, and the fact that comonads may represent effects have previously been noticed by Brookes and Geva [BG92], Kieburtz [Kie99] and Pardo [Par00].

It is interesting that Brookes and Geva consider a particular family of comonads, called *computational comonads.* The comonad $(\Box, \varepsilon, \delta)$ is computational, if in addition to the comonadic laws listed in Section 1.3 it admits a natural transformation $\gamma : A \to \Box A$, such that

- $\varepsilon_A \circ \gamma_A = id_A$
- $\delta_A \circ \gamma_A = \gamma_{\Box A} \circ \gamma_A$

The naturality of γ guarantees that for every morphism $f : A \to B$,

- $\Box f \circ \gamma_A = \gamma_B \circ f$

As evident from its type, the natural transformation γ corresponds to the extension of the modal CS4 calculus with the axiom $A \to \Box A$, and thus provides a way to coerce values into computations.

Kieburtz in [Kie99] proposes comonads for those effectful computations that may depend on the run-time environment, but do not change it. Needless to say, this is exactly how the the operator

$\Box$ in used in this dissertation. It is interesting that the comonads in [Kie99] are not computational in the sense defined by Brookes and Geva, and do not readily admit the coercion of values into computations.

Neither of the cited papers on comonads make the connection with effect handling and with modal logic.

Modal types for diverging computations

Consider a purely functional language with a fixpoint construct, defined by the following typing rule and operational semantics.

$$\frac{\Delta, x{:}A \vdash e : A}{\Delta \vdash \mathbf{fix}\ x{:}A.\ e : A}$$

$$\mathbf{fix}\ x{:}A.\ e \quad \longmapsto \quad [\mathbf{fix}\ x{:}A.\ e/x]e$$

Expressions in this language either evaluate to a value, or never terminate. Such expressions are *partial*, because they may diverge. A typical example is the expression **fix** $x{:}A.\ x$, which reduces to itself. Notice however, that the evaluation of a non-terminating expression does not perform any changes to the run-time environment. Depending on the operational semantics of the language, divergence may prevent some expressions from being evaluated, but it does not influence the outcome of those evaluations that do take place. Divergence is a *benign* effect.

In fact, divergence is such a simple effect, that non-terminating computations do not even depend on the run-time environment; if the computation does not terminate in one environment, it will not terminate in any other environment either. This is in fact one of the reasons that divergence is frequently not even considered an effect.

However, if we do want to treat diverging computations as effectful, the benign nature of divergence suggests that we should use the type system for benign effects (Section 4.3). How? The idea comes from the operational semantics. Observe that the reduction of **fix** $x.\ e$ substitutes the variable x by **fix** $x.\ e$. The fact that x is substituted by an effectful computation, should be made explicit in the variable context.

With that in mind, we introduce a name N to serve as a marker for non-termination. If an expression is possibly diverging, its support will contain the name N. In fact, because we assumed that our language is pure except for divergence, our supports will either be empty, or contain the single name N. Given the name N, we may now redefine the typing rule for **fix**, as follows.

$$\frac{\Delta, x{:}A\,[N] \vdash e : A\,[C]}{\Delta \vdash \mathbf{fix}\ x{:}A.\ e : A\,[C]}$$

Notice that the support set C of the expression **fix** $x.\ e$ may equal the singleton $\{N\}$, but may also be empty, depending on how x is used in e. Of course, if **fix** $x.\ e$ has empty support, than by the support weakening principle, it may be considered as having support $\{N\}$ as well. As a consequence, the operational semantics that substitutes $x : A\,[N]$ by **fix** $x.\ e$ obeys the prescribed supports, and will be type safe.

It is interesting that non-termination does not admit any obvious notions of handling, by which we could remove the name N from the support of a possibly non-terminating computation, and therefore restore the purity of such a computation. In fact, it may be appropriate to view non-termination as an effect that is handled by some entity outside of the language (e.g. the operating system). Of course, then we should allow that expressions with non-empty support be evaluated. This contrasts Chapter 4, where we only evaluate expressions with empty support.

To illustrate the above ideas, we present the code for a factorial function which uses fix-points and is therefore conservatively labeled as non-terminating.

```
- fix fact : int -> □_N int.
        λn:int. if n = 0 then box 1
        else box (n * unbox (fact (n - 1)));

val fact = [fn] : int -> □_N int

- unbox (fact 2) + unbox (fact 3);
val it = 8 : int
```

Notice that the fix-point expression may not be typed simply as `int -> int`, but must be given a more complicated type `int ->` $\Box_N$`int`. Indeed, the recursive reference to `fact` in the λ-abstraction must be boxed. Otherwise, the body of the λ-abstraction would have had non-empty support, which is not allowed by the type system for benign effects (Section 4.3). In this example, the function `fact` has empty support, but the result 8 is obtained with support N. We may suppress this information, however, because expressions with both empty and non-empty supports are admitted for evaluation.

Bibliography

[AM03] D. Ancona and E. Moggi. A calculus for symbolic names management. Personal communication, October 2003.

[AMdPR01] Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. Categorical and Kripke semantics for Constructive S4 modal logic. In Laurent Fribourg, editor, *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 292–307, Paris, 2001. Springer.

[AS95] Giuseppe Attardi and Maria Simi. A formalization of viewpoints. *Fundamenta Informaticae*, 23(3):149–173, 1995.

[BBdP98] P. N. Benton, G. M. Bierman, and V.C.V de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, March 1998.

[BdP00] G. M. Bierman and V. C. V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416, 2000.

[BES98] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 1998.

[BG92] Stephen Brookes and Shai Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Application of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Notes*, pages 1–44. Cambridge University Press, Cambridge, 1992.

[BHM02] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luis Pinto, and Joao Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2002.

[Bjø99] Nikolaj Bjørner. Type checking meta programs. In *Workshop on Logical Frameworks and Meta-languages*, Paris, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Symposium on Logic in Computer Science, LICS'91*, pages 203–211, Amsterdam, 1991.

[CMS01] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2001. To appear.

[CMT00] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2000.

[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.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *Symposium on Principles of Programming Languages, POPL'96*, pages 242–257, St. Petersburg Beach, Florida, 1996.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In *Symposium on Logic in Computer Science, LICS'96*, pages 184–195, New Brunswick, New Jersey, 1996.

[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.

[DF92] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[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.

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

[dP03] Valeria de Paiva. Natural deduction and context as (constructive) modality. In Patrick Blackburn, Chiara Ghidini, Roy M. Turner, and Fausto Giunchiglia, editors, *Modelling and Using Context*, volume 2680 of *Lecture Notes in Artificial Inteligence*, pages 116–129. Springer, 2003.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 1997.

[Ers77] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[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.

[Fio02] Marcelo Fiore. Semantic analysis of normalization by evaluation for typed lambda calculus. In *International Conference on Principles and Practice of Declarative Programming, PPDP'02*, pages 26–37, Pittsburgh, Pennsylvania, 2002.

[FM97] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[FM99] Melvin Fitting and Richard L. Mendelsohn. *First-Order Modal Logic*. Kluwer, 1999.

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Symposium on Logic in Computer Science, LICS'99*, pages 193–202, Trento, Italy, 1999.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

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

[Gir86] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 1995.

[GJ97] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

[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.

[GRR95] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *International Conference on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 12–23, La Jolla, California, 1995.

[Har99] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–470, 1999.

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

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Symposium on Logic in Computer Science, LICS'99*, pages 204–213, Trento, Italy, 1999.

[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.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting techniques and applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 1985.

[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.

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

[Kob97] Satoshi Kobayashi. Monad as modality. *Theoretical Computer Science*, 175(1):29–74, 1997.

[Kri63] Saul Kripke. Semantic analysis of modal logic i. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kri80] Saul A. Kripke. *Naming and Necessity.* Harvard University Press, 1980.

[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.

[LP95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[McC93] John McCarthy. Notes on formalizing context. In *International Joint Conference on Artificial Intelligence, IJCAI'93*, pages 555–560, Chambery, France, 1993.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures. In *Proceedings of the First Esprit BRA Workshop on Logical Frameworks*, pages 323–335, Antibes, France, 1990.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 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.

[Mon63] Richard Montague. Syntactical treatment of modalities, with corollaries on reflexion principles and finite axiomatizability. *Acta Philosophica Fennica*, 16:153–167, 1963.

[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.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming, ESOP'99*, pages 193–207, Amsterdam, 1999.

[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.

[NPP03] Aleksandar Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta variables. In *Proceedings of MERλIN'03*, Uppsala, Sweden, 2003.

[NT03] Michael Florentin Nielsen and Walid Taha. Environment classifiers. In *Symposium on Principles of Programming Languages, POPL'03*, pages 26–37, New Orleans, Louisiana, 2003.

[Ode94] Martin Odersky. A functional theory of local names. In *Symposium on Principles of Programming Languages, POPL'94*, pages 48–59, Portland, Oregon, 1994.

[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.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Conference on Programming Language Design and Implementation, PLDI'88*, pages 199–208, Atlanta, Georgia, 1988.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, April 2003.

[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.

[Pra65] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study.* Number 3 in Stockholm Studies in Philosophy. Almquist and Wiskell, 1965.

[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.

[PS93] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

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

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier, 1983.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie Mellon University, 2000.

[Sco70] Dana Scott. Advice on modal logic. In Karel Lambert, editor, *Philosophical Problems in Logic*, pages 143–173. Dordrecht: Reidel, 1970.

[Sco79] Dana Scott. Identity and existence in intuitionistic logic. In Michael Fourman, Chris Mulvey, and Dana Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696. Springer, 1979.

[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.

[Sim94] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

[Smo85] C. Smoryński. *Self-Reference and Modal Logic.* Springer, 1985.

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

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'00*, pages 34–43, Boston, Massachusetts, 2000.

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

[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.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 203–217, Amsterdam, 1997.

[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.