Abstract types in distributed systems

A *partial* translation of my Ph.D. dissertation

Gilles Peskine

12 June 2008

Abstract

Consider a network of nodes running ML programs that exchange data. How can data which has an abstract type on one node be accepted on another node? A safe approach is to treat abstract types as distinct whenever they are defined on different nodes. However this is too restrictive in practice, for example in the common case where an abstract type enforces a semantic invariant.

The main contributions of this thesis are threefold: I define a notion of hash of an abstract type, whereby abstract types that have the same hash are deemed compatible; I give an operational semantics for a module system that preserves types, including abstract types; I also propose a new, more general module system that is well-suited to distributed applications.

The hash of an abstract type must reflect its intended semantics, which is often not apparent from the program's code. In practice, two modules have the same hash if they have the same code. Compound modules are compatible when they are built from compatible components.

Existing operational semantics for ML modules lose information as they erase abstraction boundaries. I use coloured brackets to track the visibility of abstract types. I study two calculi equipped with brackets, a simply-typed lambda-calculus and a rich ML module calculus.

I use singleton signatures to keep track of not only type but also code sharing, so that module equivalence is defined at arbitrary signatures. A simple effect system limits type constraint to a statically checkable fragment, while permitting both applicative and generative functors. I discuss static and dynamic forms of module sealing.

Contents

Introduction

Objective

The objective of the present dissertaion is to extend an ML-like language to adapt it to distributed systems. Specifically, we are interested in the requirements that the distributed nature of the environment imposes on the type system — we will not concern ourselves with other aspects such as concurrent execution and fault tolerance.

Consider two machines A and B, each executing a program. At some point in time, A and B start exchanging data. The central question of this dissertation is, how can we make sure that A and B agree on the semantics of the exchanged data?

A network link carries sequences of bits (usually arranged in bytes). When A sends data to B, the data is encoded as a sequence of bits. This operation is known as **marshaling** (the words **pickling** and **serialisation** are synonyms). Upon reception of the bit sequence, B must perform the opposite operation (known as unmarshaling, unpickling or deserialisation). Many languages provide a standard representation of data as strings: s-expressions in Lisp, `Marshal` library in Objective Caml [L$^+$], `Pickle` library in Standard ML [PSL], `Serializable` interface in Java [Sun]... Several standards (ASN.1, XML) specify language-independent string encodings of data for communication. While the exact set of supported data shapes varies greatly, serialisation libraries and data representation standards usually specify at least how to encodes numbers (n-bit integers, little- or big-endian, decimal notation...), strings (character sets and encodings: ASCII, Unicode, ...), sequences of such...

Marshaling data entails transforming it to an unambiguous sequence of bits. Unmarshaling consists of two parts: the bit sequence must be transformed back into a workable representation of the data, and one must verify that the resulting data has the expected type or shape. For example, if the program running on B expects a number, and the program on machine A sends the string `"foo"`, the error must be detected. The usual approach in ML-like languages is to detect such errors as soon as possible, which is as soon as the program has been written (during the type-checking phase of compilation). It seems natural in an ML-like language to express the unmarshal-time check as a type constraint; but how can this constraint be imposed?

According to the ML approach, the error must be detected when compiling the program on A or B. Thus the program on A would declare a communication channel of type `string` (on which one may send `"foo"`), and the program on B would declare a channel of type `int` (on which only numbers may be received). But this only delays the problem, since the fact that A and B disagree on the type of their shared communication channel cannot be detected until A and B start communicating.

This observation leads us to desire a run-time type-check, specifically a type-check *when establishing a communication channel between programs that have not yet communicated.* (Once the programs have communicated, such a check is no longer necessary, since the programs may now have agreed on the types of future communications. For example JoCaml [MM01] has a static type system [FLMR97]; however two independently started JoCaml programs that wish to communicate must get a shared channel via a "name server", which is currently not well-typed.)

Although ML is designed to be statically typed, and most compilers erase types to save memory during execution, there are ways of checking whether a value has a certain type at run-time. However existing systems do not manage abstract types correctly, allowing only types with a predefined structure to be shared between separate programs.

One solution is to forbid values of abstract types to be marshaled. Another is to require the author of the abstract type to provide marshaling and unmarshaling functions. This however does not solve our problem: a serialisation format can usually be deduced automatically from the representation of the type, but this does not fully solve the problem of checking whether the type of the sent data is the type that is expected at the point of reception. Herein lies the gist of the matter: when are two abstract types the same?

There are two main intuitions to the nature of an abstract type. One point of view states that an abstract type in *hidden*. It has an *implementation*, which is a "concrete" type (the implementation may make use of other abstract types, but these can be traced through in turn all the way to built-in types). Hence an abstract type is a concrete type — but we do not know which. Another view is that an abstract type is a *new*, *fresh* type, distinct form any other type (in particular it is distinct from any concrete type, and it is distinct from its implementation type, in that one may not convert freely between the two).

When are two hidden types the same? One prerequisite that comes to mind immediately is that the implementation types must be the same. But this condition is neither necessary nor sufficient. One may wish to consider two hidden types as the same when their implementations have identical behaviour, even if their code differs. Conversely, just because the implementations match exactly does not mean that the types can be matched freely — for example a `Euro` type and a `Dollar` type may have the same implementation, yet should definitely not be compatible. Type abstraction can play multiple rôles, and usages may differ in terms of ideal degree of compatibility.

When are two fresh types the same? The simplest answer is "when they were created in the same operation". This approach has often been refined by proposing language constructs that may or may not create fresh types. In ML, control of type freshness is given to the *module* language, which we shall therefore study.

General outline of the dissertation

Chapter I presents the basic concepts upon which the dissertaion is based. We first study the notion of abstraction, its uses and how to express it. In ML-like languages, abstraction arises via the module language, and we highlight some points of its rich history. We also study how to add dynamic type-checking to a statically typed language.

Chapter II develops a notion of *imprint*. The imprint of a software component identifies the abstraction that it provides. We examine many sample programs in order to decide how much compatibility is desired in various conditions, and we discuss how to compute imprints so that two components have the same imprint if and only if they are supposed to be compatible.

Chapter III presents a simple first language equipped with imprints, the HAT language. This language extends the simply-typed lambda-calculus with simple modules. We keep track of abstraction domains throughout program execution using *coloured brackets*. The language also includes dynamically typed communication primitives that use imprints to test the equality of abstract types.

Chapter IV describes a new *module system* for ML which is suitable for distributed programs, the TOPHAT language. This language includes central concepts in module calculi, such as functors and sealing. Singleton types with no signature restriction allow for the expression of code equalities as well as type equalities, generalising the usual notion of type sharing. We show how to express different kinds of sealing, depending on the expected level of generativity. Like HAT, TOPHAT uses imprints to perform run-time type equality tests involving abstract types, and coloured brackets to preserve abstraction barriers during program execution.

We conclude with a survey of related work and future work perspectives.

Appendix A summarises the formal definition of the TOPHAT language introduced in chapter IV. Appendix B contains a proof of the soundness of TOPHAT.

A note about code snippets

We usually present code snippets in Objective Caml syntax. We do not expect the reader to know the fine points of the language, and will in particular explain any subtlety concerning the semantics of modules. When features that Objective Caml does not have are illustrated, we use Objective Caml-like syntax augmented as desired and describe the intended semantics in the text. Readers used to Standard ML may wish to consult a correspondance table between the two dialects [Ros].

Chapter IV

TOPHAT: a module calculus suited to distributed environments

IV.1 Introduction

Objectives

The purpose of this chapter is to present a module system that combines the usual features found in ML-like languages with a flexible management of abstract types that, as in HAT, is suited to distributed programs.

This module system is described as a typed lambda-calculus, for which we provide typing rules and a type-preserving small-step operational semantics. The system described herein purports to be a theoretic model, not a full-blown programming language, although our design choices will be motivated by practical concerns. As such, it lacks some practical features that might be thought of as syntactic sugar. We will also for the most part delay implementation considerations until section V.3.3.

Existing module systems already span a wide range of features and style With respect to expressivity, our aim is to cover the features that we think are fundamental to our specific objective of coping well with abstract types in a distributed environment. Style-wise, we have tried to provide a compositional approach, where each aspect of the language is embodied in a separate language construct that can be easily understood on its own.

Vocabulary and notations

We will endeavour to distinguish between the words "expression", meaning a specific language category (expressions typically have types, and can be evaluated), and "term", which denotes an element of any syntactic category (expression, type, module, environment, etc.). We will usually denote by $\aleph$ or $\beth$ a term of unknown syntactic category.

Let $\aleph$ be any term, x a variable and E an expression. We will write the substitution of E for x in $\aleph$ as $\{x{\leftarrow}E\}\aleph$. Similarly, we will write $\{X{\leftarrow}M\}\aleph$ for the substitution of module expression M for module variable X in $\aleph$.

Outline

We will present the TOPHAT language incrementally. Each of the following five steps refines or extends the previous language.

$\mathcal{B}$ We will start with a basic module system, including only module-building constructs that would be sufficient for our purpose in the absence of types. This system is simple but lacks expressivity as far as types are concerned.

$\mathcal{S}$ We will add singleton types (which generalise singleton kinds) to the basic system, in order to keep track of equalities between types. The resulting system will adequately model modules with no type abstraction.

$\mathcal{E}$ We will then add a sealing construct to the language in order to permit making types abstract. We will see that sealing makes the langage impure, and will equip our type system with a suitable effect system.

$\mathcal{C}$ The previous system can express type abstraction at the source level, but abstraction is lost when the program is evaluated. We will therefore provide a way of keeping track of abstraction boundaries during program evaluation, in the form of module identities and coloured brackets.

$\mathcal{D}$ We will finally be able to equip our language with dynamic typing constructs that behave reasonably in a distributed setting.

System $\mathcal{D}$ constitutes the full TOPHAT language[1].

At each stage, we will motivate the features to be introduced with examples, and we will examine how these features can be used in programs. We will discuss the choices we made when designing the theory presented here. We will then state precise the semantics of the language we define, in the form of typing rules (the static semantics) and small-step reduction rules (the dynamic semantics).

IV.2 A module calculus $\boxed{\mathcal{B}}$

The present section presents the core of a module description language. This core, which we call $\mathcal{B}$, builds on two essential features: aggregates of values and types, called *structures*; and parametric modules, called *functors*.

IV.2.1 Fundamental constructs

[Sorry, this fragment has not been translated yet.]

IV.2.2 About the base language

[Sorry, this fragment has not been translated yet.]

IV.2.3 Formal description of the core language

IV.2.3.1 Syntax

We can now formally state the syntax and semantics of our core language. We limit ourselves to the features mentioned so far, and delay singletons until section IV.3.

Since we have decided to unify the module and expression languages, objects formerly noted E and objects formerly noted M now belong to the same world, and shall be noted E and called

[1] *Total Or Partial Hashed Abstract Types*, in which the words "total" and "partial" refer to total and partial functors, also known as applicative and generative functors.

expressions. Similarly we will write T rather than S and speak of **types**. Nonetheless some expressions will intuitively be seen as modules, and their types as signatures.

E ::=	**expression or module**	T ::=	**type or signature**
$x \mid y \mid t \mid \ldots$	variables		
$()$	unit value	UNIT	unit
$\mathsf{false} \mid \mathsf{true}$	boolean (generically bv)	BOOL	booleans
$0 \mid 1 \mid \ldots$	integer (generically $\underline{n}$)	INT	integers
$\langle \mathsf{T} \rangle$	type field	TYPE	abstract type field
		$\mathsf{Typ}\,\mathsf{E}$	projection from a type field
$(\mathsf{E}_1, \mathsf{E}_2)$	pair	$\Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$	dependent sum
$\pi_i\, \mathsf{E}$	projection ($i \in \{1, 2\}$)		
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction	$\Pi x : \mathsf{T}_0.\, \mathsf{T}_1$	dependent product
$\mathsf{E}_1\, \mathsf{E}_2$	application		
$\mathsf{let}\, x = \mathsf{E}_0\, \mathsf{in}\, \mathsf{E} : \mathsf{T}$	local binding		

We will use the following abbreviations.

$\mathsf{T}_1 * \mathsf{T}_2 := \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$	product type
$\mathsf{T}_1 \to \mathsf{T}_2 := \Pi x : \mathsf{T}_1.\, \mathsf{T}_2$	arrow (function) type

In the definitions of $\mathsf{T}_1 * \mathsf{T}_2$ and $\mathsf{T}_1 \to \mathsf{T}_2$, x is a fresh variable, i.e., a variable that is not free in T_2.

IV.2.3.2 Variables

We use standard definitions of **free variables**, **bound occurrences**, **alpha-conversion** and **substitutions**. A **closed term** is one with no free variable.

We write **fv** $\aleph$ for the set of free variables of $\aleph$. We write $\{x \leftarrow \mathsf{E}\}\aleph$ for the substitution of E for x in $\aleph$.

We will systematically work *up to alpha-conversion*, i.e., any term that we write down will formally denote its equivalence class modulo alpha-conversion. Any typing or reduction step may rename variables. For example, if a typing or reduction rule requires more than one instantiation of a metavariable, each instantiation may use different representative for bound variables. This follows the tradition of the *Barendregt variable convention*, which allows for substantially clearer exposition. We will generally not mention the omnipresent possibility of alpha-conversion; syntax descriptions will mention the binding structure in the text.

IV.2.3.3 Environments

An **environment** is a finite list of (variable, type) pairs. We write nil for the empty environment, $x : \mathsf{T}$ for an environment of length 1, and use "," for concatenation, which we treat as associative. For example, an environment binding three variables will usually be written as $x : \mathsf{T}, y : \mathsf{T}', z : \mathsf{T}''$; other ways of writing the same environment are $((x : \mathsf{T}, y : \mathsf{T}'), z : \mathsf{T}'')$ and $(x : \mathsf{T}, y : \mathsf{T}'), z : \mathsf{T}''$ and $(((\mathsf{nil}, x : \mathsf{T}), y : \mathsf{T}'), z : \mathsf{T}'')$.

Environments are built from the following grammar:

Γ ::=	**environnement**
nil	empty
$\Gamma, x : \mathsf{T}$	binding of the variable x

Alternatively, environments may be seen as objects of the form $x_1 : \mathsf{T}_1, \ldots, x_k : \mathsf{T}_k$ with $k \in \mathbb{N}$.

The **domain** of an environnement $\Gamma = x_1 : \mathsf{T}_1, \ldots, x_k : \mathsf{T}_k$ is the set of variables $\{x_1, \ldots, x_k\}$. It is written **dom** Γ.

An environment binds the variables of its domain, and they are as usual subject to alpha-conversion. Writing (Γ, Γ') supposes that Γ and Γ' have disjoint domains; alpha-conversion must be performed if necessary. In a concatenation (Γ, Γ'), the variables in the domain of Γ bind in Γ'.

Note that our environments must be ordered since we have dependent types. Thus $(x : \textsc{int}, y : \mathsf{Typ}\, x)$ is a well-formed environment, whereas $(y : \mathsf{Typ}\, x, x : \textsc{int})$ is not (it could have been written as $(y : \mathsf{Typ}\, x, z : \textsc{int})$ after renaming the bound variable to z, with the remaining occurrence of x being free).

IV.2.4 Typing

IV.2.4.1 Introduction

We consider a correct program (fragment) to be an expresion E associated with a type T such that E has the type T. When E contains free variables, these must be assigned a type through an environment.

We will manipulate several forms of typing judgements, which will always be **local judgements**, of the form $\Gamma \vdash J$ (in system $\mathcal{B}$ — later local judgements will bear more annotations). System $\mathcal{B}$ has three forms of right-hand side for a local typing judgement. .

$\mathscr{J}$ ::=	**typing judgement**
$\Gamma \vdash J$	local judgement
J ::=	**local judgement right-hand side**
ok	environment correction
$T\,\mathsf{ok}$	correction of the T
$E : T$	expression typing

We present typing rule under the usual presentation as a deduction rules.

IV.2.4.2 $\boxed{\Gamma \vdash \mathsf{ok}}$ Environment corrections

Environments are built from left to right, binding by binding. Each type assigned to a variable must be valid in the environment that precedes the binding under consideration. Note that variables bound by an environment are automatically distinct as per our alpha-conversion convention.

$$\frac{}{\mathsf{nil} \vdash \mathsf{ok}}\ (\mathcal{B}/\mathbf{envok.nil}) \qquad \frac{\Gamma \vdash T\,\mathsf{ok}}{\Gamma, x : T \vdash \mathsf{ok}}\ (\mathcal{B}/\mathbf{envok.x})$$

IV.2.4.3 $\boxed{\Gamma \vdash T\,\mathsf{ok}}$ Type correctness

The correctness rules for types are standard: for base types, we require that the environment be well-formed, and for constructed types, we require each part to be well-formed (treating dependencies properly).

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{unit}\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{bool}\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{int}\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.base.int})$$

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{type}\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.type})$$

$$\frac{\Gamma \vdash T'\,\mathsf{ok} \qquad \Gamma, x : T' \vdash T''\,\mathsf{ok}}{\Gamma \vdash \Sigma x : T'.\, T''\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.pair}) \qquad \frac{\Gamma \vdash T'\,\mathsf{ok} \qquad \Gamma, x : T' \vdash T''\,\mathsf{ok}}{\Gamma \vdash \Pi x : T'.\, T''\,\mathsf{ok}}\ (\mathcal{B}/\mathbf{tok.fun})$$

IV.2.4.4 $\boxed{\Gamma \vdash E : T}$ Expression typing

Constants Basic constants have their respective type in a correct environment.

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash () : \text{UNIT}}\ (\mathcal{B}/\textbf{et.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash bv : \text{BOOL}}\ (\mathcal{B}/\textbf{et.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \underline{n} : \text{INT}}\ (\mathcal{B}/\textbf{et.base.int})$$

Variables Variables have the type stated in the environment.

$$\frac{\Gamma \vdash \mathsf{ok} \qquad \text{when } x : T \in \Gamma}{\Gamma \vdash x : T}\ (\mathcal{B}/\textbf{et.x})$$

Pairs Although our syntax allows dependent sums, system $\mathcal{B}$ is too restricted to take advantage of them (this defect will be remedied in system $\mathcal{S}$). We can only give pairs an ordinary pair type.

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 * T_2}\ (\mathcal{B}/\textbf{et.pair}) \qquad \frac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \pi_1 E : T_1}\ (\mathcal{B}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash E : T_1 * T_2}{\Gamma \vdash \pi_2 E : T_2}\ (\mathcal{B}/\textbf{et.proj.2})$$

Functions We state classical rules for typing functions (or functors) and application, keeping in mind that we have dependent types. Note that in order to type the application of a function that has a dependent type, the occurrences of the formal parameter x must be replaced by the actual argument E_0 inside the result type E. Thus an arbitrary expression can appear in a type where a simple variable formally was. This illustrates the difficulty of restricting the presence of expressions in types to certain syntactic categories.

$$\frac{\Gamma, x : T_0 \vdash E : T_1}{\Gamma \vdash \lambda x : T_0.\ E : \Pi x : T_0. T_1}\ (\mathcal{B}/\textbf{et.fun}) \qquad \frac{\Gamma \vdash E_1 : \Pi x : T_0. T \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash E_1\ E_0 : \{x \leftarrow E_0\} T}\ (\mathcal{B}/\textbf{et.app})$$

Local binding In order to type the local binding of a variable to a value, we request that the programmer specify the resulting type of the whole expression. Furthermore this type is not allowed to mention the locally bound variable. This last point is easily understood from the fact that while it would make sense for the variable x to be bound in the type of the body E, this variable cannot be free in $(\mathsf{let}\, x = E_0\ \mathsf{in}\, E : T)$. In particular, if E_0 were to create abstract types, there is no way to reference them outside the binding. The necessity for the programmer to specify the type is due to the avoidance problem mentioned in section I.2.2.6, which makes inference of T undecidable.

$$\frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma, x : T_0 \vdash E : T \qquad \Gamma \vdash T\ \mathsf{ok}}{\Gamma \vdash (\mathsf{let}\ x = E_0\ \mathsf{in}\ E : T) : T}\ (\mathcal{B}/\textbf{et.let})$$

IV.2.4.5 $\boxed{\langle T \rangle,\ \mathsf{Typ}\, E}$ Type fields

We can see $\langle \_\_ \rangle$ as a constructor for the type TYPE and $\mathsf{Typ}\,\_\_$ as the corresponding destructor. This approach yields suitable typing rules.

$$\frac{\Gamma \vdash T\ \mathsf{ok}}{\Gamma \vdash \langle T \rangle : \text{TYPE}}\ (\mathcal{B}/\textbf{et.type}) \qquad \frac{\Gamma \vdash E : \text{TYPE}}{\Gamma \vdash \mathsf{Typ}\, E\ \mathsf{ok}}\ (\mathcal{B}/\textbf{tok.field})$$

IV.2.5 Run-time

Values The class of **values** (generically written V) is given as a subgrammar of expressions.

$\mathsf{V} ::=$	**value**
$() \mid \mathsf{bv} \mid \underline{\mathsf{n}}$	constant
$\langle\mathsf{T}\rangle$	type field
$(\mathsf{V}_1, \mathsf{V}_2)$	pair
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction

IV.2.5.1 $\boxed{\mathsf{E} \longrightarrow \mathsf{E}'}$ Expression reduction

We define the dynamic behaviour of expressions via small-step reduction rules.

Head reduction rules In the language that we have defined so far, head reduction confronts each destructor with a matching constructor, and performs local bindings. We impose a *call-by-value* strategy in the rules ($\mathcal{B}$/ered.app) and ($\mathcal{B}$/ered.let). For the time being, we could allow β-reduction in its full generality, and obtain a confluent system; however we will ultimately introduce side effects, which suggests sticking to call-by-value.

$$(\lambda x : \mathsf{T}.\ \mathsf{E})\,\mathsf{V} \longrightarrow \{x \leftarrow \mathsf{V}\}\mathsf{E} \qquad (\mathcal{B}/\mathsf{ered.app})$$

$$\pi_i\,(\mathsf{V}_1, \mathsf{V}_2) \longrightarrow \mathsf{V}_i \qquad (\mathcal{B}/\mathsf{ered.proj})$$

$$\mathsf{let}\ x = \mathsf{V}\ \mathsf{in}\ \mathsf{E} : \mathsf{T} \longrightarrow \{x \leftarrow \mathsf{V}\}\mathsf{E} \qquad (\mathcal{B}/\mathsf{ered.let})$$

No reduction in types We do not define any reduction relation on types. Accordingly there is no restriction on T in order for ⟨T⟩ to be a value; in particular, if ⟨T⟩ contains embedded expressions (as in e.g., $\langle \mathsf{Typ}\,((\lambda x : \textsc{type}.\ x)\,\langle\textsc{int}\rangle)\rangle$) these need not be values. The reason is that computations in types traditionally belong in the compile-time world, hence to typing rules (and where relevant typing algorithms), rather than in the run-time world now under scrutiny. We will later (in system $\mathcal{D}$) add a construct for run-time type-checking, thus type computations will need to occur during program execution; run-time manipulation of types is also useful for generic programming (see section V.3.2.3).

Evaluation contexts We generically write C for an **evaluation context** of depth 1. These evaluation contexts are defined by the following grammar.

$\mathsf{C} ::=$	**evaluation context (of depth** 1)
$\mathsf{E}_1\,\_\_$	function argument
$\_\_\,\mathsf{V}_2$	applied function
$(\_\_, \mathsf{E}_2)$	first component of a pair
$(\mathsf{V}_1, \_\_)$	second component of a pair
$\pi_i\,\_\_$	projection ($i \in \{1, 2\}$)
$\mathsf{let}\ x = \_\_\ \mathsf{in}\ \mathsf{E} : \mathsf{T}$	local bound

We have arbitrarily fixed the evaluation order for function application (argument first, then function) and pairing (left to right). This somewhat simplifies the metatheory by not introducing spurious local nondeterminism. We could relax these constraints by authorising the reduction contexts $\_\_\,\mathsf{E}_2$ et $(\mathsf{E}_1, \_\_)$; it is folklore that the resulting reduction relation would be confluent.

The following reduction rule allows expressions to be reduced under contexts. The notation $\mathsf{C} \cdot \mathsf{E}$ means the expression resulting from placing E inside the context C.

$$\frac{\mathsf{E} \longrightarrow \mathsf{E}'}{\mathsf{C} \cdot \mathsf{E} \longrightarrow \mathsf{C} \cdot \mathsf{E}'}\ (\mathcal{B}/\mathbf{ered.context})$$

IV.3 Singletons $\boxed{\mathcal{S}}$

IV.3.1 Motivation

IV.3.1.1 Abstract types, concrete types

[Sorry, this fragment has not been translated yet.]

IV.3.1.2 Type sharing

[Sorry, this fragment has not been translated yet.]

IV.3.1.3 Value singletons

So far, we have used singleton types to express type equalities: our singletons were of the form $\mathsf{S}(\langle\mathsf{T}\rangle)$ for some type T. The purpose of these singletons was to enable making x have the type T' when x has the type T and $\langle\mathsf{T}\rangle$ and T' are equivalent: in other words, the judgement $\mathsf{t} : \mathsf{S}(\langle\mathsf{T}\rangle), x : \mathsf{T} \vdash x : \mathsf{Typ}\,\mathsf{t}$ should be derivable (one could then substitute $\langle\mathsf{T}'\rangle$ for t).

Let us now consider a functor f which creates an abstract type from a type and a value, with a signature of the form $\Pi x : (\Sigma\mathsf{t} : \textsc{type}.\,\mathsf{T}_0).\,(\Sigma\mathsf{t} : \textsc{type}.\,\mathsf{T}_1)$. As we saw in sections I.2.1.3 and II.5.1.1, $\mathsf{Typ}\,\pi_1\,(f\,x)$ and $\mathsf{Typ}\,\pi_1\,(f\,y)$ are the same types only when x and y have the same behaviour: it is not enough for them to provide the same types.

Let us consider an example potential argument for f, with $\mathsf{T}_0 = \mathsf{Typ}\,\mathsf{t} * (\mathsf{Typ}\,\mathsf{t} \to \textsc{unit})$.

```
module A = struct  type t = int  let x = (... : t * (t->unit))  end
```

The principal signature of the module `A` in Objective Caml is

```
module type S = sig  type t = int  val x : t * (t->unit)  end
```

If we want to express that some module `B` is compatible with `A`, the best we can do (whether in Objective Caml or in some other ML dialect, or in the language defined so far) is to specify that `B` has the signature `S`. Unfortunately this specification is incomplete since it does not distinguish between modules that have `x` fields with the same type but different values.

One way to illustrate this limitation is to consider an identity functor `Id1` capable of taking `A` as an argument. The principal signature of such a functor (which is based on Leroy's manifest type theory with applicative functors [Ler95]) is the following:

```
functor (A : sig  type t        val x : t * (t->unit)  end) ->
             sig  type t = A.t  val x : t * (t->unit)  end
```

Notice that the signature of `Id1(A)` is the signature `S` defined above: while it does indicate that `Id1(A).t` is equal to `A.t`, nothing connects `Id1(A).x` with `A.x` beyond them having the same type. The higher-order module theory of Dreyer, Crary and Harper [DCH03] does not perform any better on this example. With our notations, the signature of the functor `Id1` is $\Pi x : (\Sigma\mathsf{t} : \textsc{type}.\,\mathsf{Typ}\,\mathsf{t} * (\mathsf{Typ}\,\mathsf{t} \to \textsc{unit})).\,(\Sigma\mathsf{t}' : \mathsf{S}(\pi_1\,x).\,\mathsf{Typ}\,\mathsf{t}' * (\mathsf{Typ}\,\mathsf{t}' \to \textsc{unit}))$.

In the case of an identity functor $\mathsf{E}_{\mathtt{Id0}}$ acting solely on a type, i.e., whose argument has the signature $\textsc{type}$, we get a more precise signature: $\Pi x : \textsc{type}.\,\mathsf{S}(x)$, clearly indicating that the result of applying the identity functor is equivalent to the argument — $\mathsf{Typ}\,\mathsf{E}_{\mathtt{Id0}}\,(\mathsf{E}_{\mathtt{A0}})$ has the signature $\mathsf{S}(\mathsf{E}_{\mathtt{Id0}}\,(\mathsf{E}_{\mathtt{A0}}))$. We shall extend our language so that this property also holds for value fields.

We add singleton types of the form $\mathsf{S}(\mathsf{E})$, where E is any value. For example the principal type of the expression 3 is now $\mathsf{S}(3)$, the type of values that are equal to 3. We can now give `Id1` a more precise signature, by also giving the second component a singleton type: $\Pi x : (\Sigma t : \textsc{type}.\ \mathsf{Typ}\,t * (\mathsf{Typ}\,t \to \textsc{unit})).\ (\Sigma t' : \mathsf{S}(t).\ \mathsf{S}(\pi_1\, x) * \mathsf{S}(\pi_2\, x))$, i.e., in an Objective Caml-like notation

```
functor (A : sig  type t        val x : t * (t->unit)  end) ->
             sig  type t = A.t  val x = A.x            end
```

Thanks to this signature, `Id1(A)` can have the signature `sig type t = A.t val x = A.x end`, which makes it interchangeable with `A`.

IV.3.1.4 Higher-order singletons

[Sorry, this fragment has not been translated yet.]

IV.3.1.5 A practical example

Let us illustrate higher-order singletons on an example from the author's programming experience. The standard library of Objective Caml provides an implementation of finite sets via a functor `Set.Make`. This functor takes an argument with the following signature:

```
module type Set.OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

A module of signature `Set.OrderedType` provides a type as well as a function which must implement a total order; a set is represented as a search tree. An example of a module with this signature is `String`: `Set.Make(String).t` is therefore a type for sets of strings. The result returned by `Set.Make` has a signature called `Set.S` from which we quote a relevant excerpt:

```
module type Set.S = sig
  type elt                    (*type of elements*)
  type t                      (*type of sets*)
  val add : elt -> t -> t     (*addition function*)
  ...
end
```

An annotation in the definition of `Set.Make` specified that `Set.Make(M).elt = M.t`.

The program under consideration manipulates symbols, which are internally implemented as strings. However only suitable approved strings may be symbols, therefore the type of symbols is an abstract type provided by a module which we call `Syntax`.

```
module Syntax : sig
  type symbol
  val name : symbol -> String.t
  ...
end
```

Since several other modules in the program manipulate sets of symbols, we wish to provide this type alongside `symbol`. How do we mention the type of symbol sets in the signature of `Syntax`? We must specify a module of signature `Set.S`, indicating that the element type is that of symbols. For that purpose we need to define a symbol module.

```
module Syntax : sig
  module Symbol : Set.OrderedType
  module SymSet : Set.S with type elt = Symbol.t
                      and type t = Set.Make(Symbol).t
  ...
end
```

We could also write as follows:

```
module Syntax : sig
  module Symbol : Set.OrderedType
  module SymSet : Set.S
  ...
end with module SymSet = Set.Make(Symbol)
```

The signatures above are equivalent in Objective Caml.
The difficulty arises when writing the implementation of the `Syntax` module. We may write

```
module Syntax = struct
  module Symbol = struct type t = String.t let compare = String.compare end
  module SymSet = Set.Make(Symbol)
  ...
end
```

or even

```
module Syntax = struct
  module Symbol = String
  module SymSet = Set.Make(Symbol)
  ...
end
```

Unfortunately the resulting `SymSet` module is not compatible with `Set.Make(String)`. Since Objective Caml only ever compares type fields of modules, its type analysis remembers the equality between `Symbol.t` and `String.t` but not that between `Symbol` and `String`, therefore the types `Set.Make(String).t` and `Set.Make(Symbol).t` cannot safely be declared compatible.

Since our `Syntax` module calls other, lower-level modules that manipulate sets of strings, the incompatibility of `Set.Make(String).t` with `Set.Make(Symbol).t` is a major problem. The solution we chose was to only expose the symbol type, and not its comparison function:

```
module Syntax = struct
  type symbol = String.t
  module SymSet = Set.Make(String)
  ...
```

```
end : sig
  type symbol
  module SymSet : Set.S with type t = symbol
  ...
end
```

The disadvantage of this signature is that is hides the choice of a set implementation: that fact that `SymSet` is the result of an application of the `Set.Make` functor does not appear. This is a problem because some users of the `Syntax` modules manipulate data with more complex structures (e.g., sets of sets of symbols) built from functors that take an argument produced by `Set.Make`. We had to provide these data structures alongside `SymSet` in the `Syntax` module, even though these extra data structures had nothing to do in `Syntax` from a code organisation point of view.

In this case, simply being able to write `module Symbol = String` in the implementation of the `Syntax` module in such a way as to make the types `Set.Make(String).t` and `Set.Make(Symbole).t` interchangeable would have permitted the code to be organised properly, in particular with respect to abstraction. In system $\mathcal{S}$, this is possible, since the `Symbol` module will have the signature `S(x called "String")` which result in `Set.Make(String)` and `Set.Make(Symbol)` being compatible within the implementation of the `Syntax` module.

IV.3.2 Properties

[Sorry, this fragment has not been translated yet.]

IV.3.3 Typing rules

We state typing rules for system $\mathcal{S}$. The operational semantics (consisting of the reduction rules ($\mathcal{S}$/ered.app), ($\mathcal{S}$/ered.proj), ($\mathcal{S}$/ered.let), ($\mathcal{S}$/ered.context), as well as the definitions of values and reduction contexts) is unchanged from system $\mathcal{B}$.

System $\mathcal{S}$ contains new typing judgements for subtyping, conversion and convertibility.

$J ::=$	**local judgement right-hand side**
$\ldots$	
$T \longrightarrow T'$	typing conversion
$T \equiv T'$	convertibility equivalence on types
$E \longrightarrow E'$	expression conversion
$E \equiv E'$	convertibility equivalence on expressions
$T_1 <: T_2$	subtyping

Most typing rules of $\mathcal{B}$ are included in $\mathcal{S}$. The following rules are taken as is from $\mathcal{B}$ (and will not be repeated here):

- environment correction: all rules — ($\mathcal{S}$/envok.nil), ($\mathcal{S}$/envok.x);
- type correction: all rules — ($\mathcal{S}$/tok.base.unit), ($\mathcal{S}$/tok.base.bool), ($\mathcal{S}$/tok.base.int), ($\mathcal{S}$/tok.type), ($\mathcal{S}$/tok.pair), ($\mathcal{S}$/tok.fun), ($\mathcal{S}$/tok.field);
- expression typing: all rules except projections and local binding — ($\mathcal{S}$/et.base.unit), ($\mathcal{S}$/et.base.bool), ($\mathcal{S}$/et.base.int), ($\mathcal{S}$/et.x), ($\mathcal{S}$/et.pair), ($\mathcal{S}$/et.fun), ($\mathcal{S}$/et.app), ($\mathcal{S}$/et.type).

We omit local binding in system $\mathcal{S}$ because it is superfluous (see section IV.2.1.4), thus avoiding the need to take them into account in singleton typing rules.

IV.3.3.1 $\boxed{\Gamma \vdash \mathsf{T} <: \mathsf{T}' \ ; \ \dots}$ Subtyping

The subtyping relation explains how an expression may have more than one type, some of which are more precise than others. Intuitively, the type T is a subtype of T' whenever any expression that has the type T also has the type T'. We engrave the forward implication with an *implicit subtyping* rule.

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} <: \mathsf{T}'}{\Gamma \vdash \mathsf{E} : \mathsf{T}'} \ (\mathcal{S}/\textbf{et.sub})$$

Our subtyping relation is defined syntactically (by deduction rules) rather than semantically, in that nothing mandates the reverse application: it is possible for $\Gamma \vdash \mathsf{E} : \mathsf{T}'$ to be derivable whenever $\Gamma \vdash \mathsf{E} : \mathsf{T}$ is without the judgement $\Gamma \vdash \mathsf{T} <: \mathsf{T}'$ being derivable. Whether a subtyping relation should be semantic (i.e., fully capture type subsumption for expressions) is debatable. On the one hand, a semantic subtyping relation permits a set-theoretic interpretation of types as sets of expressions. On the other hand, the rules needed to enforce semanticity would be fragile, in that they would not play well with extensions of the system. For example, if V is a value of type T, then with semantic subtyping $\mathsf{T} <: \mathsf{S}(\mathsf{V})$ must hold whenever T contains the single value V, which may happen coincidentally. Consider for instance the type $\Pi \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t} \to \mathsf{Typ}\,\mathsf{t}$, which obviously contains the polymorphic identity function $(\lambda \mathsf{t} : \textsc{type}.\ \lambda \mathsf{x} : \mathsf{Typ}\,\mathsf{t}.\ \mathsf{x}))$. In a suitably weak system, such as system $\mathcal{S}$, a parametricity [Wad89] result ensures that there is no other function of this type. However adding either dynamic type-checking (as we will do in system $\mathcal{D}$) or an unrestricted fixpoint combinator would let one write other functions of this type.

If two types are interconvertible, they are subtypes of one another. Thus subtyping includes computational equivalences on types.

$$\frac{\Gamma \vdash \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash \mathsf{T} <: \mathsf{T}'} \ (\mathcal{S}/\textbf{tsub.eq})$$

Subtyping is a preorder. The rule $(\mathcal{S}/\mathsf{tsub.eq})$ enforces reflexivity; we state transitivity.

$$\frac{\Gamma \vdash \mathsf{T} <: \mathsf{T}' \qquad \Gamma \vdash \mathsf{T}' <: \mathsf{T}''}{\Gamma \vdash \mathsf{T} <: \mathsf{T}''} \ (\mathcal{S}/\textbf{tsub.trans})$$

IV.3.3.2 $\boxed{\mathsf{S}(\mathsf{E})}$ Singletons

Singleton types appear through three generic rules, which have no constraint on the type of the expression whose singleton is taken. The singleton $\mathsf{S}(\mathsf{E})$ is well-formed as soon as E has some type T; any well-typed expression E thus has the type $\mathsf{S}(\mathsf{E})$, and $\mathsf{S}(\mathsf{E})$ is a subtype of any of its types. Note that in order to prove that E has the type $\mathsf{S}(\mathsf{E})$, one must first find some type T that E has and then apply $(\mathcal{S}/\mathsf{et.sing})$.

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E})\ \mathsf{ok}} \ (\mathcal{S}/\textbf{tok.sing}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{E} : \mathsf{S}(\mathsf{E})} \ (\mathcal{S}/\textbf{et.sing}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E}) <: \mathsf{T}} \ (\mathcal{S}/\textbf{tsub.sing})$$

The rule $(\mathcal{S}/\mathsf{et.sing})$ (combined with the subtyping rules) is a particularly powerful instance of a selfification rule in a module language with manifest types [HL94, Ler94] (see section I.2.2.2).

IV.3.3.3 $\boxed{\Gamma \vdash \mathsf{E} : \mathsf{T} \ \ ; \Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}$ Expression typing

As in system $\mathcal{B}$, we first assign non-dependent types to pairs. A dependent type can be obtained via the subtyping rule $(\mathcal{S}/\mathsf{tsub.cong.pair})$ (recall that $\mathsf{T}_1 * \mathsf{T}_2$ is an abbreviation for $\Sigma \mathsf{x} : \mathsf{T}_1.\ \mathsf{T}_2$). In this rule, note that the second premise contains the stronger hypothesis on x, namely $\mathsf{x} : \mathsf{T}_1'$, which follows from the fact that the hypothesis $\mathsf{x} : \mathsf{T}_2'$ might not be enough to ensure that T_1'' be valid. A third premise ensures that $\Sigma \mathsf{x} : \mathsf{T}_2'.\ \mathsf{T}_2''$ is well-formed, which requires that T_2'' be well-formed under the weaker hypothesis $\mathsf{x} : \mathsf{T}_2'$.

$$\frac{\Gamma \vdash T_1 <: T_1' \qquad \Gamma, x : T_1 \vdash T_2 <: T_2' \qquad \Gamma, x : T_1' \vdash T_2' \ \mathsf{ok}}{\Gamma \vdash \Sigma x : T_1.\, T_2 <: \Sigma x : T_1'.\, T_2'} \ (\mathcal{S}/\textbf{tsub.cong.pair})$$

In order to type a projection, the argument expression must be given a (dependent) pair type. Typing the first projection is simple, as its type is readily available in the pair type. Typing the second projection is more complicated. If the expression E has the type $\Sigma x : T_1.\, T_2$ then $\pi_2\, E$ only has the type T_2 with a suitably strong hypothesis on the variable x. For instance the expression $(3,3)$ has the type $\Sigma x : \text{INT}.\, \mathsf{S}(x)$, and while $x : \mathsf{S}(3) \vdash \pi_2\,(3,3) : \mathsf{S}(x)$ holds, $x : \text{INT} \vdash \pi_2\,(3,3) : \mathsf{S}(x)$ does not.

$$\frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_1\, E : T_1} \ (\mathcal{S}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2 \qquad \Gamma \vdash E_1 : \mathsf{S}(\pi_1\, E)}{\Gamma \vdash \pi_2\, E : \{x \leftarrow E_1\} T_2} \ (\mathcal{S}/\textbf{et.proj.2})$$

The premises of the rule ($\mathcal{S}$/et.proj.2) are usually unduely constraining, and we will often use one of two admissible variants that only require the first premise $\Gamma \vdash E : \Sigma x : T_1.\, T_2$. The most common rule replaces the variable x by the first projection of E in T_2. Another variant keeps track of the first component via a variable x which is constrained to the type $\mathsf{S}(\pi_1\, E)$.

$$\frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_2\, E : \{x \leftarrow_c \pi_1\, E\} T_2} \ (\text{et.proj.2s}) \qquad \frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma, x : \mathsf{S}(\pi_1\, E) \vdash \pi_2\, E : T_2} \ (\text{et.proj.2x})$$

We state the usual congruence rule for subtyping through dependent products. This rule is similar to ($\mathcal{S}$/tsub.cong.pair), with hypotheses suitably reversed when dealing with the contravariant argument type.

$$\frac{\Gamma \vdash T_0' <: T_0 \qquad \Gamma, x : T_0' \vdash T_1 <: T_1' \qquad \Gamma, x : T_0 \vdash T_1 \ \mathsf{ok}}{\Gamma \vdash \Pi x : T_0.\, T_1 <: \Pi x : T_0'.\, T_1'} \ (\mathcal{S}/\textbf{tsub.cong.fun})$$

IV.3.3.4 $\boxed{\Gamma \vdash T \equiv T' \ ; \ \Gamma \vdash E \equiv E'}$ Convertibility equivalences

We define an equivalence relation on types and one on expressions as the smallest equivalence relation containing the appropriate conversion relation. (Strictly speaking these are two families of relations, indexed by environments.) These relations are called **convertibility**.

$$\frac{\Gamma \vdash T \ \mathsf{ok}}{\Gamma \vdash T \equiv T} \ (\mathcal{S}/\textbf{teq.refl}) \qquad \frac{\Gamma \vdash T_2 \equiv T_1}{\Gamma \vdash T_1 \equiv T_2} \ (\mathcal{S}/\textbf{teq.sym})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 \qquad \Gamma \vdash T_2 \equiv T_3}{\Gamma \vdash T_1 \equiv T_3} \ (\mathcal{S}/\textbf{teq.trans}) \qquad \frac{\Gamma \vdash T_1 \longrightarrow T_2}{\Gamma \vdash T_1 \equiv T_2} \ (\mathcal{S}/\textbf{teq.conv})$$

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash E \equiv E} \ (\mathcal{S}/\textbf{eeq.refl}) \qquad \frac{\Gamma \vdash E_2 \equiv E_1}{\Gamma \vdash E_1 \equiv E_2} \ (\mathcal{S}/\textbf{eeq.sym})$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 \qquad \Gamma \vdash E_2 \equiv E_3}{\Gamma \vdash E_1 \equiv E_3} \ (\mathcal{S}/\textbf{eeq.trans}) \qquad \frac{\Gamma \vdash E_1 \longrightarrow E_2}{\Gamma \vdash E_1 \equiv E_2} \ (\mathcal{S}/\textbf{eeq.conv})$$

IV.3.3.5 $\boxed{\Gamma \vdash T \longrightarrow T'}$ Type conversion

Type conversion mostly consists in conversion of embedded expressions. Types additionally undergo some slight simplification.

Conversion is only defined on valid types; the rules defining conversion contain correction premises in addition to the conversion premises in context rules.

Contexts At the type level, conversion applies recursively to all subtypes. We state this via context rules which allow conversion of both type and expression subterms of types.

$$\frac{\Gamma \vdash T_1 \longrightarrow T_1' \qquad \Gamma, x : T_1 \vdash T_2\ \mathsf{ok}}{\Gamma \vdash \Sigma x : T_1 .\ T_2 \longrightarrow \Sigma x : T_1' .\ T_2}\ (\mathcal{S}/\textbf{tconv.cong.pair.1})$$

$$\frac{\Gamma, x : T_1 \vdash T_2 \longrightarrow T_2' \qquad \Gamma \vdash T_1\ \mathsf{ok}}{\Gamma \vdash \Sigma x : T_1 .\ T_2 \longrightarrow \Sigma x : T_1 .\ T_2'}\ (\mathcal{S}/\textbf{tconv.cong.pair.2})$$

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash T_1\ \mathsf{ok}}{\Gamma \vdash \Pi x : T_0 .\ T_1 \longrightarrow \Pi x : T_0' .\ T_1}\ (\mathcal{S}/\textbf{tconv.cong.fun.arg})$$

$$\frac{\Gamma \vdash T_0\ \mathsf{ok} \qquad \Gamma, x : T_0 \vdash T_1 \longrightarrow T_1'}{\Gamma \vdash \Pi x : T_0 .\ T_1 \longrightarrow \Pi x : T_0 .\ T_1'}\ (\mathcal{S}/\textbf{tconv.cong.fun.ret})$$

$$\frac{\Gamma \vdash E \longrightarrow E'}{\Gamma \vdash \mathsf{S}(E) \longrightarrow \mathsf{S}(E')}\ (\mathcal{S}/\textbf{tconv.cong.sing}) \qquad \frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \text{TYPE}}{\Gamma \vdash \mathsf{Typ}\, E \longrightarrow \mathsf{Typ}\, E'}\ (\mathcal{S}/\textbf{tconv.cong.field})$$

Simplifications The term $\mathsf{Typ}\,\langle T\rangle$ can be seen as a destructor applied to the matching constructor applied to T; it is equivalent to T.

$$\frac{\Gamma \vdash T\ \mathsf{ok}}{\Gamma \vdash \mathsf{Typ}\,\langle T\rangle \longrightarrow T}\ (\mathcal{S}/\textbf{tconv.field})$$

Semantic rules We declare that the UNIT type contains a single value: this type is isomorphic to a singleton, and the rule ($\mathcal{S}$/tconv.unit) enshrines this equivalence into the syntactic definition of type equivalence. The choice of orientation in this rule does not matter greatly.

$$\frac{\Gamma \vdash\ \mathsf{ok}}{\Gamma \vdash \mathsf{S}(()) \longrightarrow \text{UNIT}}\ (\mathcal{S}/\textbf{tconv.unit})$$

IV.3.3.6 $\boxed{\Gamma \vdash E \longrightarrow E'}$ Expression conversion

If E evaluates to E' and E has the type $\mathsf{S}(E)$, then type preservation requires that E' have the type $\mathsf{S}(E)$. This is ensured by making E convertible to E'. Conversion thus includes run-time reduction[2].

Conversion is only defined on well-typed expressions; the rules defining conversion contain correction premises in addition to the conversion premises in context rules.

We do not state any conversion rule for local binding expressions (i.e., expressions of the form $\mathsf{let}\, x = E_0\ \mathsf{in}\ E : T$). As indicated in section IV.2.1.4, this construct is just syntactic sugar when E_0 is pure; in $\mathcal{E}$, where local binding becomes useful, it is always judged impure and thus not subject to conversion.

Contexts Unlike for the run-time reduction relation, there is no particular advantage to keeping conversion deterministic, while confluence of conversion is a key property of our metatheoretic study. We therefore permit arbitrary evaluation strategies, and allow reduction in any context. In preparation for the addition of impure constructs to the language, which only come in conversion inside function bodies that hide the impurity, we state the slightly peculiar rule ($\mathcal{S}$/econv.cong.fun.body) to allow the conversion of any pure subexpression of an impure subexpression of a pure expression.

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash E_1 : T_1}{\Gamma \vdash (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1)}\ (\mathcal{S}/\textbf{econv.cong.fun.arg})$$

[2]For pure expressions, as we shall see in system $\mathcal{E}$.

$$\frac{\Gamma, x : T_0 \vdash E \longrightarrow E' \qquad \Gamma, x : T_0, y : S(E) \vdash E_1 : T_1}{\Gamma \vdash (\lambda x : T_0.\ \{y \leftarrow E\} E_1) \longrightarrow (\lambda x : T_0.\ \{y \leftarrow E'\} E_1)} \ (\mathcal{S}/\textbf{econv.cong.fun.body})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \Pi x : T_0.\, T_1 \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash E\, E_0 \longrightarrow E'\, E_0} \ (\mathcal{S}/\textbf{econv.cong.app.fun})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : T_0 \qquad \Gamma \vdash E_1 : \Pi x : T_0.\, T_1}{\Gamma \vdash E_1\, E \longrightarrow E_1\, E'} \ (\mathcal{S}/\textbf{econv.cong.app.arg})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E, E_2) \longrightarrow (E', E_2)} \ (\mathcal{S}/\textbf{econv.cong.pair.1}) \qquad \frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash (E_1, E) \longrightarrow (E_1, E')} \ (\mathcal{S}/\textbf{econv.cong.pair.2})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_i\, E \longrightarrow \pi_i\, E'} \ (\mathcal{S}/\textbf{econv.cong.proj}) \qquad \frac{\Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \langle T \rangle \longrightarrow \langle T' \rangle} \ (\mathcal{S}/\textbf{econv.cong.field})$$

Head reduction The following rules describe the usual evaluation of lambda terms with pairs.

$$\frac{\Gamma, x : T_0 \vdash E_1 : T_1 \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow E_0\} E_1} \ (\mathcal{S}/\textbf{econv.app}) \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash \pi_i\, (E_1, E_2) \longrightarrow E_i} \ (\mathcal{S}/\textbf{econv.proj})$$

IV.3.3.7 Extensionality

We state extensionality rules for system $\mathcal{S}$. Such rules can have many forms; we choose to use conversion rules, oriented as eta-expansions. For example the rule ($\mathcal{S}$/econv.eta.pair) may be read as "any expression that is typable as a pair can be rewritten in such a way as to expose the pair structure".

Given the choice of using conversion rules (η conversions to supplement the β conversions above), there is a further choice between expansions and contractions. A major technical advantage of expansions is that they do not hurt the confluence of the system, unlike eta-contractions [Klo80]. Expansions do however have the obvious defect of breaking normalisation. In practice, it seems preferable to express extensionality using expansions, and when normalisation is required to restrict their use to a finite domain (given by the structure of the type of the converted expression) [Gog05].

$$\frac{\Gamma \vdash E : \text{TYPE}}{\Gamma \vdash E \longrightarrow \langle \mathsf{Typ}\, E \rangle} \ (\mathcal{S}/\textbf{econv.eta.field})$$

$$\frac{\Gamma \vdash E : \Pi x : T_0.\, T_1}{\Gamma \vdash E \longrightarrow (\lambda x : T_0.\ E\, x)} \ (\mathcal{S}/\textbf{econv.eta.fun}) \qquad \frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash E \longrightarrow (\pi_1\, E, \pi_2\, E)} \ (\mathcal{S}/\textbf{econv.eta.pair})$$

IV.4 Sealing $\boxed{\mathcal{E}}$

IV.4.1 Sealing

[Sorry, this fragment has not been translated yet.]

IV.4.2 An effect system

IV.4.2.1 Introduction

[Sorry, this fragment has not been translated yet.]

IV.4.2.2 Purity

[Sorry, this fragment has not been translated yet.]

IV.4.2.3 Projectibility, separability and comparability

In the system we are describing, a "well-behaved" module is a pure module. Purity is a very strong notion: a pure module is fully known statically (it has its singleton type, and singleton types fully characterise an object). There are finer notions to determine legitimate uses of a module; in this section we will discuss some of these.

Classical module calculi, notably Harper and Lillibridge's translucent sums [HL94, Lil97] and Leroy's manifest types [Ler94, Ler95] focus on the concept of **projectibility** (see section I.2.2.2). The question is, given a module expression M, whether the term M.t may be used to form a type. If so, the module M is said to be **projectible**. In our notation, E is projectible if and only if $\mathsf{Typ}\,\pi_1\,\mathsf{E}$ is a correct type. We approximate projectibility by purity: $\mathsf{Typ}\,\pi_1\,\mathsf{E}$ is correct if and only if E is pure (and has an appropriate signature). This is indeed an approximation since purity is a stronger notion; for example, in the following code fragment, the modules A and B are both projectible, but only A is pure, while B is impure.

```
module A = struct  type t = int  let x = 3  end
module B = struct  type t = int  let x = ref 3  end
```

A closely related notion is that of **comparability**: a module is said to be **comparable** when its equivalence with another module can be tested. In the calculus of Dreyer, Crary and Harper [DCH03], the notions de comparability and projectability coincide, since testing the equivalence of two modules amounts to comparing their type components. In our calculus, purity stands for comparability as well as projectability (we treat type and value components identically).

In section I.3.1.1, we mentioned the issue of *phase separation*, i.e., clearly differentiating between the static phase of the program, which includes a type-check that rejects programs that would go wrong, and the dynamic phase, during which computation proceeds without errors. In the core of ML, each phase is closely associated with one part of the language: type-checking is mostly concerned with types, and computation is mostly concerned with expressions. This is no longer true when modules are considered, as they mix types and expressions at the syntactic level. Nevertheless one usually tries to separate types and expressions in the metatheory of modules, in order to distinguish between the static and dynamic aspects.

In his analysis of ML modules [Dre05], Dreyer distinguishes between two levels of purity in modules. A module is said to be **totally pure**[3] if it is pure in our sense, i.e., that its evaluation does not trigger an effect of any kind. A module is said to be **partially pure** if its type components can be fully determined without triggering an effect. For example the module B above is partially pure but not totally pure. A projectible module must be partially pure.

One difficulty with partial purity is in deciding whether the effects of an expression have an influence on its type parts. Total purity is of course a sufficient condition. A weaker sufficient condition is **separability**. This notion was introduced by Harper, Mitchell and Moggi [HMM90] and is expounded in the context of a module calculus with functors by Dreyer [Dre05]. A module is **separable** if its type components do not depend at all on any computation that may have effects, in particular any core-expression-level computation. A separable module is always partially pure,

[3]Actually Dreyer uses the wording "dynamically pure" and "statically pure" where we use "totally pure" and "partially pure". We changed the terminology because we will use "dynamically pure" and "statically pure" in a different sense, following other work by Dreyer [DCH03].

hence projectible, but may be impure, like B above. Conversely, in a language with first-class modules, one can easily write pure inseparable modules, such as the module C in the following program fragment.

```
let n = read_int ()
module A  = struct  type t = int  let x = 3  end
module B  = struct  type t = int  let x = ref 3  end
module A' = struct  type t = bool let x = true  end
module B' = struct  type t = bool let x = ref true  end
module C = if n >= 0 then A else A'
module D = if n >= 0 then B else B'
```

The module D is partially pure but neither totally pure nor separable.

An important conclusion of the discussion of type singletons and module equivalence in IV.3.1 was that in the presence of functors, separability is hard to analyse — and this is why we did not try to analyse it, and instead integrated expressions with types when testing module equivalence. Separability looks all the less enticing to us as we eventually want to be able to compare types dynamically, which means that our notion of equivalence must work well even in inseparable cases. In the present work, we do no try to go further than (total) purity. If refinements are desired, rather than introduce separability, we suggest instead to make the effect system more sophisticated, and in particular to make it possible to detect partial purity by "declassifying" effects that do not impact the value of an expression.

IV.4.3 Formal presentation

We give a formal description of system $\mathcal{E}$, which consists of adding the sealing construct to $\mathcal{S}$ and, more importantly, an effect system.

IV.4.3.1 Syntax

We first give the syntax of effects.

$\gamma ::=$		**effect**
	$\mathtt{P}$	pure
	$\mathtt{I}$	impure

Recall that effects are ordered: the relation $\gamma_1 \sqsubseteq \gamma_2$ is such that $\mathtt{P} \sqsubseteq \mathtt{I}$ (but not the converse). We write $\gamma_1 \sqcup \gamma_2$ for the least upper bound of γ_1 and γ_2, and $\gamma_1 \sqcap \gamma_2$ for their greatest lower bound.

The syntax of system $\mathcal{E}$ extends that of system $\mathcal{S}$ by adding an effect annotation where necessary, viz.,

- on function types, henceforth written $\Pi x : T_0.\, {}^{\gamma}T_1$; they are abbreviated as $T_0 \to^{\gamma} T_1$ when x is not free in T_1;
- on expression typing judgements, henceforth written $E :^{\gamma} T$.

We sometimes omit the effect annotation when it is $\mathtt{P}$, thus we might write the type of a pure function as $\Pi x : T_0.\, T_1$ or $T_0 \to T_1$. Furthermore the syntax of expressions now comprises sealing.

$T ::=$		**type**
	...	
	$\Pi x : T_0.\, {}^{\gamma}T_1$	dependent product (also written $T_1 \to^{\gamma} T_2$ when $x \notin \mathbf{fv}\, T_1$)

E ::=	**expression (module)**
...	
$\mathsf{E} \mathbin{!!} \mathsf{T}$	sealing

J ::=	**typing judgement right-hand side**
...	
$\mathsf{E} :^{\gamma} \mathsf{T}$	expression typing

IV.4.3.2 $\boxed{\mathsf{E} \longrightarrow \mathsf{E}'}$ Run-time

The only run-time novelty of system $\mathcal{E}$ is the need to reduce a sealing construct. The usual intuition in ML-like languages is that types have no bearing on execution, only on static type-checking; in this light, $\mathsf{E} \mathbin{!!} \mathsf{T}$ is equivalent to E at run-time.

$$\mathsf{V} \mathbin{!!} \mathsf{T} \longrightarrow \mathsf{V} \qquad (\mathcal{E}/\mathsf{ered.seal})$$

The sealed expression is first reduced to a value.

C ::=	**evaluation context (of depth 1)**
...	
$\_ \mathbin{!!} \mathsf{T}$	sealing

The rules ($\mathcal{E}$/ered.app), ($\mathcal{E}$/ered.proj), ($\mathcal{E}$/ered.let) et ($\mathcal{E}$/ered.context) are inherited from system $\mathcal{B}$ via $\mathcal{S}$.

IV.4.3.3 $\boxed{\Gamma \vdash \ldots}$ Typing: correction, equivalences, subtyping

System $\mathcal{E}$ contains all the typing rules of system $\mathcal{S}$, and adds one for sealing. However the inherited rules must usually be modified to add an effect annotation. We will restate affected rules and explain the effect of effects.

Inhreited rules The following rules are taken as is from system $\mathcal{B}$ via $\mathcal{S}$:

- ($\mathcal{E}$/envok.nil), ($\mathcal{E}$/envok.x);
- ($\mathcal{E}$/tok.base.unit), ($\mathcal{E}$/tok.base.bool), ($\mathcal{E}$/tok.base.int), ($\mathcal{E}$/tok.type), ($\mathcal{E}$/tok.pair);
- ($\mathcal{E}$/teq.refl), ($\mathcal{E}$/teq.sym), ($\mathcal{E}$/teq.trans), ($\mathcal{E}$/teq.conv);
- ($\mathcal{E}$/eeq.sym), ($\mathcal{E}$/eeq.trans), ($\mathcal{E}$/eeq.conv);
- ($\mathcal{E}$/tconv.cong.pair.1), ($\mathcal{E}$/tconv.cong.pair.2), ($\mathcal{E}$/econv.cong.field), ($\mathcal{E}$/tconv.cong.sing), ($\mathcal{E}$/tconv.field), ($\mathcal{E}$/tconv.unit);
- ($\mathcal{E}$/tsub.trans), ($\mathcal{E}$/tsub.eq), ($\mathcal{E}$/tsub.cong.pair).

Apart from expression typing and from ($\mathcal{E}$/tsub.cong.fun), the modifications to the rules of system $\mathcal{S}$ consist of requiring expressions embedded in types to be pure, and allow dependent product types to bear effect annotations. The rules ($\mathcal{E}$/econv.cong.fun.arg) and ($\mathcal{E}$/econv.cong.fun.body) do however permit the body of the function to be pure, as all that is required is that the function itself be a pure expression.

$$\frac{\Gamma \vdash \mathsf{T}'\,\mathsf{ok} \qquad \Gamma, x : \mathsf{T}' \vdash \mathsf{T}''\,\mathsf{ok}}{\Gamma \vdash \Pi x : \mathsf{T}'.\,{}^{\gamma}\mathsf{T}''\,\mathsf{ok}}\,(\mathcal{E}/\textbf{tok.fun}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \textsc{type}}{\Gamma \vdash \mathsf{Typ}\,\mathsf{E}\,\mathsf{ok}}\,(\mathcal{E}/\textbf{tok.field}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E})\,\mathsf{ok}}\,(\mathcal{E}/\textbf{tok.sing})$$

$$\frac{\Gamma \vdash E :^{P} T}{\Gamma \vdash E \equiv E}\ (\mathcal{E}/\textbf{eeq.refl}) \qquad \frac{\Gamma \vdash E :^{P} T}{\Gamma \vdash S(E) <: T}\ (\mathcal{E}/\textbf{tsub.sing})$$

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash T_1\ \mathsf{ok}}{\Gamma \vdash \Pi x : T_0.\, {}^{\gamma}T_1 \longrightarrow \Pi x : T_0'.\, {}^{\gamma}T_1}\ (\mathcal{E}/\textbf{tconv.cong.fun.arg})$$

$$\frac{\Gamma \vdash T_0\ \mathsf{ok} \qquad \Gamma, x : T_0 \vdash T_1 \longrightarrow T_1'}{\Gamma \vdash \Pi x : T_0.\, {}^{\gamma}T_1 \longrightarrow \Pi x : T_0.\, {}^{\gamma}T_1'}\ (\mathcal{E}/\textbf{tconv.cong.fun.ret})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E :^{P} \text{TYPE}}{\Gamma \vdash \mathsf{Typ}\, E \longrightarrow \mathsf{Typ}\, E'}\ (\mathcal{E}/\textbf{tconv.cong.field})$$

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash E_1 :^{\gamma} T_1}{\Gamma \vdash (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1)}\ (\mathcal{E}/\textbf{econv.cong.fun.arg})$$

$$\frac{\Gamma, x : T_0 \vdash E \longrightarrow E' \qquad \Gamma, x : T_0, y : S(E) \vdash E_1 :^{\gamma} T_1}{\Gamma \vdash (\lambda x : T_0.\ \{y \leftarrow E\}E_1) \longrightarrow (\lambda x : T_0.\ \{y \leftarrow E'\}E_1)}\ (\mathcal{E}/\textbf{econv.cong.fun.body})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_2 :^{P} T_2}{\Gamma \vdash (E, E_2) \longrightarrow (E', E_2)}\ (\mathcal{E}/\textbf{econv.cong.pair.1}) \qquad \frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_1 :^{P} T_1}{\Gamma \vdash (E_1, E) \longrightarrow (E_1, E')}\ (\mathcal{E}/\textbf{econv.cong.pair.2})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E :^{P} \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_i\, E \longrightarrow \pi_i\, E'}\ (\mathcal{E}/\textbf{econv.cong.proj})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E :^{P} \Pi x : T_0.\, {}^{P}T_1 \qquad \Gamma \vdash E_0 :^{P} T_0}{\Gamma \vdash E\, E_0 \longrightarrow E'\, E_0}\ (\mathcal{E}/\textbf{econv.cong.app.fun})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E :^{P} T_0 \qquad \Gamma \vdash E_1 :^{P} \Pi x : T_0.\, {}^{P}T_1}{\Gamma \vdash E_1\, E \longrightarrow E_1\, E'}\ (\mathcal{E}/\textbf{econv.cong.app.arg})$$

$$\frac{\Gamma \vdash E_1 :^{P} T_1 \qquad \Gamma \vdash E_2 :^{P} T_2}{\Gamma \vdash \pi_i\,(E_1, E_2) \longrightarrow E_i}\ (\mathcal{E}/\textbf{econv.proj}) \qquad \frac{\Gamma, x : T_0 \vdash E_1 :^{P} T_1 \qquad \Gamma \vdash E_0 :^{P} T_0}{\Gamma \vdash (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow E_0\}E_1}\ (\mathcal{E}/\textbf{econv.app})$$

$$\frac{\Gamma \vdash E :^{P} \text{TYPE}}{\Gamma \vdash E \longrightarrow \langle \mathsf{Typ}\, E \rangle}\ (\mathcal{E}/\textbf{econv.eta.field})$$

$$\frac{\Gamma \vdash E :^{P} \Pi x : T_0.\, {}^{\gamma}T_1}{\Gamma \vdash E \longrightarrow (\lambda x : T_0.\ E\, x)}\ (\mathcal{E}/\textbf{econv.eta.fun}) \qquad \frac{\Gamma \vdash E :^{P} \Sigma x : T_1.\, T_2}{\Gamma \vdash E \longrightarrow (\pi_1\, E, \pi_2\, E)}\ (\mathcal{E}/\textbf{econv.eta.pair})$$

Subtyping for functions The congruence rule for subtyping accross dependent products needs to take effects into account. A function type is smaller than another function type when the domain of the first is smaller, the image of the first is larger, and the first allows fewer effects to occur during execution.

$$\frac{\Gamma \vdash T_0' <: T_0 \qquad \Gamma, x : T_0' \vdash T_1 <: T_1' \qquad \Gamma, x : T_0 \vdash T_1\ \mathsf{ok} \qquad \text{when } \gamma \sqsubseteq \gamma'}{\Gamma \vdash \Pi x : T_0.\, {}^{\gamma}T_1 <: \Pi x : T_0'.\, {}^{\gamma'}T_1'}\ (\mathcal{E}/\textbf{tsub.cong.fun})$$

IV.4.3.4 $\boxed{\Gamma \vdash E :^{\gamma} T}$ Expression typing

Expression typing judgements now carry an effect annotation.

Constants, variables, type fields Constantes, variables and type fields are always pure.

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash () :^{\mathtt{P}} \text{UNIT}}\ (\mathcal{E}/\textbf{et.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash bv :^{\mathtt{P}} \text{BOOL}}\ (\mathcal{E}/\textbf{et.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \underline{n} :^{\mathtt{P}} \text{INT}}\ (\mathcal{E}/\textbf{et.base.int})$$

$$\frac{\Gamma \vdash \mathsf{ok} \qquad \text{when } x : T \in \Gamma}{\Gamma \vdash x :^{\mathtt{P}} T}\ (\mathcal{E}/\textbf{et.x}) \qquad \frac{\Gamma \vdash T\,\mathsf{ok}}{\Gamma \vdash \langle T \rangle :^{\mathtt{P}} \text{TYPE}}\ (\mathcal{E}/\textbf{et.type})$$

Pairs Pairs are simple data structures: the type of a pair simply indicates the types of its components. The information as to which component of the pair carries which effect is lost. Therefore typing a pair requires a common effect annotation to be found for its components (the rule ($\mathcal{E}$/et.sub) used on each side allows one to use the least upper bound). Similarly, the effects of a first projection are the effects of the original expression. The second projection can only be used on a pure expression since the expression appears in a type[4].

$$\frac{\Gamma \vdash E_1 :^{\gamma} T_1 \qquad \Gamma \vdash E_2 :^{\gamma} T_2}{\Gamma \vdash (E_1, E_2) :^{\gamma} T_1 * T_2}\ (\mathcal{E}/\textbf{et.pair})$$

$$\frac{\Gamma \vdash E :^{\gamma} \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_1 E :^{\gamma} T_1}\ (\mathcal{E}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash E :^{\mathtt{P}} \Sigma x : T_1.\, T_2 \qquad \Gamma \vdash E_1 :^{\mathtt{P}} S(\pi_1 E)}{\Gamma \vdash \pi_2 E :^{\mathtt{P}} \{x \leftarrow E_1\} T_2}\ (\mathcal{E}/\textbf{et.proj.2})$$

Functions An immediate function is always pure. The abstraction construct suspends the effects of the function, which are reflected in the effect annotation on the function type. When a function is applied, the effects of the body are released and add to the effects of the function expression. We require the argument of a function to be pure as it is substituted into the result type.

$$\frac{\Gamma, x : T_0 \vdash E :^{\gamma} T_1}{\Gamma \vdash \lambda x : T_0.\ E :^{\mathtt{P}} \Pi x : T_0.\ {}^{\gamma} T_1}\ (\mathcal{E}/\textbf{et.fun}) \qquad \frac{\Gamma \vdash E_1 :^{\gamma_1} \Pi x : T_0.\ {}^{\gamma_2} T \qquad \Gamma \vdash E_0 :^{\mathtt{P}} T_0}{\Gamma \vdash E_1\, E_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow E_0\} T}\ (\mathcal{E}/\textbf{et.app})$$

Local binding In the rule ($\mathcal{E}$/et.app), the argument E_0 must be pure. In order to lift this restriction, one may use a local binding construct instead, as discussed in section IV.2.1.4. One may then no longer substitute E_0 in the result type, thus a premise of ($\mathcal{E}$/et.let) imposes that the result type does not mention the locally bound variable x — T is the type of the whole expression as well as the type of the body E. The effects of the expression are the union of that of E_0 and E. Since we only distinguish between two effects, and the case where both E_0 and E is not useful as function application can be used instead, we directly state the rule assuming the impure effect $\mathtt{I}$ throughout. With a richer effect system, we would use the least upper bound of the effects of E_0 and E as the result effect, although we would impose that this bound not be $\mathtt{P}$ in order to keep the pure fragment of the language as small as possible.

$$\frac{\Gamma \vdash E_0 :^{\mathtt{I}} T_0 \qquad \Gamma, x : T_0 \vdash E :^{\mathtt{I}} T \qquad \Gamma \vdash T\,\mathsf{ok}}{\Gamma \vdash (\mathsf{let}\, x = E_0\, \mathsf{in}\, E : T) :^{\mathtt{I}} T}\ (\mathcal{E}/\textbf{et.let})$$

If E is an impure expression of type $T_1 * T_2$, the second projection of E can be encoded as $\mathsf{let}\, x = E\, \mathsf{in}\, \pi_2 x : T_2$. If E_0 is an impure expression and E has the type T_1, then the application of E to E_0 can be encoded as $\mathsf{let}\, x = E_0\, \mathsf{in}\, E\, x : T_2$. Note that in both cases T_2 is not allowed to contain x: the type of E may not be dependent.

Subtyping The rule ($\mathcal{E}$/et.sub) combines implicit subtyping with implicit subeffectuation: any expression whose effects are constrained by γ has its effects constrained by any γ' such that $\gamma \sqsubseteq \gamma'$.

[4]Actually it would suffice to require E_1 to be pure, with effects allowed in E, however we will not have a use for this generalisation.

$$\frac{\Gamma \vdash E :^{\gamma} T \qquad \Gamma \vdash T <: T' \qquad \text{when } \gamma \sqsubseteq \gamma'}{\Gamma \vdash E :^{\gamma'} T'}\ (\mathcal{E}/\mathsf{et.sub})$$

Singletons We qualify the rule ($\mathcal{E}$/et.sing) to restrict singletons to pure expressions.

$$\frac{\Gamma \vdash E :^{\mathtt{P}} T}{\Gamma \vdash E :^{\mathtt{P}} S(E)}\ (\mathcal{E}/\mathsf{et.sing})$$

Sealing A new rule describes how to type a sealing. In general, in the expression $E \mathbin{!!} T$, the "natural" type for E is a subtype of T, and the subtyping rule must be applied. The effects of $E \mathbin{!!} T$ are those of E, plus the effect of the sealing; since we do not distinguish between effects, the effect annotation on $E \mathbin{!!} T$ is always $\mathtt{I}$.

$$\frac{\Gamma \vdash E :^{\gamma} T}{\Gamma \vdash (E \mathbin{!!} T) :^{\mathtt{I}} T}\ (\mathcal{E}/\mathsf{et.seal})$$

IV.4.4 Applicativity

IV.4.4.1 Applicative functors

Consider a functor whose body is sealed, i.e., $\lambda x : T_0.\ (E \mathbin{!!} T)$. According to our description of sealing, each application of this functor causes the expression $E \mathbin{!!} T$ to be evaluated, producing a fresh batch of abstract types. Therefore the types $\mathsf{Typ}\,\pi_1\,x_1$ and $\mathsf{Typ}\,\pi_1\,x_2$ in the following program are incompatible:

$$\begin{array}{l}\mathsf{let}\ f = \lambda x : \text{UNIT}.\ ((\langle T' \rangle, E) \mathbin{!!} \Sigma x : \text{TYPE}.\ T_1)\ \mathsf{in} \\ \mathsf{let}\ x_1 = f\,()\ \mathsf{in}\ \mathsf{let}\ x_2 = f\,()\ \mathsf{in}\ \ldots\end{array}$$

This means that f is a **generative functor** (see section I.2.2.3). A functor whose body is sealed is always generative; this is reflected by its type, which has the form $\Pi x : T_0.\ ^{\mathtt{I}}T$ indicating that the application of the functor has a side effect (namely type creation). On the other hand, a functor whose body is pure (thus in particular not sealed) has a type of the form $\Pi x : T_0.\ ^{\mathtt{P}}T$, and does not participate in creating abstraction: it is a **transparent functor**.

Sometimes we would like for a functor to create abstraction, but for repeated applications of the functor to the same arguments to produce compatible abstract types. A typical example is a functor creating a data structure, where the arguments describe the elements of the data structure. Such functors are called **applicative functors** [Ler95]. We shall see two ways of supporting applicative functors.

One method is to add a new notion of sealing to the language, such that sealing the same module twice yields compatible results (in the sense that their abstract types are equivalent). This notion of sealing is called **weak sealing**, and we will write a weak sealing as $E :: T$. The form of sealing which always creates fresh abstract types, which we already know as $E \mathbin{!!} T$, is called **strong sealing**. We will see how to formalise weak sealing in section IV.4.4.2. Using it, the types $\mathsf{Typ}\,\pi_1\,x_1$ and $\mathsf{Typ}\,\pi_1\,x_2$ in the following program are compatible:

$$\begin{array}{l}\mathsf{let}\ g = \lambda x : \text{UNIT}.\ ((\langle T' \rangle, E) :: \Sigma x : \text{TYPE}.\ T_1)\ \mathsf{in} \\ \mathsf{let}\ x_1 = f\,()\ \mathsf{in}\ \mathsf{let}\ x_2 = f\,()\ \mathsf{in}\ \ldots\end{array}$$

There is actually another way to build applicative functors without extending the language. One may seal the functor itself, rather than its body. Then all abstraction happens when the functor is defined, and none when it is applied. For example, in the following program, g is an applicative functor, and the types $\mathsf{Typ}\,\pi_1\,y_1$ and $\mathsf{Typ}\,\pi_1\,y_2$ are compatible (we assume that E is pure).

$$\begin{array}{l}\mathsf{let}\ g = (\lambda x : \text{UNIT}.\ (\langle T' \rangle, E))\ !!\ (\Pi x : \text{UNIT}.\,{}^{\mathsf{P}}\Sigma x : \text{TYPE}.\ T_1)\ \mathsf{in} \\ \mathsf{let}\ y_1 = g\,()\ \mathsf{in}\ \mathsf{let}\ y_2 = g\,()\ \mathsf{in}\ \ldots\end{array}$$

In either case, an applicative functor has a pure functor type $\Pi x : T_0.\,{}^{\mathsf{P}}T_1$, as opposed to the impure functor type $\Pi x : T_0.\,{}^{\mathsf{I}}T_1$ of a generative functor. This does not prevent an applicative functor from creating abstract types, if the result signature T_1 contains TYPE fields. The method for defining an applicative functor using strong sealing conveys an interesting idea about how abstract types are created: the side effect of creating the abstract types happens when the functor is defined, or more precisely when the functor is sealed, making an applicative functor from a transparent functor. If the functor is defined and sealed at the top level of the program, the side effect happens during program initialisation.

An applicative functor can be transformed into a generative functor at any time by sealing it to the appropriate generative functor signature (just like a transparent functor can be made applicative): $\Pi x : T_0.\,{}^{\mathsf{P}}T_1$ is a subtype of $\Pi x : T_0.\,{}^{\mathsf{I}}T_1$ (this is contained in the subtyping rule ($\mathcal{E}$/tsub.cong.fun)). The converse transformation, of an abstraction-creating functor to a less-abstracting functor (generative to applicative, or applicative to transparent), is undesirable, since there would be a loss of abstraction. Our effect system prevents such loss: applying a generative functor triggers an effect, and the only way to hide this effect is to wrap it in a lambda-abstraction whose type records the effect.

IV.4.4.2 Static sealing: formalisation $\boxed{\mathcal{W}}$

Introduction We shall define a new, "weak" notion of sealing such that sealing the same module twice produces compatible results. The purpose of this notion is to define applicative functors, and these provide a good way to understand weak sealing. We have seen how to seal a transparent functor to make it applicative: there is then an effect when the applicative functor is defined. The family of abstract types created by an applicative functor (which is indexed by the arguments of the functor) is fixed once and for all. We mentioned earlier that when encoding applicative functors using strong sealing, the effect happens during program initialisation. In fact the effect of weak sealing can be considered to happen at compile-time (more precisely during type-checking). Such a weak sealing is called **static sealing**, as opposed to the **dynamic sealing** $\mathsf{E}\ !!\ \mathsf{T}$. (We will discuss other weak forms of sealing in section IV.4.4.4). Let us now formally define static sealing.

Syntax We define system $\mathcal{W}$ which extends system $\mathcal{E}$. The expression language has the new static sealing construct $\mathsf{E} :: \mathsf{T}$. Local binding now carries an effect annotation, whose meaning we will explain when commenting on the typing rule ($\mathcal{W}$/et.let); we will often omit this extra annotation in examples where it does not matter. The type language remains unchanged.

$E ::=$	**expression (module)**
$\quad\ldots$	
$\quad \mathsf{let}\ x = E_0\ \mathsf{in}\ E :^{\gamma} T$	local binding
$\quad E :: T$	static sealing

The main novation is that the effect system is now larger.

$\gamma ::=$	**effect**
`P`	pure
`I`	dynamic effect
`S`	static effect
`IS`	both a static and a dynamic effect

The order relation on effects is given by $\mathtt{P} \sqsubseteq \mathtt{I} \sqsubseteq \mathtt{IS}$ and $\mathtt{P} \sqsubseteq \mathtt{S} \sqsubseteq \mathtt{IS}$. Only `P` and `I` may appear in function types, i.e., as the γ in $\Pi x : T_0.\ ^{\gamma}T_1$.

The type system of $\mathcal{W}$ is mostly identical with $\mathcal{E}$: the differences are confined to a few rules that feature impure expressions. Rules that are parametric over an effect instantiation may have the effect instantiated by `S` or `IS`, with (for ($\mathcal{W}$/et.sub)) the extended order relation.

Typing a static sealing is similar to typing a dynamic sealing. In either case, the effect of the sealing is added to the effects of the function body. The effect of the sealing is either `I` for dynamic sealing or `S` for static sealing.

$$\frac{\Gamma \vdash E :^{\gamma} T}{\Gamma \vdash (E \mathbin{!!} T) :^{\gamma \sqcup \mathtt{I}} T}\ (\mathcal{W}\textbf{/et.seal.dyn}) \qquad \frac{\Gamma \vdash E :^{\gamma} T}{\Gamma \vdash (E :: T) :^{\gamma \sqcup \mathtt{S}} T}\ (\mathcal{W}\textbf{/et.seal.stat})$$

Lambda-abstraction hides dynamic effects (although the type of the abstraction remembers the effect), but static effects always remain apparent. The functor result effect $\gamma \sqcap \mathtt{S}$ can be seen as "the static part of γ", while the effect of the whole functor $\gamma \sqcup \mathtt{I}$ is the dynamiac part of γ. We recall the rule for application, which is unchanged, but do note that $\gamma_2 \sqsubseteq \mathtt{I}$ always holds.

$$\frac{\Gamma, x : T_0 \vdash E :^{\gamma} T_1}{\Gamma \vdash \lambda x : T_0.\ E :^{\gamma \sqcap \mathtt{S}} \Pi x : T_0.\ ^{\gamma \sqcap \mathtt{I}} T_1}\ (\mathcal{W}\textbf{/et.fun}) \qquad \frac{\Gamma \vdash E_1 :^{\gamma_1} \Pi x : T_0.\ ^{\gamma_2} T \qquad \Gamma \vdash E_0 :^{\mathtt{P}} T_0}{\Gamma \vdash E_1\, E_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow E_0\} T}\ (\mathcal{W}\textbf{/et.app})$$

In system $\mathcal{E}$, we forced local binding expressions $\mathsf{let}\ x = E_0\ \mathsf{in}\ E : T$ to be impure, in order to keep the set of pure, hence comparable expressions small (as soon as E_0 is pure the expression can be written $(\lambda x{:}T_0.\ E)\, E_0$). We will keep doing this here; however we cannot simply force the expression to have the effect `I` since there are now other effects: if the expression should have the effect `S`, we do not want to force it to `IS` (nor "forget" the static effect and only keep `I`). We now require that the programmer specify the overall effect along with the type (because of the avoidance problem); since we do not want the expression to be pure we simply forbid this effect annotation from being `P`.

$$\frac{\Gamma \vdash E_0 :^{\gamma} T_0 \qquad \Gamma, x : T_0 \vdash E :^{\mathtt{I}} T \qquad \Gamma \vdash T\ \mathsf{ok} \qquad \text{when } \gamma \neq \mathtt{P}}{\Gamma \vdash (\mathsf{let}\ x = E_0\ \mathsf{in}\ E :^{\gamma} T) :^{\gamma} T}\ (\mathcal{W}\textbf{/et.let})$$

Execution The dynamic semantics of system $\mathcal{W}$ is the same as that of $\mathcal{E}$, with the rule ($\mathcal{W}$/ered.seal) indifferently accepting a dynamic or static sealing. Since this reduction simply erases the abstraction, the degree of generativity does not matter.

IV.4.4.3 Equivalences in the presence of static sealing

We saw in section IV.4.4.1 that dynamic sealing does not commute with functor abstraction: $\lambda x : T_0.\ (E \mathbin{!!} T)$ and $(\lambda x : T_0.\ E) \mathbin{!!} (\Pi x : T_0.\ ^{\gamma}T)$ are not equivalent — if $\gamma = \mathtt{I}$, they have the same type (they are both generative functors), but the first expression is pure while the second is impure. In contrast, static sealing commutes with functor abstraction: the expressions $E_1 = \lambda x : T_0.\ (E :: T)$ and $E_2 = (\lambda x : T_0.\ E) :: (\Pi x : T_0.\ ^{\gamma}T)$ can be used interchangeably. ML (or at least Objective Caml) programmers often take this equivalence for granted. We will say that E_1 and E_2 are **equitypable**, meaning that for any Γ, γ' and T', the typing judgement $\Gamma \vdash E_1 :^{\gamma'} T'$ is derivable if and only if $\Gamma \vdash E_1 :^{\gamma'} T'$ is. Equitypability will be the notion of interest in the present section, as we will look at program fragments that have the same run-time behaviour apart from the usage of sealing, but are distinguishable at the typing level as they create abstraction in different amounts.

To prove that E_1 and E_2 are equitypable, first note that both E_1 and E_2 require $\Gamma, x : T_0 \vdash E :^{\gamma} T$ to hold in order for each of them to be well-typed (by case analysis on their potential typing derivations). Then E_1 is typable by the rule ($\mathcal{W}$/et.seal.stat) followed by ($\mathcal{W}$/et.fun), while E_2 is typable by the rule ($\mathcal{W}$/et.fun) followed by ($\mathcal{W}$/et.seal.stat) (extra applications of ($\mathcal{W}$/et.sub) may be inserted, but they do not have significant impact as all operations involved are covariant in the result type). Both E_1 and E_2 have the principal type $\Pi x : T_0. {}^{\gamma \sqcap \mathtt{I}} T$ and the principal effect $\mathtt{S}$.

Static sealing also commutes with other constructs. For example, $(E_1, E_2) :: (T_1 * T_2)$ is equitypable with $(E_1 :: T_1, E_2 :: T_2)$, as well as with $(E_1 :: T_1, E_2)$ and $(E_1, E_2 :: T_2)$, assuming that each E_i has the type T_i. The key reason is that the presence of static sealing in any position makes the whole pair statically impure. Similarly the expressions $\pi_i (E :: T_1 * T_2)$ and $(\pi_i E) :: T_i$ are equitypable when E has the type $T_1 * T_2$.

Let us now consider a local binding $E' = \mathsf{let}\, x = E_0 \,\mathsf{in}\, E :^{\gamma} T$. Sealing only E is manifestly not equivalent to sealing E', since the set of signatures that E or E' may be sealed to is different: only E may be sealed by a type mentioning x. However the difference is but of little importance, the reason being that the influence of sealing on effects is the same in both cases, viz., introducing $\mathtt{S}$ if sealing statically, or introducing $\mathtt{I}$ if sealing dynamically. As for the type of the expression, it remains T if E is sealed (assuming the whole expression remains well-typed), and it becomes some subtype of T is E' is sealed. In particular, $\mathsf{let}\, x = E_0 \,\mathsf{in}\, (E :: T_1) : T$ is equitypable with $(\mathsf{let}\, x = E_0 \,\mathsf{in}\, E :^{\gamma} T) :: T$ as long as E has the type T_1 in the appropriate environment.

A sealing (whether dynamic or static) cannot appear in a function argument. These considerations show us that we do not lose expressivity if we limit the presence of static sealing inside a program to just two kinds of places: on a locally bound module $\mathsf{let}\, x = (E_0 :: T_0) \,\mathsf{in}\, E :^{\gamma} T$ and on an applied functor $(E_1 :: T_2)\, E_0$. In the first case, removing the sealing could allow E to make use of a more precise type for x (any type of E_0, not limited by E_0) — in other words the abstraction provided by the sealing would vanish with it. The usefulness of sealing an applied functor is of the same order: in order for $(E_1 :: T_2)\, E_0$ to be well-typed, T_2 must be a function[5] $\Pi x : T_0. {}^{\gamma} T_1$. If T_1 is smaller than necessary, the abstraction could migrate above the application (one could write $(E_1\, E_0) :: \{x \leftarrow E_0\} T_1$ instead). If T_0 is larger than necessary, the choice of possible types of E_0 is limited, so E_0 becomes more abstract than as seen by the function. In fact, $(E_1 :: \Pi x : T_0. {}^{\gamma} T_1)$ is equitypable with $\mathsf{let}\, x = (E_0 :: T_0) \,\mathsf{in}\, E_1\, x :^{\gamma} T_1$. Hence, in summary, *static sealing is only useful on a locally bound module.*

IV.4.4.4 Other forms of sealing

We have so far formalised two forms of sealing: static sealing, which creates a new family of abstract types for each syntactic occurrence of the sealing operator, and dynamic sealing, which creates a new family of abstract types each time the sealing operator is evaluated. These two forms correspond to the *weak sealing* $E :: T$ and *strong sealing* $E :> T$ proposed by Dreyer, Crary and Harper [DCH03], and our effect system follows theirs[6] (with one minor difference: they declare strong sealing as having a static effect as well as a dynamic effect, which uselessly strict but of little incidence).

Dreyer [Dre05] distinguishes between three forms of sealing:

[5] T_2 could actually also be a singleton, but then the sealing would not be creating any abstraction.

[6] We use different notations however: we see effect annotations as indicating effects, whereas they see these annoations as purity annotations; thus we write `S` for a static effect where they write `D` for dynamic purity, and we write `I` for a dynamic effect where they write `S` for static purity.

- impure sealing $\mathsf{impure}(\mathsf{E} :> \mathsf{T})$ is the strong sealing of Dreyer, Crary and Harper [DCH03] mentioned in the previous paragraph;

- separable sealing $\mathsf{E} :> \mathsf{T}$ and inseparable sealing $\mathsf{pure}(\mathsf{E} :> \mathsf{T})$ both correspond to our static sealing, being only distinguished by their separability, which we do not take into account (see section IV.4.2.3).

Many other variants can be conceived, with varying strengths. **Ascription** consists in constraining an expression to a type without restricting the view to it, i.e., without introducing abstraction: if E has the type T, the ascription $\mathsf{E} :_{\mathrm{a}} \mathsf{T}$ has any type that E has; in particular, if E is pure, then $\mathsf{E} :_{\mathrm{a}} \mathsf{T}$ is also pure and has the type $\mathsf{S}(\mathsf{E})$. Ascription can be seen as a degenerate form of sealing.

Minimal sealing creates a comparable abstract type: $\mathsf{E} :_{\mathrm{s}} \mathsf{T}$ has the type T and the purity of E. Thus minimal sealing is a new way to construct pure expressions; two minimally sealed expressions are comparable. Minimal sealing cheerfully generates coincidental type equivalences, whenever the same expression happens to be sealed to the same type. A vairant of minimal sealing consists in declaring $\mathsf{E} :_{\mathrm{s}} \mathsf{T}$ to be equivalent to $\mathsf{E} :_{\mathrm{s}} \mathsf{T}'$ whenever both are valid. Yet another variant consists in having a whole family of minimal sealings indexed by a name, considering two minimal sealings of the same expression to be equivalent if and only if they carry the same name.

Note that minimal sealing can be emulated using static sealing. All we need is a standard library function providing an applicative functor

$$\begin{aligned} f_{\mathtt{minseal}} = {} & \lambda t : \text{TYPE}.\ \lambda x : \text{STRING}. \\ & ((t, ((\lambda x : \mathsf{Typ}\, t.\ x), (\lambda x : \mathsf{Typ}\, t.\ x))) :: \\ & \ \Sigma t' : \text{TYPE}.\ (\mathsf{Typ}\, t \to \mathsf{Typ}\, t') * (\mathsf{Typ}\, t' \to \mathsf{Typ}\, t)) \end{aligned}$$

or in Objective Caml syntax

```
let MinSeal = functor (A : sig type t end) ->
  struct  type t = A.t  let a x = x     let c x = x     end :
  sig     type t        val a : t->A.t  val c : A.t->t  end
end
```

For any type T and any name x, $f_{\mathtt{minseal}}\,\langle\mathsf{T}\rangle\, x$ provides an abstract type and conversion functions between that type and T. (In Objective Caml, we should define once `module` $\mathtt{M}_{\mathsf{T}}^{name}$ `= struct type t = T end` for every type T and name *name* since structures are generative.) This defines the named variant of minimal sealing; getting rid of x yields the basic variant. As we remarked in section IV.4.4.1, $f_{\mathtt{minseal}}$ could equally well be defined using dynamic sealing.

We have limited our exposition to two forms of sealing because, together with the easily definable minimal sealing, they seem to be sufficient for all practical purposes. We can roughly partition uses of sealing into three categories:

- abstract datatypes, in which abstraction enforces algebraic properties that go beyond the expressive power of the type system: static sealing is the perfect match;

- named variants of isomorphic types (e.g., `dollar` and `euro`): minimal sealing is suitable;

- abstract types used to limit access to some resource, which require dynamic sealing.

IV.4.4.5 Mutual encodings of static and dynamic sealing

Can we go even further and be content with a single form of sealing? The answer is "sort of": while static and dynamic sealing can be encoded in terms of one another, a global program transformation is required either way.

Let us first express static sealing in terms of dynamic sealing. We start with two observations. Firstly, the two forms of sealing are equivalent if the sealing is executed exactly once. Secondly, we saw in section IV.4.4.3 that it suffices to study static sealing on locally bound expressions $\mathsf{let}\, x = E_0 :: T_0 \,\mathsf{in}\, E :^{\gamma} T$.

We can see any program as a sequence of local bindings $\mathsf{let}\, x_1 = E_1 :: T_1 \,\mathsf{in}\, \ldots \mathsf{let}\, x_k = E_k :: T_k \,\mathsf{in}\, E$ (we omit the return type on lets as they are not important here). We call E the body of the program, and the exposed local bindings are said to be toplevel. A program is said to be in *prenex sealings form* if none of the expressions $E_1, \ldots, E_k$ contain any static sealing. Finally a *clean* program is one in prenex sealings form where E contains no static sealing either, i.e., all static sealings are toplevel. In a clean program, static sealings can be replaced by dynamic sealings without affecting the typing of the program. We will show how to transform any program into an equitypable clean program.

Le $E_0 :: T_0$ be a subexpression of the program, so that the body of the program is $E_0 :: T_0$ in some context C, which we write $E = C \cdot (E_0 :: T_0)$. We can replace $E_0 :: T_0$ by $E'_0 = (\lambda y_1 {:} T_1. \;\ldots\; \lambda y_j {:} T_j.\, E_0 :: T_0 \ldots)\, y_1 \ldots y_j$ where $y_1, \ldots, y_j$ are the variables bound by C from outermost to innermost, omitting toplevel bindings. If $j = 0$ we instead take $j = 1$, $T_1 = \textsc{unit}$ and $E'_0 = (\lambda y {:} \textsc{unit}.\, E_0 :: T_0)\,()$. In every case, E'_0 is the application of a lambda-abstraction to one or more parameters. All free variables in this lambda-abstraction are bound at the toplevel. We can therefore extract it out of the context C in order to bind it above, going by beta-expansion and let lifting from $E = C \cdot (E_0 :: T_0)$ to $E' = (\mathsf{let}\, f = E'_0 \,\mathsf{in}\, C \cdot (f\, y_1 \ldots y_j))$ where f is a fresh variable. Now, as we saw earlier, static sealing can be lifted out of a lambda-abstraction, so E'_0 is equitypable with some expression $E''_0 :: T''_0$. Let $E'' = (\mathsf{let}\, f = E''_0 :: T''_0 \,\mathsf{in}\, C \cdot (f\, y_1 \ldots y_j))$. Provided E_0 itself contains no static sealing, E'' is in prenex sealings form, and the number of toplevel bindings has increased by 1.

Iterating the transformation we have just described over all of the static sealings in the initial program (from inside out), we can put any program in prenex sealings form. By replacing each toplevel static sealing with a dynamic one, we obtain an equitypable program that does not use static sealing. One intuitive view of this transformation is that each static sealing creates a single family of abstract types, and we lift the creation of this family to be performed exactly once during program initialisation.

We now turn to the dual problem of encoding dynamic sealing into static sealing. The difference between static sealing and dynamic sealing is the effect of the construction. One way to force a dynamic effect is to apply a generative functor, and a generative functor can be created by any sealing, including static sealing, of a transparent functor. This leads Dreyer, Crary and Harper [DCH03] to propose the following encoding of strong sealing (which is almost our dynamic sealing) into weak sealing (identical to our static sealing):

$$E :> T = ((\lambda x : \textsc{unit}.\; E) :: (\Pi x : \textsc{unit}.\, {}^{\mathtt{I}}T))\,()$$

Unlike dynamic sealing $E \;!!\; T$, strong sealing $E :> T$ has a static effect in addition to its dynamic effect: $E :> T$ is equitypable with $E \;!!\; T :: T$. This effect cannot be discharged by a lambda-abstraction, so that generative functors cannot be pure in Dreyer, Crary and Harper's system. Nonetheless we can move the extra static sealings to the toplevel by applying the transformation above.

In summary, static and dynamic sealing can be encoded in terms of one another, albeit incurring the cost of a global transformation. In the remainder of this document, we will study a language with only dynamic sealing (which we prefer due to its more compositional semantics). We propose to treat static sealing as an additional construct which should be provided in a programming language alongside dynamic sealing, and then elaborated away inside the compiler.

IV.4.4.6 On applicativity through functor sealing

The observation that the position of the sealing determines whether a functor is applicative or generative does not seem to be universally known in the ML community. It requires being able to seal a functor, whereas early module systems for ML only had sealing of structures. In Objective Caml, where functors are systematically applicative, it is customary to seal the body of a functor, and sealing the functor is considered equivalent (and needlessly complex as the type of the argument is then repeated) [Ler]. Note that if sealing is interpreted as an effect, the fact that it does not commute with lambda-abstraction is unsurprising.

Early module systems for ML only defined sealing on structures, and the possibilities of functor sealing seeped in slowly and with a low profile. Russo [Rus98] distinguishes applicative functors from generative functors by their definition rather than by their signature, with the defect that a generative functor can be directly seen as an applicative functor [Dre02] as seen in section I.2.2.4. Shao [Sha99] remarks in passing that sealing a transparent functor is a way of building an applicative functor. This possibility is also mentioned by Dreyer, Crary and Harper [DCH03, Dre05] but they recommend using weak sealing to build applicative functors.

One argument in favor of weak sealing ([DCH03], §2, p. 7) is that it can be applied to a single member of a structure in the body of the functor, which makes some types abstract already in following members, whereas sealing the functor only makes types abstract once the functor is applied. In our notation, the functors under discussion are of the form $\lambda x{:}T_0.\,\mathsf{let}\,y = E_1 :: T_1\ \mathsf{in}\ E_2 : T_2$. We saw in section IV.4.4.5 that this is precisely the case when the transformation of static sealing into dynamic sealing requires a global code reorganisation.

Dreyer [Dre05] (§1.2.7) mentions in particular the case of a functor whose body defines and uses a "generative" declared type (`datatype` in Standard ML, ordinary variant or record type in Objective Caml). If declared types are modelled by an abstract type obtained through dynamic sealing, such a functor is automatically generative. However we do not see any reason to insist on dynamic sealing: minimal sealing would do just fine, as the generative nature of declared types only serves to differentiate between the constructor and destructor names of different declared types. Since minimal sealing can easily be modelled by dynamic sealing, the lack of sealing other than dynamic is not a problem on this count.

IV.5 Colours and brackets $\boxed{\mathcal{C}}$

In system $\mathcal{E}$, the sealing construct affects typing but not evaluation, as witnessed by the reduction rule for sealing which just forgets the sealing:

$$\mathsf{E}\ !!\ \mathsf{T} \longrightarrow \mathsf{E} \qquad (\mathcal{E}/\mathsf{ered.seal})$$

This rule is type-preserving in the sense that if the left-hand side is well-typed then the right-hand is also well-typed and has the same type. However information is clearly lost: this rule is not abstraction-preserving. This lack is no concern when evaluating a single program, as type preservation ensures that nothing can "go wrong" and the whole program text is available for any

If T is of the form ...	then $\mathsf{E} \mathbin{!!} \mathsf{T}$ reduces to an expression of type...
$\Sigma \mathsf{t} : \mathrm{TYPE}.\ (\mathrm{INT} \to \mathsf{Typ}\,\mathsf{t})$	$\mathsf{S}(\pi_1\,a) * (\mathrm{INT} \to \mathsf{Typ}\,\pi_1\,a)$
$\Sigma \mathsf{t} : \mathrm{TYPE}.\ \Sigma \mathsf{t}' : \mathrm{TYPE}.\ (\mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}' * \mathrm{INT})$	$\mathsf{S}(\pi_1\,a) * \mathsf{S}(\pi_1\,\pi_2\,a) * \mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}' * \mathrm{INT}$
$\Pi x : \mathsf{T}_0.\ \Sigma \mathsf{t} : \mathrm{TYPE}.\ (\mathsf{Typ}\,\mathsf{t} * \mathrm{INT})$	$\Pi x : \mathsf{T}_0.\ \mathsf{S}(\pi_1\,(a\,x)) * \mathsf{Typ}\,\mathsf{t} * \mathrm{INT}$

In each case, a is the nonce (fresh module identity) created by the sealing operation.

Figure IV.1: Examples of nonce generation

analysis that might rely on the typing of the program. However we aim to rid ourselves of the strict phase separation between type-checking and evaluation, by introducing a facility for type-checking. This facility requires additional information to remember the distinction between an abstract type and its representation type as long as it matters, which is as long as dynamic type-checking might be performed, i.e., throughout program evaluation. We will now study system $\mathcal{C}$, which is based on $\mathcal{E}$ but where the reduction of a sealing preserves the abstraction.

IV.5.1 Module identities

IV.5.1.1 Nonce generation

The rule ($\mathcal{E}$/ered.seal) does not properly reflect our intention regarding the semantics of sealing. We described sealing as creating a new type. Consider the example sealing expression $(\langle \mathrm{INT} \rangle, 3) \mathbin{!!} \Sigma \mathsf{t} : \mathrm{TYPE}.\ \mathsf{Typ}\,\mathsf{t}$. It reduces by ($\mathcal{E}$/ered.seal) to $(\langle \mathrm{INT} \rangle, 3)$, which has the type $\Sigma \mathsf{t} : \mathsf{S}(\langle \mathrm{INT} \rangle).\ \mathsf{Typ}\,\mathsf{t} = \mathsf{S}(\langle \mathrm{INT} \rangle) * \mathrm{INT}$, whereas we would like a type of the form $\Sigma \mathsf{t} : \mathsf{S}(\langle \mathsf{T}' \rangle).\ \mathsf{Typ}\,\mathsf{t} = \mathsf{S}(\langle \mathsf{T}' \rangle) * \mathsf{T}'$ where T' is distinct from any previously existing type (especially INT).

More precisely, we do not need to create a type but a *module identity*, as can be seen by looking at the sealing of modules with a more complex structure. For example, if the same sealed module defines several abstract types, these types share a common unique identity. If the sealed module is a functor, a new identity must be created but once when the sealing construct is evaluated, and this identity will be used each time the functor is applied. Each module identity characterises one instantiation of the abstraction, which may produce any number of fresh types: as many as there are abstract type fields for a structure, an unbounded number for a functor (since the argument must be taken into account)...

Figure IV.1 shows a few examples of uses of module identities. A fresh module identity is called a **nonce** (or (h)apax), and written a. These nonces have the same universal uniqueness property as those used in models of security. They generalise the *stamps* of MacQueen [AM91]. Unlike the stamps of many module systems, our nonces can designate modules of arbitrary signatures (in particular functors). Also the creation of a nonce is performed when the sealing construct is evaluated, and not whenever a module is built. Nonces correspond to the singularised identities described in section II.6.1.2.

At the syntactic level, we will for the time being consider nonces an extra construct in the syntax of modules, which should not appear in source programs[7]. However the purpose of nonces is to designate abstract types, and we will eventually restrict their presence to "type components" (see section IV.5.1.4). We assume an infinite supply of distinct nonces (similar to the infinite supply of variable names).

[7]Note that (as we shall see) nonce-free programs have nonce-free types, so types of source programs are expressible in the source language.

IV.5.1.2 Lexes

Evaluating a sealing construct $\mathsf{E} \mathbin{!!} \mathsf{T}$ requires a fresh nonce, i.e., one that is not present in the original term. We manage this freshness requirement by using a store of nonces, called a **lexis** (or stamp book), and written B. A lexis keeps track of nonces in use as well as the module implementation and signature from the sealing construct from which the nonce originated. The syntax of a lexis is thus

$$\mathsf{B} = (\mathsf{a}_1 = \mathsf{E}_1 : \mathsf{T}_1, \dots, \mathsf{a}_k = \mathsf{E}_k : \mathsf{T}_k)$$

As with environments, we treat lexis concatenation as associative, and the empty lexis (written nil) is a neutral element for this operation.

Lexes adorn evaluation judgements as well as typing judgement. A nonce a is well-typed and has type T when the ambient lexis contains the binding $\mathsf{a} = \mathsf{E} : \mathsf{T}$ for some E (just as a variable x has the type T when the environment contains the binding $\mathsf{x} : \mathsf{T}$). The reduction relation for system $\mathcal{C}$ is formally defined on pairs consisting of a lexis and a type; a reduction will be written $\mathsf{B} \vdash \mathsf{E} \longrightarrow \mathsf{B}' \vdash \mathsf{E}'$. However in most instances the lexis does not affect the reduction and does not change, and we will then continue to write $\mathsf{E} \longrightarrow \mathsf{E}'$. Reducing a sealing augments the lexis:

$$\mathsf{B} \vdash \mathsf{E} \mathbin{!!} \mathsf{T} \longrightarrow \mathsf{B}, \mathsf{a} = \mathsf{E} : \mathsf{T} \vdash \mathsf{E}'$$

The nonce a is chosen fresh, i.e., outside the domain of B. Since the language does not include binders for nonces, any nonce appearing in E or T must be recorded in B.

An alternative to this global store would be to introduce a “new” binder for nonces, classically writing $(\nu \mathsf{a} = \mathsf{E}_0 : \mathsf{T}_0)\mathsf{E}$. We prefer the use of a global store not only because we have no need for a nonce binder, but also because managing the migration of ν binders accross other syntax nodes and above environments[8] would be problematic.

IV.5.1.3 From sealing to brackets

We have seen that reducing a sealing $\mathsf{E} \mathbin{!!} \mathsf{T}$ creates a fresh nonce a, and the result is an expression E' of a type T', where T' uses the nonce a as the identity of the newly created module in order to precisely specify the abstract parts of T.

This type T' is called a **strengthening** of T, or more precisely it is what we will call the **selfification** of the type T for the module identity a (see section I.2.2.2). We will write this selfification as $\mathbf{self}^{\mathsf{T}}(\mathsf{a})$. The general idea behind selfification is to mirror the original structure of the type, but replace the parts originally left abstract by a reference to the newly created nonce. Figure IV.1 shows a few examples of selfification; we will defer the task of formulating a precise definition until section IV.5.1.5.

Sealing must transform the expression E of type T into an expression E' with essentially the same behaviour as E but a different type $\mathbf{self}^{\mathsf{T}}(\mathsf{a})$. This new type is more precise than T: it is a subtype of T. Although T is usually not the most precise type of E, E cannot have the type $\mathbf{self}^{\mathsf{T}}(\mathsf{a})$ in general: if the same expression is sealed twice, the resulting expressions E'_1 and E'_2 should have the respective incompatible types $\mathbf{self}^{\mathsf{T}}(\mathsf{a}_1)$ et $\mathbf{self}^{\mathsf{T}}(\mathsf{a}_2)$. Thus E' must contain a reference to the specific choice of nonce a.

The most obvious way to construct E' is to start with E and apply a type coercion to it: $\mathsf{E}' = \mathsf{coerce}\,\mathsf{E}\,\mathsf{to}\,\mathbf{self}^{\mathsf{T}}(\mathsf{a})$. However how this coercion should interact with the rest of the language is not obvious. What does an expression of the form $\mathsf{coerce}\,\mathsf{E}\,\mathsf{to}\,\mathsf{T}'$ reduce to? With such little information in a readily extractible form, when is $\mathsf{coerce}\,\mathsf{E}\,\mathsf{to}\,\mathsf{T}'$ well-typed?

[8]A nonce may appear in the type of a bound variable.

We were already confronted with this problem in the simpler setting of the HAT language, as discussed in III.1.1. We will use the same solution as then, to wit, **coloured brackets** [ZGM99]. The expression $[\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}$ denotes the coercion of E to the type $\mathbf{self}^{\mathsf{T}}(\mathsf{a})$, but records the justification a for equating the implementation E with the abstraction. (Recall that E is recorded in the entry for a in the ambient lexis.) More generally, if E_1 is any expression, and if T_2 is equivalent to the type of E_1 modulo the equivalence between a and its implementation, the expression $[\mathsf{E}_1]_{\mathsf{a}}^{\mathsf{T}_2}$ has type T_2. This expression is called the coloured bracket (or brackets) surrounding E_1 and annotated with the colour a and the type T_2.

The expression E_1 should be seen as *inside* the bracket, while the annotations a and T_2 are carried *by* the bracket. Nonces (or fresh module identities) a have the same rôle as the hashes (or structual module identities) h in HAT. The colour a indicates the possibility of using the extra typing equality between a and its implementation E inside the bracket. We say that a is transparent inside the bracket. We will discuss the syntax and semantics of colours in system $\mathcal{C}$ more fully in section IV.5.2.

The reduction rule for sealing is thus as follows:

$$\mathsf{B} \vdash \mathsf{E} \mathbin{!!} \mathsf{T} \longrightarrow \mathsf{B}, \mathsf{a} = \mathsf{E} : \mathsf{T} \vdash [\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}$$

As in HAT, the next reduction steps are devoted to pushing the coloured brackets towards the inside of E.

IV.5.1.4 Abstract types

In the expression $[\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}$, the nonce a can only appear in two positions: as the colour annotation on the bracket, and in building the type annotation on said bracket. It seems therefore possible to restrict the places where nonces may appear in the syntax. But do we have to?

Treating a nonce a as a full-fledged expression definitely simplifies the overall language structure. Under a lexis containing the binding $\mathsf{a} = \mathsf{E} : \mathsf{T}$, the expression a has the type T, and the transparency of a can be simply expressed by the conversion $\mathsf{a} \longrightarrow \mathsf{E}$.

Such uninhibited use of nonces nonetheless causes two problems, one theoretic, one practical. Both problems arise from reducing certain expressions containing nonces.

Consider for instance the sealing $\mathsf{E} \mathbin{!!} \mathsf{T}$ where $\mathsf{E} = (\langle \text{INT} \rangle, (2, 3))$ and $\mathsf{T} = \Sigma \mathsf{t} : \text{TYPE}. \, \mathsf{Typ}\, \mathsf{t} * \mathsf{Typ}\, \mathsf{t}$, resulting in the lexis binding $\mathsf{a} = \mathsf{E} : \Sigma \mathsf{t} : \mathsf{S}(\pi_1\, \mathsf{a}). \, \mathsf{Typ}\, \mathsf{t} * \mathsf{Typ}\, \mathsf{t}$. While it may be reasonable to treat such expressions as a, maybe even $\pi_1\, \mathsf{a}$, as values — although $\pi_1\, \mathsf{a}$ has a destructor at the head, which is odd in a value — the same does not go for $\pi_2\, \mathsf{a}$. Typing excepted, the expected behaviour of this expression is the same as $(2, 3)$. Evaluating $\pi_2\, (\mathsf{E} \mathbin{!!} \mathsf{T})$ yields the expression $[(2, 3)]_{\mathsf{a}}^{\mathsf{Typ}\, \pi_1\, \mathsf{a} * \mathsf{Typ}\, \pi_1\, \mathsf{a}}$. We might reduce a to $[\mathsf{E}]_{\mathsf{a}}^{\Sigma \mathsf{t} : \mathsf{S}(\pi_1\, \mathsf{a}). \, \mathsf{Typ}\, \mathsf{t} * \mathsf{Typ}\, \mathsf{t}}$ and then let bracket pushing do the rest; however, for type preservation, this expression must still have the original type $\mathsf{S}(\mathsf{a})$.[9]. Experience with a preliminary version of this system shows that obtaining such a typing requires a substantially more complex metatheory as analysing the types that are equivalent to $\mathsf{Typ}\, \pi_1\, \mathsf{a}$ becomes intractable.

On the practical side, such a reduction requires the implementation of a to be available at any time, whereas a nonce is intuitively an opaque piece of data on which only an equality predicate is defined. In particular, the implementation of a nonce might be cryptographically hidden, as we will see in section IV.5.2.1.

We saw that the principle of selfification is to mirror the original structure of the type while substituting the appropriate projection of the nonce for abstract type fields (TYPE) in the signature.

[9]Note that a must be pure, since it is meant for use in types.

Therefore the nonce a only appears in $\mathbf{self}^{\mathsf{T}}(a)$ as a projection of a field for which the signature indicates TYPE. Such a projection has the form $\mathsf{Typ}\,A$ where A may be a nonce, a pair projection $\pi_i\,A$, or the application of a functor to an argument $A\;\mathsf{E}_0$. A term A of the grammar we just describe shall be called a **module component**.

From now on, we will not allow a to be an expression, prefering to add a new entry to the syntax of types. A **component type** $(\!|A|\!)$ is a type, denoting what we would have written $\mathsf{Typ}\,A$. Where the expression A was desired, we can now write $\langle(\!|A|\!)\rangle$ (provided that the projection goes all the way to a single field of type TYPE).

Expressing the transparency of a nonce takes on a more complicated form since the equivalence $a \equiv \mathsf{E}$ (where E is the implementation of a) is no longer grammatical. Transparency must be expressed separately for each component type. Let a be the underlying nonce of the module component A_1 (which we shall write as $a = \mathbf{underl}(A_1)$). When a is transparent, $(\!|A_1|\!)$ is equivalent to $\mathsf{Typ}\,\mathsf{E}_1$ where E_1 is the projection of E with the same shape as A_1. We will say that A_1 is **revealed** as E_1, which we write $\mathsf{E}_1 = \mathbf{reveal}^{B}(A_1)$ (where B is the ambient lexis). For instance, if $\mathsf{T} = \Sigma t_1 : \text{TYPE}.\, \Sigma t_2 : \text{TYPE}.\, \Sigma t_3 : \mathsf{S}(\langle\text{INT}\rangle).\, \mathsf{T}_4$, then transparency of a entails the equivalences $(\!|\pi_1\,a|\!) \equiv \mathsf{Typ}\,\pi_1\,\mathsf{E}$ and $(\!|\pi_1\,\pi_2\,a|\!) \equiv \mathsf{Typ}\,\pi_1\,\pi_2\,\mathsf{E}$.

IV.5.1.5 Selfification

The core of the sealing operation consists in replacing the abstract components of a type by the corresponding projections of a certain nonce. Generally speaking, let us study the selfification of a type T for a module component A, written $\mathbf{self}^{\mathsf{T}}(A)$.

Base cases There are three kinds of elementary signatures: manifest type fields $\mathsf{S}(\langle\mathsf{T}_1\rangle)$ (`sig type t = T1 end`), abstract type fields TYPE (`sig type t end`), and term fields (`sig val x : T2 end`). The purpose of selfification is to transform abstract type fields into manifest type fields: the selfificaiton of TYPE by A is $\mathsf{S}(\langle(\!|A|\!)\rangle)$. Type fields that were already manifest (i.e., $\mathsf{S}(\langle\mathsf{T}_1\rangle)$) are left unchanged, as are term fields since no extra information is required. Type fields thus always gain singleton types, while term fields are unchanged.

Structures Selfifying a structure conserves its decomposition into fields, with each field selfified separately. For example, in ML notation,

```
sig
  type t1
  type t2
  type u = int * t1
  val f :  int -> t1 -> u
end
```

selfified by the name M is

```
sig
  type t1 = M.t1
  type t2 = M.t2
  type u = int * t1
  val f :  int -> t1 -> u
end
```

Note that the name of the module appears more than once: the same name M is used as part of the global designation of all the abstract types in the signature.

Selfifying a pair $\mathsf{T}_1 * \mathsf{T}_2$ by a component A naturally yields the pair $\mathbf{self}^{\mathsf{T}_1}(\pi_1 A) * \mathbf{self}^{\mathsf{T}_2}(\pi_2 A)$. A natural generalisation to dependent pairs is achieved by independently selfifying each component, thus $\mathbf{self}^{\Sigma x:\mathsf{T}_1.\,\mathsf{T}_2}(A) = \Sigma x : \mathbf{self}^{\mathsf{T}_1}(\pi_1 A).\, \mathbf{self}^{\mathsf{T}_2}(\pi_2 A)$.

It is tempting to try to go further: since x is now fully known, why not substitute it in T_2? Thus in the example above references to `t1` and `u` could become M.`t1` and M.`u` respectively. However this is not possible in our language, which includes signatures that cannot be expressed in ML — to wit, dependencies (and in particular equalities) on term fields. If the first component of a pair contains term fields, its signature does not become a singleton after selfification, and the effort to specialise the second component cannot proceed further. For example the selfification of $\Sigma x : \text{INT}.\, S(x)$ by a is $\Sigma x : \text{INT}.\, S(x)$, no more. The selfification of a dependent pair is therefore still dependent in general.

Functors The notation TYPE admits two radically different interpretations, depending on whether it is used in the argument or in the result of a functor. In the argument, TYPE denotes a type that will remain unknown until the functor is applied, and will vary from application to application: a functor with a TYPE-qualified argument is polymorphic. Thus selfifying a functor type $\Pi x : T_0.\, {}^{\gamma}T_1$ does not restrict the domain of the arguments of the functor[10]: the selfification has the form $\Pi x : T_0.\, {}^{\gamma}T_1'$. As for the interpretation of TYPE in the result type of the functor, it depends on whether the functor is applicative or generative. With an applicative type, the effective identity of the type field is fully determined by the argument passed to the functor, and selfification consists of eta-expanding the body as $\Pi x : T_0.\, {}^{\gamma}\mathbf{self}^{\,T_1}(A\,x)$. On the other hand, if the functor is generative, each application generates a new identity; the selfification operation must then be delayed until the application is performed, and T_1 must remain abstract for the duration.

Definition Selfification is defined by structural induction on the type as follows:

$$\begin{aligned}
\mathbf{self}^{\,BT}(A) &= BT && \text{if } BT \text{ is a base type } (\text{UNIT}, \text{BOOL}, \text{INT})\\
\mathbf{self}^{\,\Sigma x:T_1.\,T_2}(A) &= \Sigma x : \mathbf{self}^{\,T_1}(\pi_1 A).\, \mathbf{self}^{\,T_2}(\pi_2 A)\\
\mathbf{self}^{\,\Pi x:T_0.\,{}^{P}T_1}(A) &= \Pi x : T_0.\, {}^{P}(\mathbf{self}^{\,T_1}(A\,x))\\
\mathbf{self}^{\,\Pi x:T_0.\,{}^{I}T_1}(A) &= \Pi x : T_0.\, {}^{I}T_1\\
\mathbf{self}^{\,S(E')}(A) &= S(E)\\
\mathbf{self}^{\,\text{TYPE}}(A) &= S(\langle (\!| A |\!) \rangle)
\end{aligned}$$

Evaluation of type fields The table above does not state how to compute $\mathbf{self}^{\,\mathsf{Typ}\,E}(A)$. The reason for this omission is that the computation depends entirely in the value of E: the definition of selfification cannot be purely syntactic. Intuition whispers that the selfification of two equivalent types at the same identity should be equivalent; this implies for instance $\mathbf{self}^{\,\mathsf{Typ}\,\langle T\rangle}(A) = \mathbf{self}^{\,T}(A)$. Therefore E must be reduced to a value, which will have the form $\langle T\rangle$ for typing reasons, in order to compute the selfification of $\mathsf{Typ}\,E$.

IV.5.2 Colors

IV.5.2.1 Colouring

Coloured bracket According to their introduction in section IV.5.1.3, a coloured bracket $[E]_a^T$ lets the expression E be given the type T using the typing equations resulting from the knowledge of the implementation of a. As in HAT, we will have a wider view of coloured brackets as the syntactic manifestation of a boundary between the inside and the outside of the module. This boundary takes the form of the sealing construct in the source code, and coloured brackets embody it at run time.

[10]The selfification is a functor $\Pi x : T_0'.\, {}^{\gamma}T_1'$ per conservation of the overall structure.

Coloured syntax One way to describe the syntax of system $\mathcal{C}$ from the syntax of $\mathcal{E}$ would be to associate a colour to each node of the syntax. This colour would represent the origin of that node, i.e., from which sealed module the node comes from. The brackets provide this information in a different form: the colour of a syntax node is that carried by the innermost surrounding bracket.

In the absence of a surrounding bracket, the colour is the **ambient colour** carried by the jugdement in which the term under consideration is placed. The final form of an expression typing judgement in system $\mathcal{C}$ includes a colour annotation:

$$\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\gamma} \mathsf{T}$$

So does the final form of an expression reduction rule:

$$\mathsf{B} \vdash \mathsf{E} \longrightarrow_{\mathsf{c}} \mathsf{B}' \vdash \mathsf{E}'$$

Colours appear in other places in the syntax. The rule of thumb is that anywhere a type is attributed to an expression, a colour must also be provided. For instance a lexis binding has the form $(\mathsf{a} = \mathsf{E} :_{\mathsf{c}} \mathsf{T})$, and an environment binding has the form $(\mathsf{x} :_{\mathsf{c}} \mathsf{T})$.

Colour semantics In a typing judgement, the colour determines which abstract types may be revealed. Following the intuition outlined in section IV.5.1.4, they are the components of transparent nonces in the indicated colour. A reduction rule ($\mathcal{C}$/ered.colAbs) allows the revelation of transparent nonces.

Reduction of a sealing We can finally state the rule for sealing reduction in full. Let us first consider our usual example consisting of the module `struct type t = int let x = 3 end` sealed to the signature `sig type t val x :  t end`.

$$\begin{aligned} \mathsf{nil} \ \vdash \ (\langle \textsc{int} \rangle, 3) \mathbin{!!} (\Sigma \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t}) \quad &\longrightarrow_{\bullet} \\ \mathsf{a} = (\langle \textsc{int} \rangle, 3) :_{\bullet} (\Sigma \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t}) \ &\vdash \ [(\langle \textsc{int} \rangle, 3)]_{\mathsf{a}}^{\Sigma \mathsf{t} : \mathsf{S}(\langle (\!| \pi_1\, \mathsf{a} |\!) \rangle).\ \mathsf{Typ}\,\mathsf{t}} \end{aligned}$$

Subsequent reduction steps push the brackets towards the inside of the value, as in HAT (see section III.1.2.2).

In general a sealing $\mathsf{V} \mathbin{!!} \mathsf{T}$ is evaluated in the ambient colour c to $[\mathsf{V}]_{\mathsf{c}'}^{\mathsf{T}'}$, where $\mathsf{c}' = \mathsf{c} \cup \{\mathsf{a}\}$ et $\mathsf{T}' = \mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$, with a being a fresh nonce. Thus the reduction rule ($\mathcal{C}$/ered.seal)[11] is as follows:

$$\mathsf{B} \vdash \mathsf{V} \mathbin{!!} \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}, \mathsf{a} = \mathsf{V} :_{\mathsf{c}} \mathsf{T} \vdash [\mathsf{V}]_{\mathsf{c} \cup \{\mathsf{a}\}}^{\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})}$$

The ambient colour c may be necessary just to ensure that V has the type T; for V to have the type $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$ requires the transparency of a in addition[12]. A colour can thus be a (finite) set of nonces, which we shall write as $\mathsf{c} = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$. The semantics of a colour is to render its elements transparent. Until now, we had only seen singleton colours $\{\mathsf{a}\}$, abbreviated as a; we call such colours **primary colours**. By synecdoche we will also call an element of a colour a primary colour. The **empty colour**, for which we will prefer the notation $\bullet$, makes no nonce transparent.

[11] Specialised to an uncoloured sealing — see section IV.5.3.6.

[12] This situation is possible because a sealing can be embedded inside another, possibly with a functor interposed which prevents from pushing the brackets induced by the outermost sealing before reaching the evaluation of the functor body containing the innermost sealing. This scenario did not arise in HAT where module definitions were always sequential.

Transparency A nonce is said to be **transparent** or **opaque** (in a colour c, often obvious from context) depending on whether it is, or is not, an element of the colour. (This definition will be generalised under a more semantic form for a larger class of colours in section IV.5.2.3.)

Colour weakening Reducing a sealing moves the sealed expression from a colour to a larger colour. Intuitively, this should not cause any typing trouble: the larger colour provides more typing equations, so more typings are possible. We shall indeed state a **colour weakening** lemma: if E has the type T in the colour c, and c' is a well-formed colour containing c, then E has the type T in c'.

Border colour In a coloured bracket $[\mathsf{E}]^{\mathsf{T}}_{\mathsf{c}'}$, the type annotation T lives on the border between the inner colour c' and the outer (ambient) colour c. We can formalise this by requiring that T be valid in both c and c'. By the colour weakening lemma, it suffices that T be valid in $\mathsf{c} \cap \mathsf{c}'$, and we will use this requirement in typing rules.[13]

Colours and security Let us briefly mention the security interpretation of coloured brackets. Colours can be seen as capabilities handed to expressions — in our application these capabilities unlock type equations. The brackets mark and maintain the boundaries of privileged chunks of code. Nonces are the usual nonces of abstract cryptography. This interpretation was first formulated early in the history of coloured brackets [PS00] and has been studied, in particular, under the name $\lambda_{\mathtt{seal}}$ [SP04].

IV.5.2.2 Semantics of a type and dependencies of a nonce

Semantics of a type in a colour A colour denotes equalities between types, so that there may be more than one way to express a type in a given colour. For instance, if the ambient lexis contains $a_1 = (\langle \textsc{int} \rangle, E_1) :_\bullet \Sigma t : \textsc{type}.\, T_1$, then $(\!|\pi_1\, a_1|\!)$ is equivalent to $\textsc{int}$ in the colour $\{a_1\}$ but not in the empty colour $\bullet$. This possibility for a term to be a valid type in different colours with different semantics (in terms of what expressions have that type) is the key to the expressivity of coloured brackets, as the type annotation is considered in both the inner and the outer colour. Note that the set of terms having a given type is a monotone function of the ambient colour.

Semantics vs. validity Since a type may contain embedded expressions, the very validity of a type (and not just its semantics) may depend on the colour. In the lexis above, the type $\mathsf{S}((\lambda x : (\!|\pi_1\, a_1|\!).\ x)\, 3)$ is only valid in a colour that makes a_1 transparent. Our type system can apparently accomodate this phenomenon, simply by virtue of annotating every typing judgement with a colour, including type correctness $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{T}\, \mathsf{ok}$.

Lexis binding Consider in particular the reduction of a sealing $\mathsf{B} \vdash \mathsf{V}\ !!\ \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}' \vdash [\mathsf{V}]^{\mathbf{self}^{\mathsf{T}}(a)}_{\mathsf{c} \cup \{a\}}$. Since V and T are only known to be valid in the colour c, c must be recorded together with V and T in B', hence $\mathsf{B}' = \mathsf{B}, a = \mathsf{V} :_{\mathsf{c}} \mathsf{T}$. The elements of c are called the **dependencies** of the nonce a.

[13]Using $\mathsf{c} \cap \mathsf{c}'$ rather than replicating a premise in the colour c and c' has technical advantages, mainly in that just because the judgements $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{J}$ and $\mathsf{B}; \Gamma \vdash_{\mathsf{c}'} \mathsf{J}$ are derivable does not automatically mean their derivations have the same shape. We suspect that a common shape can be found, and would work equally well to derive $\mathsf{B}; \Gamma \vdash_{\mathsf{c} \cap \mathsf{c}'} \mathsf{J}$, but proving such a result looks very difficult in our syntactic approach. Furthermore, given the presence of variables in colours (see section IV.5.2.3), we would have to use a semantic intersection that operates on the transparent closures of the colours (see section B.1.2).

Colour well-formation If the nonce a is used (as part of a type $(\!|\mathsf{A}|\!)$) in some colour, the equivalences between projections of a and of its implementation V must hold. Hence the colour in question must contain c (the only colour where V is known to be valid). Therefore a nonce can only be transparent if its dependencies are also transparent. We express this constraint in a side condition in the rule for colour well-formation (($\mathcal{C}$/envok.c.a)). Another option would be to automatically make all dependencies transparent, so that $\{\mathsf{a}\}$ and $\{\mathsf{a}\} \cup \mathsf{c}$ would have the same semantic. We choose the restriction on well-formation as simpler and because it is similar to the condition for using a nonce in an expression.

Use of a nonce in an expression If the nonce a is used in an expression, its type T must be valid. Therefore the ambient colour must contain c (the only colour where T is known to be valid). Therefore a nonce can only be used if its dependencies are transparent, which is expressed in a side condition in the rule ($\mathcal{C}$/ac.a).

Concretisation We might attempt another approach to uses of nonces in an expression. Instead of giving the nonce a the type T taken directly from the sealing expression that created a, we might create a type T' that is equivalent to T in colours containing c but is valid in any colour. In the colour c, we can replace any dependency of a by its implementation. The type resulting from performing all such possible replacements meets the stated requirement. This operation is called **concretisation** of the type T for the colour c. Concretisation is also mutually recursively defined on types, expressions and module components. Concretisation is a copy function, except for the following cases:

$$\begin{aligned}
\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}((\!|\mathsf{A}_1|\!)) &= \mathsf{Typ}\,\mathbf{reveal}^{\mathsf{B}}(\mathsf{A}_1) && \text{if } \mathbf{underl}(\mathsf{A}_1) \in \mathsf{c} \\
\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}((\!|\mathsf{A}_1|\!)) &= (\!|\mathsf{A}_1|\!) && \text{if } \mathbf{underl}(\mathsf{A}_1) \notin \mathsf{c} \\
\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}([\mathsf{E}]^{\mathsf{T}}_{\mathsf{c}'}) &= [\mathsf{E}]^{\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}\cap\mathsf{c}'}(\mathsf{T})}_{\mathsf{c}'}
\end{aligned}$$

(other cases by simple induction)

We get $\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{T} \equiv \mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T})$ as soon as c' contains c.

The advantage of concretisation is to allow the nonce a (with dependencies c) to be used in any colour c', whether or not $\mathsf{c} \subseteq \mathsf{c}'$ holds. This is achieved by giving a the type $\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T})$. Unfortunately, when c is not included in the ambient colour, $\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T})$ is generally not equivalent to T even if the latter is well-formed, which is somewhat confusing. Forcing concretisation instead of restricting nonce use to a suitable colour does not on balance simplify the system design, which is why we eschew it here.

Seal-time concretisation Could we concretise the type of a nonce when the nonce is created, rather than when the nonce is used? The reduction rule for sealing would look like the following:

$$\mathsf{B} \vdash \mathsf{V} \,!!\, \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}, \mathsf{a} = [\mathsf{V}]^{\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T})}_{\mathsf{c}} : \mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T}) \vdash [\mathsf{V}]^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}_{\mathsf{c}\cup\{\mathsf{a}\}}$$

The type of the sealed expression is then universal, i.e., valid in any colour. A lexis binding is also universal, and so need not be annotated with a colour. The slight loss of expressivity triggered by the forced concretisation is thus compensated by a simplification of the type system.

Unfortunately we are here using coloured brackets outside their operating parameters. Usually the type T will contain some abstract components (otherwise the sealing is useless). However the point of selfification was to replace these abstract components in the type annotation on the bracket by a manifest type (using the abstract module identity a). In fact, the type annotation on coloured brackets must be monomorphic, i.e., completely specified, free of TYPE fields (section IV.5.3 will discuss the concept further).

IV.5.2.3 Variables in colours

We saw that in HAT substitution of a value of type T for a variable assumed to have the type T does not always result in a well-typed term. An addditional hypothesis is required stating that the value have the type T in any colour at which the variable is used. One way to ensure that this additional requirement is met is to associate a colour to each variable, which will be the colour of the syntax node at which the variable is bound, and then only allow using the variable in this colour. Then type preservation by substitution only require that the substituted-in value have the right type in the colour of the variable. Reduction rules affecting colours push brackets inside data constructors; they do not affect the colours of variables inside reduced terms (the case of pushing a bracket inside a lambda-abstraction will be discussed in section IV.5.3.3). Compared with HAT, we gain the advantage that beta-reduction ($\mathcal{C}$/ered.app) does not mention any bracket. Annotating variable binding sites with colours is consistent with the principle stated above that anything that is attributed a type is also attributed a colour. However we shall see that confining variables to a colour is not, in itself, sound, and requires additional machinery to make the system sound.

As usual, the colour of a variable binding is given by the innermost surrounding bracket, or in the absence of one by the surrounding colour. A variable binding in an environment records that colour, so that an environment binding in system $\mathcal{C}$ has the form $(x :_c \mathsf{T})$.

The simplest way to state the variable typing rule would be $x :_c \mathsf{T} \vdash_c x :^{\mathsf{P}} \mathsf{T}$, i.e., the variable x is (only) usable in its colour of definition. In order for colour weakening to hold, this condition needs to be relaxed to allow using x in any colour containing the colour of definition, i.e.,

$$x :_c \mathsf{T} \vdash_{c'} x :^{\mathsf{P}} \mathsf{T} \quad \text{when } c \subseteq c'$$

Unfortunately this formulation does not suffice to ensure that colour weakening holds, as shown by the following example:

$$\mathsf{B}; \mathsf{nil} \vdash_c (\lambda x : \mathrm{INT}.\ [[x]_c^{\mathrm{INT}}]_{c_1}^{\mathrm{INT}}) :^{\mathsf{P}} \mathrm{INT} \to \mathrm{INT}$$

Provided that B, c and c_1 are well-formed, this typing judgement is correct. There is no requirement for the intermediate colour c_1 to have any connection with c. If c is widened to some colour c' (such that $c \subseteq c'$), the judgement above becomes

$$\mathsf{B}; \mathsf{nil} \vdash_{c'} (\lambda x : \mathrm{INT}.\ [[x]_c^{\mathrm{INT}}]_{c_1}^{\mathrm{INT}}) :^{\mathsf{P}} \mathrm{INT} \to \mathrm{INT}$$

The colour of the innermost bracket remains c, as there is no indication that it should change during weakening. As a result, the term is no longer well-typed — the variable x is now used in a colour that is smaller than its colour of definition.

One way to perceive this problem is to consider the colour of a variable occurrence as explicitly bound to the colour of the binding of the variable, as opposed to these colours merely having to always being related. One might say that occurrence colours must be computed by name rather than by value. Each variable is then assigned a "symbolic primary colour" (contrast with nonces as constant primary colours). This new primary colour is written identically with the variable. A colour therefore has the form $c = \{a_1, \ldots, a_k, x_1, \ldots, x_n\}$, i.e., a finite set of nonces and colours[14]. A variable can only be used if it is present in the ambient colour, which gives us the following rule:

$$x :_c \mathsf{T} \vdash_{c'} x :^{\mathsf{P}} \mathsf{T} \quad \text{when } x \in c'$$

A further point to note is that the colour c in which x is defined may itself contain other variables — and it may of course contain nonces. These variables and nonces will automatically be considered

[14] Another, symbolic, notation might be $c = \{a_1, \ldots, a_k\} \cup \mathsf{col}(x_1) \cup \cdots \cup \mathsf{col}(x_n)$ where the symbol col notes the colour of definition of its argument and the symbol $\cup$ is interpreted as set union.

transparent whenever x is. In other words, if $x \in c'$, then any element of $c \cup \{x\}$ is transparent in c'. We note this by the typing judgement $x :_c T \vdash_{c'} c \cup \{x\}\,\mathsf{transparent}$. The rule for using a variable in an expression is finally

$$\Gamma_0, x :_c T, \Gamma_1 \vdash_{c'} x :^{P} T \quad \text{when } \Gamma_0, x :_c T, \Gamma_1 \vdash_{c'} x\,\mathsf{transparent}$$

Since variables now occur in colours, colours are subject to alpha-conversion and substitution. Additionally, a substitution must now include a target colour along with a target expression for the variable: the substitution of x by E in the colour c will be written $\{x \leftarrow_c E\}$. The following example illustrates the interaction of substitutions and coloured brackets, assuming that $x \in c$:

$$\{x \leftarrow_{c_0} E_0\}[x]_c^{S(x)} = [E_0]_{(c\setminus\{x\})\cup c_0}^{S(E_0)}$$

IV.5.2.4 Absolute brackets, additive brackets

A coloured bracket $[E]_{c'}^{T}$ lets the expression E (of colour c') be used in any ambient colour c. This is a special case of a relation between the inner colour c' and the outer colour c. Let $\mathscr{R}$ be any binary relation on colours. We can write $[E]_{\mathscr{R}}^{T}$ for a coloured bracket with relation $\mathscr{R}$, which is well-typed in the ambient colour c if and only if there exists a colour c' such that E has the type T in c' and $(c, c') \in \mathscr{R}$ (we assume throughout this discussion that T is valid in the outer colour).

When $\mathscr{R}$ can be an arbitrary relation, an important piece of information is lost, as the inner colour becomes ambiguous. We will therefore limit our analysis to the case where the inner colour is a (partial) function of the outer colour: $[E]_f^T$ is well-typed in the ambient colour c if and only if E has the type T in $f(c)$. Furthermore an important property that ensures that colour weakening will hold is that any widening of the inner colour matches a widening of the outer colour, in other words f must be monotone increasing ($c_1 \subseteq c_2$ implies $f(c_1) \subseteq f(c_2)$).

The coloured brackets that we have seen so far correspond to the case where the function f is total and constant. Such brackets are known as **absolute brackets**. At this point the only way to introduce a bracket in an expression is the reduction of a sealing, in which the bracket has some arbitrary outer colour c_0 and an internal colour of the form $c_0 \cup \{a\}$. Rather than an absolute bracket, we could use an **additive bracket** whose relation is $c \mapsto c \cup \{a\}$ (a total, one-to-one function from the inner colour to the outer colour, thus a (partial) one-to-one function from the outer colour to the inner colour[15]). We write such an additive bracket as $[E]_{+a}^{T}$.

Additive brackets blend in nicely with the rest of the language. In particular occurrences of variables under additive brackets trigger none of the problems discussed in section IV.5.2.3: in an expression such as $\lambda x{:}T.\ [[x]_{+a_2}^{T}]_{+a_1}^{T}$, the colour of the bound occurrence is automatically a superset of the binding colour. Hence we could dispense with the additional complexity resulting from having variables in colour, provided additive brackets were sufficient for our purposes.

Unfortunately, additive brackets are not expressive enough. They can never restrict the set of typing equations accessible in a subexpression, and in particular do not provide a way to enforce that an expression is independent from any surrounding colour. Although this does not impact the intrinsic validity of $\mathcal{C}$, it does limit possible applications. In the security interpretation of brackets, an additive bracket provides additional privileges to the surrounding code, which prohibits any modelling of a , and means that any ordinary code called by privileged code would inherit the privileges. In the context of our objective to cope with distributed system, we will need universal brackets, which ensure that their contents is usable in any context: these can be expressed naturally as absolute brackets annotated with the empty colour $\bullet$ (see section IV.6.3.1). This second application motivates our choice of only including absolute brackets in the language (other forms of brackets being then superfluous).

[15]Since a is fresh, c_0 cannot contain a.

IV.5.3 Polymorphism

IV.5.3.1 Coloration of a type

We saw that how a coloured bracket can be used to build a value of an abstract type, thanks to a suitable type annotation on the bracket. For instance, if a is a nonce whose implementation is $(\langle\textsc{int}\rangle, 3)$ et and signature $\Sigma \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t}$, the expression $[3]^{(\!|\pi_1\,\mathsf{a}|\!)}_{\{\mathsf{a}\}}$ is a value of the abstract type $(\!|\pi_1\,\mathsf{a}|\!)$, whereas $[3]^{\textsc{int}}_{\{\mathsf{a}\}}$ evaluates to 3. Now consider a bracket around a type field: $[\langle\textsc{int}\rangle]^{\mathsf{S}(\langle(\!|\pi_1\,\mathsf{a}|\!)\rangle)}_{\{\mathsf{a}\}}$ is the abstract type field written more simply as $\langle(\!|\pi_1\,\mathsf{a}|\!)\rangle$, while $[\langle\textsc{int}\rangle]^{\mathsf{S}(\langle\textsc{int}\rangle)}_{\{\mathsf{a}\}}$ is equivalent to the simple $\langle\textsc{int}\rangle$. Another expression that might be written is $[\langle\textsc{int}\rangle]^{\textsc{type}}_{\{\mathsf{a}\}}$; but what does it mean?

In an expression of the form $[\langle\mathsf{T}\rangle]^{\textsc{type}}_{\mathsf{c}'}$, the annotation carried by the bracket is not sufficient to decide between the abstract and the concrete version of the type T. The annotation TYPE does not provide any abstraction — any abstraction would result from the use of an abstract type (i.e., of a nonce) in T. Rather TYPE is here an incompletely specified type, a mere indication of the signature of the module expression rather than a full specification of its type field.

The expression $[\langle\mathsf{T}\rangle]^{\textsc{type}}_{\mathsf{c}'}$ denotes a type field, the type in question being described as the type T as seen in the colour c'. As we saw in section IV.5.2.2, the colour has a double influence on the type. On the one hand, it contributes in determining whether the type is well-formed (i.e., whether the judgement $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}\,\mathsf{ok}$ holds). On the other hand, it contributes in determining the semantics of the type, that is, which expressions have this type. For instance, if a is the above nonce, the type $(\!|\pi_1\,\mathsf{a}|\!)$ is well-formed in any colour; in the colour a, the values 3 and $[3]^{\mathsf{Typ}\,\pi_1\,\mathsf{a}}_{\{\mathsf{a}\}}$ both have this type, while only the latter does in the empty colour. In general, a larger colour makes more types valid[16] makes more expressions have a given type.

We have seen a way to transform a type into an equivalent universal type, i.e., a type that is valid in any colour and, in the original colour, characterises the same expressions: this is the concretisation operation $\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}'}(\mathsf{T})$. However $\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}'}(\mathsf{T})$ does not have the same semantics in other colours, for all it is valid; hence $[\langle\mathsf{T}\rangle]^{\textsc{type}}_{\mathsf{c}'}$ cannot be replaced by $\langle\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}'}(\mathsf{T})\rangle$.

The semantics of the expression $[\langle\mathsf{T}\rangle]^{\textsc{type}}_{\mathsf{c}'}$ is novel: it cannot be expressed by previously seen means. It is not clear whether such an expression should be accepted at all. We shall evaluate the pros and cons of allowing such expressions. But first, we need to characterise them precisely, which we do by assigning kinds to types.

IV.5.3.2 Kinds

We would like to recognise types that fully specify values. Obvious such types are singletons: all pure expressions having a given singleton types are essentially equivalent. Following a very strict interpretation, one might say that the only fully specified types are singletons. However, under extensional equivalence, a type can be a singleton semantically without being one syntactically; for example the type $\mathsf{S}(3) * \mathsf{S}(4)$ does not contain any more values than $\mathsf{S}((3,4))$. Some types may even contain a single value up to observational equivalence for reasons having to do with the language as a whole, such as parametricity [Wad89] results in ML that ensure that the only function of type $\forall\alpha, \alpha \to \alpha$ (which we would write $\Pi \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t} \to \mathsf{Typ}\,\mathsf{t}$) is the identity function.

In fact, we are trying to characterise the signatures that fully specify the type fields that they contain. For example, although the type BOOL contains two observationally distinguishable values, we are content with BOOL as a specification of a boolean value: we do not treat BOOL as an abstract

[16]The colour has a bearing on embedded expressions. For instance $\mathsf{S}((\lambda x : \mathsf{Typ}\,\pi_1\,\mathsf{a}.\ x)\,3)$ is only well-formed in a colour that reveals a.

type. We aim to describe a programming language, not a proof language; as a consequence we limit abstraction to types, and allow the revelation of $[\mathsf{true}]^{\text{BOOL}}_{\mathsf{c}'}$ as true.

The paradigm of an incomplete signature is TYPE, which designates an unspecified type field. More generally, any type containing TYPE in a covariant position, such as $\text{INT} * \text{TYPE}$ or $\text{INT} \rightarrow \text{TYPE}$, is incompletely specified. The presence of TYPE in a contravariant position does not indicate incompleteness, as shown for example in the constant function type $\Pi \mathsf{t} : \text{TYPE}.\ \mathsf{S}(\mathsf{V})$.

In order to formalise this notion, we equip system $\mathcal{C}$ with two **kinds**. The kind $\wr$ contains fully specified types such as $\mathsf{S}(\mathsf{E})$, INT or $\Pi \mathsf{t} : \text{TYPE}.\ \mathsf{T}$ with T of kind $\wr$. A type of kind $\wr$ is said to be **fully specified** or **completely specified**. The kind $*$ contains all types irregardless of their level of specification; a type that does not have the kind $\wr$ is said to be **partially specified** or **incompletely specified**. Kinds, written K, are equipped with an order relation $\wr \leqslant *$; the least upper bound of two kinds is written $\mathsf{K}_1 \vee \mathsf{K}_2$ ($\wr \vee * = *$) and the greatest lower bound is written $\mathsf{K}_1 \wedge \mathsf{K}_2$ ($\wr \wedge * = \wr$). Type kinding comes with a very simple subkinding relation ship: if T has the kind K_1 and $\mathsf{K}_1 \leqslant \mathsf{K}_2$ then T has the kind K_2.

It is tempting to call a fully specified type **monomorphic** (as opposed to **polymorphic** for a partially specified type). Another tempting designation is concrete (vs. abstract). These terminologies is slightly misleading out of context (as attested by their multiplicity). In a way the type TYPE is a type variable (under which interpretation $\wr$ is the kind of closed types), and the meaning of its presence depends on how type variables are quantified. If they are quantified universally, types of kind $*$ are polymorphic; if they are quantified existentially, types of kind $*$ are abstract. For aesthetic reasons, we will usually use the words "monomorphic" and "polymorphic" (the latter usually meaning non-monomorphic rather than just having the kind $*$). We do however warn the reader to take this terminology with a grain of salt.

Type kinding rules are fairly simple: TYPE is polymorphic, any other constructor is monomorphic if and only if its components are. In particular, all base types (BOOL, INT, etc.) other than TYPE are monomorphic, as are singletons. A product type is monomorphic if its components are. A function type is monomorphic if its result type is.

The remaining case is that of the projection of a type field from an expression: when is $\mathsf{Typ}\,\mathsf{E}$ monomorphic? Since the types $\mathsf{Typ}\,\langle\mathsf{T}\rangle$ and T are equivalent, $\mathsf{Typ}\,\langle\mathsf{T}\rangle$ must be monomorphic if and only if T is. Given the presence in our language of the dependent type $\mathsf{Typ}\,\mathsf{E}$, we need to reflect type kinding at the expression typing level. We annotate the type TYPE with a kind annotation, writing $\text{TYPE}^{\wr}$ or TYPE^{*}: the type TYPE^{K} characterises type fields whose contents is a type of kind K. For example $\langle\text{INT}\rangle$ and $\langle\text{TYPE}^{*} \rightarrow \text{INT}\rangle$ have the type $\text{TYPE}^{\wr}$, and so does $\langle\text{INT} * \mathsf{Typ}\,\mathsf{x}\rangle$ when x has the type $\text{TYPE}^{\wr}$; whereas $\langle\text{TYPE}^{\mathsf{K}}\rangle$ and $\langle\text{INT} \rightarrow \text{TYPE}^{\wr}\rangle$ only have the type TYPE^{*}. One must be careful not to confuse the type assigned to the expression with the type contained in the field: for example $\langle\text{TYPE}^{*}\rangle$ is an expression containing a type field, which happens to be the type of arbitrary type fields (in ML, this would be a signature field in a module, e.g., `struct module type S : sig end end` in Objective Caml). The expression $\langle\text{TYPE}^{*}\rangle$ has the momonorphic type $\mathsf{S}(\langle\text{TYPE}^{*}\rangle)$, but not the type of monomorphic fields $\text{TYPE}^{\wr}$ — nor does $\langle\text{TYPE}^{\wr}\rangle$ since $\text{TYPE}^{\wr}$ does not have the kind $\wr$.

IV.5.3.3 Brackets and function application; polymorphic functions

Reducing a sealing introduces a coloured bracket during evaluation. The type on this bracket is produced by the selfification operation, which creates a monomorphic type. This is the central point of selfification: the type on a sealing is usually incompletely specified, and selfification completes the selfification, by replacing the unspecified parts by projections of the new name. When a coloured bracket is pushed inside a data structure, the monomorphic nature of the type annotations is preserved — all the terms being manipulated have monomorphic brackets (as opposed to polymorphic brackets whose type annotation is partially specified). One aspect of bracket pushing in system $\mathcal{C}$ remains to be described however, namely pushing brackets inside a function.

Consider the expression $[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\,{}^{P}T_1}$ in some ambient colour c. In order for it to be well-typed, T_2 must be a subtype of T_0 and E must have the type T_1 in the colour c'. When this expression is applied to an argument V of type T_0, the result must be that of beta-reduction $\{x \leftarrow V\}E$, with any necessary coloured brackets thrown in. Let us study how to manage brackets during evaluation.

We announced in section IV.5.2.3 that beta-reduction would remain unadorned, which forces us to rule on the the fate of brackets as soon as they are pushed under the lambda. This does not constrain our latitude regarding the choice of semantics: we are effectively giving a symbolic name x to the effective argument (as well as the outer colour c, which is the colour of the argument). The question is therefore how to reduce $[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\,{}^{P}T_1}$.

The most obvious target uses coloured brackets both around the body of the function (to mark the border on exit from the function) and around each occurrence of the parameter (to mark the border when entering the function). The argument must also be protected in the return type.

$$[\lambda z : T_2.\ E]_{c'}^{\Pi y:T_0.\,{}^{P}T_1} \longrightarrow_c \lambda x : T_0.\ [\{z \leftarrow_{\{x\}} [x]_{c\cup\{x\}}^{T_0}\}E]_{c'}^{\{y \leftarrow_{\{x\}} [x]_{c\cup\{x\}}^{T_0}\}T_1} \qquad \text{(ered.col.fun.P-POLY)}$$

The bracket around the parameter x in the function body and in the return type must allow x to be used inside, hence the colour annotating the bracket must contain x (adding c as well is technically useless since x automatically brings in the colour of the binding c). The choice of what type annotation to put on this bracket is not so obvious. We know that E and T_1 are well-typed as soon as their variable has the type T_0 (note that T_2 is a subtype of T_0); T_0 is valid in c given the annotation on the bracket in the redex, therefore T_0 is a possible choice.

Nothing however requires T_0 to be monomorphic, even if $\Pi y : T_0.\,{}^{P}T_1$ is. Therefore this rule introduces brackets carrying polymorphic type annotations. In fact, for T_0 to be partially specified means that the function $\lambda z : T_0.\ E$ is a **polymorphic function**[17]. This terminology follows that of ML: in ML, a polymorphic function has a type scheme $\forall \alpha, T_0 \to T_1$, which corresponds in our dependently typed system to $\Pi t : \text{TYPE}^{\wr}.\ T_0 \to T_1$ (with $\mathsf{Typ}\ t$ corresponding to α).

If we want to adopt bracket pushing into functions as stated above, we must accept polymorphic brackets. Let us now study these further, in the light of how they can appear. We will then look for a way of avoiding them.

IV.5.3.4 Polymorphic types and values

We still assume that bracket pushing into functions happens according to the rule (ered.col.fun.P-POLY) from section IV.5.3.3. Polymorphic brackets result from applying a polymorphic function inside a colour other than its colour of definition: a bracket $[E]_{c'}^{\text{TYPE}^{\wr}}$ appears when applying a function $[f]_{c'}^{\Pi t:\text{TYPE}^{\wr}.\ T_1}$. Let us first look at a very simple example: the identity function on the type $\text{TYPE}^{\wr}$, published from c' under the type $\Pi t : \text{TYPE}^{\wr}.\ S(t)$.

$$\left([\lambda t : \text{TYPE}^{\wr}.\ t]_{c'}^{\Pi t:\text{TYPE}^{\wr}.\ S(t)}\right) \langle \text{INT}\rangle \longrightarrow_c \left(\lambda t : \text{TYPE}^{\wr}.\ [[t]_{c\cup\{t\}}^{\text{TYPE}^{\wr}}]_{c'}^{S([t]_{c\cup\{t\}}^{\text{TYPE}^{\wr}})}\right) \langle \text{INT}\rangle$$
$$\longrightarrow_c [[\langle \text{INT}\rangle]_{c}^{\text{TYPE}^{\wr}}]_{c'}^{S([\langle \text{INT}\rangle]_{c}^{\text{TYPE}^{\wr}})}$$

[17]This is a case of the strong connection between partial specification and polymorphism when the variable is universally quantified that we mentioned in section IV.5.3.2.

In this case, the final value is $[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}$ — the identity function returns its argument protected by a spurious bracket annotated by the ambient colour.

Let us now examine the polymorphic identity function $\lambda \mathsf{t} : \textsc{type}^{\wr}.\ \lambda x : \mathsf{Typ}\,\mathsf{t}.\ x$ published from c' under the type $\Pi \mathsf{t} : \textsc{type}^{\wr}.\ \mathsf{Typ}\,\mathsf{t} \to \mathsf{Typ}\,\mathsf{t}$.

$$
\begin{aligned}
&\left([\lambda \mathsf{t} : \textsc{type}^{\wr}.\ \lambda x : \mathsf{Typ}\,\mathsf{t}.\ x]_{\mathsf{c}'}^{\Pi \mathsf{t}:\textsc{type}^{\wr}.\ \mathsf{Typ}\,\mathsf{t} \to \mathsf{Typ}\,\mathsf{t}}\right)\ \langle\textsc{int}\rangle\ 3 \\
&\qquad\longrightarrow_{\mathsf{c}} \left(\lambda \mathsf{t} : \textsc{type}^{\wr}.\ [\lambda x : \mathsf{Typ}\,[\mathsf{t}]_{\mathsf{c}\cup\{\mathsf{t}\}}^{\textsc{type}^{\wr}}.\ x]_{\mathsf{c}'}^{\mathsf{Typ}\,[\mathsf{t}]_{\mathsf{c}\cup\{\mathsf{t}\}}^{\textsc{type}^{\wr}} \to \mathsf{Typ}\,[\mathsf{t}]_{\mathsf{c}\cup\{\mathsf{t}\}}^{\textsc{type}^{\wr}}}\right)\ \langle\textsc{int}\rangle\ 3 \\
&\qquad\longrightarrow_{\mathsf{c}} [\lambda x : \mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}.\ x]_{\mathsf{c}'}^{\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}} \to \mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}\ 3 \\
&\qquad\longrightarrow_{\mathsf{c}} \left(\lambda x : \mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}.\ [[x]_{\mathsf{c}\cup\{x\}}^{\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}]_{\mathsf{c}'}^{\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}\right)\ 3 \\
&\qquad\longrightarrow_{\mathsf{c}} [[3]_{\mathsf{c}}^{\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}]_{\mathsf{c}'}^{\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}
\end{aligned}
$$

The argument 3 is now surrounder by two brackets. The inner bracket places the value into the colour c' with a polymorphic value; the outer bracket, in spite of its similar appearance, has a differnet rôle as the seemingly polymorphic type $\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}$ is in fact monomorphic in the outside colour c.

In order to generalise upon these examples, a few concepts are worth noting. A bracket $[\langle \mathsf{T}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}$ is a **polymorphic type parameter** for a function. A bracket $[\mathsf{V}]_{\mathsf{c}}^{\mathsf{Typ}\,[\langle \mathsf{T}\rangle]_{\mathsf{c}}^{\textsc{type}^{\wr}}}$ is a **polymorphic value**. A polymorphic value has no apparent structure, since it is protected by a bracket (which cannot be reduced away since its type annotation itself has no apparent structure). The body of the function is unable to manipulate polymorphic values in a way other than polymorphic. This approach should work to model parametrically polymorphic languages such as ML. However it is problematic in a non-parametric language with generics or dynamic type-checking (which we will introduce in system $\mathcal{D}$).

We shall not go any further along the lines of studying polymorphic values. It remains to be seen how to reduce polymorphic brackets. In particular, how can the polymorphic identity function return its argument with no superfluous bracket? (Note that the argument passes through the colour c'; how can we make sure that this passage is harmless?)

IV.5.3.5 Colour fusion

We present a solution to the problem of managing brackets around polymorphic function calls. This solution lacks expressivity and finesse, but remains attractive in a certain light — not least because of its simplicity. The idea is to merge the colour of the argument with the colour of the function body.

$$[\lambda x : \mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi x:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1} \longrightarrow_{\mathsf{c}} \lambda x : \mathsf{T}_0.\ [\mathsf{E}]_{\mathsf{c}'\cup\{x\}}^{\mathsf{T}_1} \qquad (\mathcal{C}/\mathsf{ered.col.fun.P})$$

We abandon any thought of protecting the argument: all type equations required to type the argument are allowed when executing the function. This rule is very simple, technically speaking: one bracket turns into one bracket, with a smaller type annotation and smaller contents.

This rule enjoys a certain symmetry: applying $[\lambda x : \mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi x:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1}$ to an argument V yields

$$[\{x \leftarrow_{\mathsf{c}} \mathsf{V}\}\mathsf{E}]_{\mathsf{c}\cup\mathsf{c}'}^{\{x \leftarrow_{\mathsf{c}} \mathsf{V}\}\mathsf{T}_1}$$

so that the computations are simply performed in the union of the colours of the expressions that come into contact (V and E)

Let us check the result of an application of the polymorphic identity function using this rule.

$$\begin{aligned}
&\left([\lambda t:\text{TYPE}^{\wr}.\ \lambda x:\mathsf{Typ}\,t.\ x]_{c'}^{\Pi t:\text{TYPE}^{\wr}.\ \mathsf{Typ}\ t\to\mathsf{Typ}\ t}\right)\ \langle\text{INT}\rangle\ 3\\
&\qquad\longrightarrow_c\ \left(\lambda t:\text{TYPE}^{\wr}.\ [\lambda x:\mathsf{Typ}\,t.\ x]_{c'\cup\{t\}}^{\mathsf{Typ}\ t\to\mathsf{Typ}\ t}\right)\ \langle\text{INT}\rangle\ 3\\
&\qquad\longrightarrow_c\ [\lambda x:\mathsf{Typ}\,\langle\text{INT}\rangle.\ x]_{c\cup c'}^{\mathsf{Typ}\,\langle\text{INT}\rangle\to\mathsf{Typ}\,\langle\text{INT}\rangle}\ 3\\
&\qquad\longrightarrow_c\ \left(\lambda x:\mathsf{Typ}\,\langle\text{INT}\rangle.\ [x]_{c\cup c'\cup\{x\}}^{\mathsf{Typ}\,\langle\text{INT}\rangle}\right)\ 3\\
&\qquad\longrightarrow_c\ [3]_{c\cup c'}^{\mathsf{Typ}\,\langle\text{INT}\rangle}\ \longrightarrow_c\ [3]_{c\cup c'}^{\text{INT}}\ \longrightarrow_c\ 3
\end{aligned}$$

In our study of system $\mathcal{C}$, we will retain this fusion formulation of bracket pushing around a function.

IV.5.3.6 Generative functors

We saw in section IV.5.1.5 that selfification does not affect generative functors. Since selfification produces a type that is meant to annotate a coloured bracket (as it is used in ($\mathcal{C}$/ered.seal)), the resulting type must be monomorphic. Therefore a generative functor type $\Pi x : T_0.\ ^{\mathtt{I}}T_1$ must be monomorphic even if T_1 is polymorphic[18].

We have stated a rule ($\mathcal{C}$/ered.col.fun.P) to push coloured brackets bearing an applicative functor type. The transposition to a generative functor type is not straightforward. A naive proposal would be

$$\frac{}{[\lambda x:T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathtt{I}}T_1}\longrightarrow_c \lambda x:T_0.\ [E]_{c'\cup\{x\}}^{T_1}}$$

However T_1 may be polymorphic, in which case the right-hand side is ill-typed. Intuitively, the rule above cannot be suitable because the left-hand side is a generative functor, whose every application triggers the creation of a new nonce, whereas this aspect is simply not present in the right-hand side.

Nonces are generated by the reduction rule ($\mathcal{C}$/ered.seal) for sealing expressions. Let us therefore introduce a sealing in the right-hand side. We may attempt to place the sealing inside or outside the coloured brackets:

$$\frac{}{[\lambda x:T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathtt{I}}T_1}\longrightarrow_c \lambda x:T_0.\ ([E]_{c'\cup\{x\}}^{T_1}\ !!\ T_1)}$$

$$\frac{}{[\lambda x:T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathtt{I}}T_1}\longrightarrow_c \lambda x:T_0.\ [E\ !!\ T_1]_{c'\cup\{x\}}^{T_1}}$$

In both cases, although the intuitive behaviour is acceptable, formal correction is lacking, as the type annotation on the coloured bracket may still be polymorphic. However this is a benign form of ill typing, as reduction will change the type into a monomorphic one before the coloured bracket is reduced.

A sealing construct expresses a static border between abstraction domains, whereas a coloured bracket is a dynamic border. We have here a border that is both static and dynamic. We will note it by a **coloured sealing**, written $E\ !!_{c'}\ T$. This sealing acts like normal sealing, except that it gives the expression E the additional knowledge of abstract types designated by c'. A coloured sealing

[18] In a way a generative functor type is amorphous: it is not yet fully specified, but will give rise to a monomorphic type when the functor is applied.

therefore includes the effect of an additive bracket (see section IV.5.2.4). A normal sealing is the special case where the coloured sealing adds no extra knowledge: $\mathsf{E} \mathbin{!!} \mathsf{T} = \mathsf{E} \mathbin{!!}_{\bullet} \mathsf{T}$.

Pushing a coloured bracket bearing a generative functor type shall produce a coloured sealing:

$$[\lambda x : \mathsf{T}_2.\ \mathsf{E}]_{c'}^{\Pi x:\mathsf{T}_0.^{P}\mathsf{T}_1} \longrightarrow_c \lambda x : \mathsf{T}_0.\ (\mathsf{E} \mathbin{!!}_{c'} \mathsf{T}_1) \qquad (\mathcal{C}/\mathsf{ered.col.fun.I})$$

Note that thanks to the additivity of c' we do not need to include x, in contrast with $(\mathcal{C}/\mathsf{ered.col.fun.P})$. A coloured sealing is reduced by the rule $(\mathcal{C}/\mathsf{ered.seal})$ which we can finally state in its full glory:

$$\frac{}{\mathsf{B} \vdash \mathsf{V} \mathbin{!!}_{c'} \mathsf{T} \longrightarrow_c \mathsf{B}, a = \mathsf{V} :_{c \cup c'} \mathsf{T} \vdash [\mathsf{V}]^{\mathbf{self}^{\mathsf{T}}(a)}_{c \cup c' \cup \{a\}}} \ (\mathcal{C}/\mathsf{ered.seal})$$

IV.5.4 Evaluation

IV.5.4.1 Syntax

The syntax of system $\mathcal{C}$ extends system $\mathcal{E}$ with two new constructs that should not appear in source programs: abstract types and coloured brackets. Furthermore the signature TYPE now carries a kind annotation, and sealing now carries a colour annotation (the notation $\mathsf{E} \mathbin{!!} \mathsf{T}$ is kept as an abbreviation for $\mathsf{E} \mathbin{!!}_{\bullet} \mathsf{T}$).

$\mathsf{K} ::=$	**kind**
$\wr$	monomorphic (fully specified)
$*$	polymorphic (partially specified)
$\mathsf{T} ::=$	**type**
...	
TYPE^{K}	abstract type field
$(\!\mid A \mid\!)$	abstract type
$\mathsf{E} ::=$	**expression**
...	
$\mathsf{E} \mathbin{!!}_c \mathsf{T}$	sealed and coloured module
$[\mathsf{E}]_c^{\mathsf{T}}$	coloured bracket
$A ::=$	**module component**
a	nonce
$A\ \mathsf{E}$	application
$\pi_i A$	projection ($i \in \{1, 2\}$)
$\xi ::=$	**primary colour**
a	nonce
x	variable
$c ::=$	**colour**
$\bullet$	empty colour (also written $\{\}$)
$\{a_1, \dots, a_k, x_1, \dots, x_k\}$	finite set of primary colours

Recall that kinds are equipped with an order relation, written $\mathsf{K}_1 \leqslant \mathsf{K}_2$, such that $\wr \leqslant *$. We write $\mathsf{K}_1 \vee \mathsf{K}_2$ for the least upper bound of K_1 and K_2, and $\mathsf{K}_1 \wedge \mathsf{K}_2$ for their greatest lower bound.

If A is a module component, its underlying nonce $\mathbf{underl}(A)$ is formally defined as follows:

$$\begin{aligned}\mathbf{underl}(a) &= a\\ \mathbf{underl}(A\,E) &= \mathbf{underl}(A)\\ \mathbf{underl}(\pi_i A) &= \mathbf{underl}(A)\end{aligned}$$

Revelation of a module component is defined as follows:

$$\begin{aligned}\mathbf{reveal}^B(a) &= E && \text{where } a = E :_c T \in B\\ \mathbf{reveal}^B(A\,E) &= (\mathbf{reveal}^B(A))\,E\\ \mathbf{reveal}^B(\pi_i A) &= \pi_i\,(\mathbf{reveal}^B(A))\end{aligned}$$

Typing judgements now carry a lexis and a colour. Additional right-hand sides to those in system $\mathcal{E}$ are colour transparency, revelation of a module component and conversion and convertibility for components.

$\mathscr{J}$ ::=	**typing judgement**
$B; \Gamma \vdash_c J$	local judgement

J ::=	**local judgement right-hand side**
...	
$T : K$	type kinding (generalising $T\,\mathsf{ok}$)
$c_0\,\mathsf{transparent}$	colour transparency
$A \triangleright E : T$	component revelation
$A \longrightarrow A'$	component conversion
$A \equiv A'$	convertibility equivalence on components

We write $\xi\,\mathsf{transparent}$ for $\{\xi\}\,\mathsf{transparent}$.

Environments now contain colour annotations. We also state the syntax of lexes.

B ::=	**lexis**
nil	empty
$B, a = E :_{c_0} T$	nonce a with implemented by E with the signature T

Γ ::=	**environnement**
nil	empty
$\Gamma, x :_c T$	binding of the variable x

Following the definition for environments, the domain of a lexis B, i.e., the set of nonces that it records, is written $\mathbf{dom}\,B$.

Since colours may contain variables, they are affected by substitutions. A substitution specifies both an expression and a colour to replace the variable with. The substitution of E for x under c in $\aleph$ is written $\{x \leftarrow_c E\}\aleph$.

IV.5.4.2 Values and abstract components

Brackets and values As in HAT, the set of values depends on the ambient colour. We first define a grammatical notion of *quasi-value*, which is a value with some possibly-eliminatable brackets. In addition to the values of system $\mathcal{E}$ (which are the same as in system $\mathcal{B}$), coloured brackets may appear in quasi-values and (with restrictions) in values. In order for an expression of the form $[V]_{c'}^{T}$ to be a value, V must be a value (in the colour c'), and T must have an appropriate form. If T has apparent structure, the bracket pushing rules allow $[V]_{c'}^{T}$ to be reduced. The only case where $[V]_{c'}^{T}$ may be a value is when T is an abstract type $(\!|A|\!)$. Even then, $[V]_{c'}^{(\!|A|\!)}$ may be reducible in some colours.

Quasi-values The language of **quasi-values** in system $\mathcal{C}$ is a supergrammar of the one for $\mathcal{B}$, with brackets carrying an abstract type annotation thrown in. The abstract type annotation must itself be in a reduced form, called **component value**, where functor arguments are all values.

$V ::=$	**quasi-value**
$\ldots$	
$[V]_{c'}^{(\!|A^V|\!)}$	potentially abstraction-making coloured bracket

$A^V ::=$	**component value**
a	nonce
$A^V\, V$	application to a quasi-value
$\pi_i\, A^V$	projection ($i \in \{1, 2\}$)

Irreducioble coloured brackets A quasi-value of the form $[V]_{c'}^{(\!|A^V|\!)}$ is only a value if the bracket cannot be eliminated. Intuitively a coloured bracket is indispensible only if it actually creates abstraction, which translates as the requirement that the underlying nonce of A^V must be opaque in the ambient colour yet transparent in the inside colour c'. We will analyse the behaviour of a coloured bracket expression according to the transparency of the underlying nonce when presenting bracket elimiation rules in section IV.5.4.2.

Values and abstract components The set of values depends on the ambient colour: we write V^c for a value in the colour c. The set of values is described as a family of grammars parametrised by a colour; it is a subset of quasi-values. In order for a quasi-value $[V]_{c'}^{(\!|A^V|\!)}$ to be a value, the bracket must be indispensible in the sense described above, and the quasi-values in A^V must themselves be values in the appropriate colour.

$V^c ::=$	**value in** c
$() \mid bv \mid \underline{n}$	constant
$\langle T \rangle$	type field
(V_1^c, V_2^c)	pair
$\lambda x : T.\ E$	lambda-abstraction
$[V^{c'}]_{c'}^{(\!|A^{V^{c \cap c'}}|\!)}$	coloured bracket, if $A^{V^{c \cap c'}}$ is abstract in c but concrete in c'

$A^{V^c} ::=$	**abstract component in** c
a	nonce, if opaque in c
$A^{V^c}\, V^c$	application of a functor to a value
$\pi_i\, A^{V^c}$	projection ($i \in \{1, 2\}$)

Strictly speaking, since transparency of a nonce is a semantic value depending on a lexis and an environment, the notions of values and abstract components should be indexed by a lexis and an environment. In practice the lexis and environments will always be clear from context, so we omit them.

IV.5.4.3 $\boxed{B \vdash E \longrightarrow_c B' \vdash E'}$ Reduction

Evaluatin contexts We reduce expressions under brackets. Although brackets are initially introduced around values, this property is not preserved by reduction; specifically, pushing a bracket inside a function body results in a bracket surrounding an arbitrary expression. When the type annotation on a bracket is the type field of a module, the module expression must also be reduced.

$\mathsf{C} ::=$	**evaluation context (of depth 1)**
$\dots$	
$\_\_ \mathrel{!!_{\mathsf{c}_1}} \mathsf{T}$	sealing
$[\_\_]^{\mathsf{T}}_{\mathsf{c}_1}$	coloured bracket
$[\mathsf{V}^{\mathsf{c}_1}]^{\mathsf{Typ}\,\_\_}_{\mathsf{c}_1}$	type field on a bracket

Formally speaking, the set of evaluation contexts, like values, depends on a lexis and an environment. In the context $[\_\_]^{\mathsf{T}}_{\mathsf{c}_1}$, the expression inside is reduced in the colour c_1. In the context $[\mathsf{V}^{\mathsf{c}_1}]^{\mathsf{Typ}\,\_\_}_{\mathsf{c}_1}$, the expression inside is reduced in the colour $\mathsf{c} \cap \mathsf{c}_1$ (the intersection of the colours outside and inside the border upon which the expression lies).

Computational rules System $\mathcal{C}$ inherits the rules that were already present in $\mathcal{B}$, viz., ($\mathcal{C}$/ered.app), ($\mathcal{C}$/ered.proj), ($\mathcal{C}$/ered.let), ($\mathcal{C}$/ered.context). These rules can be used in any colour and any lexis; the colour is added to substitution when required. Values are also considered in their ambient colour. Reduction inside contexts happens in the inside colour of the context. Appending A lists all the rules of system $\mathcal{C}$, included the inherited rules.

The rule for reducing a sealing is modified to surround the value with a coloured bracket, and to take the colour annotation on the sealing into account.

$$\mathsf{B} \vdash \mathsf{V}^{\mathsf{c}\cup\mathsf{c}'} \mathrel{!!_{\mathsf{c}'}} \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}, \mathsf{a} = \mathsf{V}^{\mathsf{c}\cup\mathsf{c}'} :_{\mathsf{c}\cup\mathsf{c}'} \mathsf{T} \vdash [\mathsf{V}^{\mathsf{c}\cup\mathsf{c}'}]^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}_{\mathsf{c}\cup\mathsf{c}'\cup\{\mathsf{a}\}} \quad (\mathcal{C}/\mathsf{ered.seal})$$

$$\text{where } \mathsf{a} \text{ is fresh (i.e., } \mathsf{a} \notin \mathbf{dom}\,\mathrm{B})$$

Reductions in types Until system $\mathcal{E}$, types contained in expressions did not influence reduction. This is no longer the case in system $\mathcal{C}$, since the reduction of a coloured bracket depends on the type annotation carried by the bracket, specifically on the head constructor on this type. Nonetheless our computational needs on types are small enough — we only need to reach a weak head normal form, and only in a single context within expressions, so we do not need to introduce a reduction on types. The only destructor in the syntax of types is $\mathsf{Typ}\,\_\_$; its argument can be reduced via the context $[\mathsf{V}^{\mathsf{c}_1}]^{\mathsf{Typ}\,\_\_}_{\mathsf{c}_1}$, after which the destructor can be eliminated with ($\mathcal{C}$/ered.colTyp).

$$[\mathsf{V}^{\mathsf{c}'}]^{\mathsf{Typ}\,\langle\mathsf{T}\rangle}_{\mathsf{c}'} \longrightarrow_{\mathsf{c}} [\mathsf{V}^{\mathsf{c}'}]^{\mathsf{T}}_{\mathsf{c}'} \qquad (\mathcal{C}/\mathsf{ered.colTyp})$$

Abstract types are peculiar, as $(\!|A|\!)$ is in weak head normal form if and only if the underlying nonce is opaque, but must be revealed if it is transparent.

$$\mathsf{B} \vdash [\mathsf{V}^{\mathsf{c}'}]^{(\!|A|\!)}_{\mathsf{c}'} \longrightarrow_{\mathsf{c}} \mathsf{B} \vdash [\mathsf{V}^{\mathsf{c}'}]^{\mathsf{Typ}\,\mathbf{reveal}^{\mathrm{B}}(A)}_{\mathsf{c}'} \qquad (\mathcal{C}/\mathsf{ered.colAbs})$$

$$\text{if } \mathbf{underl}(A) \in \mathsf{c} \cap \mathsf{c}'$$

Bracket pushing When a bracket surrounds a value, and the type annotation on the bracket is not an abstract type, the bracket is pushed inside a value. The bracket pushing rules mostly follow the same principle as in HAT (see sections III.1.2.2 and III.2.5). The selection of the bracket pushing rule relies on the type that is apparent on the bracket (specifically its head constructor); the effect on the expression is to push the bracket inside the constructor for this type.

$$[()]_{c'}^{\mathrm{UNIT}} \longrightarrow_c () \qquad (\mathcal{C}/\mathsf{ered.col.base.unit})$$

$$[bv]_{c'}^{\mathrm{BOOL}} \longrightarrow_c bv \qquad (\mathcal{C}/\mathsf{ered.col.base.bool})$$

$$[\underline{n}]_{c'}^{\mathrm{INT}} \longrightarrow_c \underline{n} \qquad (\mathcal{C}/\mathsf{ered.col.base.int})$$

$$[V^{c'}]_{c'}^{\mathsf{S}(E)} \longrightarrow_c E \qquad (\mathcal{C}/\mathsf{ered.col.sing})$$

$$[(V_1^{c'}, V_2^{c'})]_{c'}^{\Sigma x:T_1.\,T_2} \longrightarrow_c ([V_1^{c'}]_{c'}^{T_1}, [V_2^{c'}]_{c'}^{\{x \leftarrow_c [V_1^{c'}]_{c'}^{T_1}\}T_2}) \qquad (\mathcal{C}/\mathsf{ered.col.pair})$$

In the case of functions, we adopt the colour fusion rule explained in section IV.5.3.5. In the case of a generative functor, new types must be created whenever the functor is applied, so we add a sealing to the body of the functor; the colour annotation on the sealing plays the role of a coloured bracket.

$$[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\,^{\mathsf{P}}T_1} \longrightarrow_c \lambda x : T_0.\ [E]_{c' \cup \{x\}}^{T_1} \qquad (\mathcal{C}/\mathsf{ered.col.fun.P})$$

$$[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\,^{\mathsf{I}}T_1} \longrightarrow_c \lambda x : T_0.\ (E\ !!_{c' \cup \{x\}}\ T_1) \qquad (\mathcal{C}/\mathsf{ered.col.fun.I})$$

When a bracket immediately surrounds another bracket and neither bracket can be reduced by one of the already mentioned pushing rules, i.e., given an expression of the form $[[V^{c_2}]_{c_2}^{(\!|A_2|\!)}]_{c_1}^{(\!|A_1|\!)}$ where $[V^{c_2}]_{c_2}^{(\!|A_2|\!)}$ is a value, there are three possible behaviours.

- If the annotation on the outer bracket makes it simplfiable, the outer bracket disappears. In HAT, this is performed by $(\mathcal{H}/\mathsf{ered.col.le})$. Here the rule $(\mathcal{C}/\mathsf{ered.colAbs})$ is used, followed by computations on the revealed expression in the type annotation and possibly later bracket pushing.
- If the annotation on the outer bracket is abstract outside but concrete inside, the expression is a value.
- The remaining case is when the annotation on the outer bracket is abstract outside as well as inside. In HAT, typing ensures that A_1 and A_2 are equal, and the outer bracket is erased by the rule $(\mathcal{H}/\mathsf{ered.col.col})$.

In our present systems, which includes functors and colours with non-trivial intersections, the situation is more complex. A new possibility arises that $A_1 = a_1\,V_1$ and $A_2 = a_2\,V_2$; then typing ensures that (as in HAT) $a_1 = a_2$, but the arguments are only known to be equivalent in the intermediate colour c_1. The arguments may not be equivalent in c, so c_1 is (sometimes) an obligatory intermediate. We state a weaker rule, which (as in function application) merges the colours in play.

$$[[V^{c_2}]_{c_2}^{(\!|A_2|\!)}]_{c_1}^{(\!|A_1|\!)} \longrightarrow_c [V^{c_2}]_{c_1 \cup c_2}^{(\!|A_1|\!)} \qquad (\mathcal{C}/\mathsf{ered.col.merge})$$

if A_1 et A_2 are both opaque in c_1 but A_2 is concrete in c_2

IV.5.5 Typing

The type system of system $\mathcal{C}$ inherits from that of $\mathcal{E}$, but all rules must be modified to add lexes and colours. For most rules, this modification is done mechanically by permitting an arbitrary lexis and colour. Each judgement $\Gamma \vdash J$ becomes $B; \Gamma \vdash_c J$. When a variable is bound by the environment, it must be added to the colour: $\Gamma, x : T \vdash J$ becomes $B; \Gamma, x :_c T \vdash_{c \cup \{x\}} J$. Substitutions must also be decorated with the appropriate colours. Typical examples are given by the rules $(\mathcal{C}/\mathsf{et.fun})$ et $(\mathcal{C}/\mathsf{et.app})$ given below:

$$\frac{\mathsf{B}; \Gamma, x :_{c} \mathsf{T}_0 \vdash_{c \cup \{x\}} \mathsf{E} :^{\gamma} \mathsf{T}_1}{\mathsf{B}; \Gamma \vdash_{c} \lambda x : \mathsf{T}_0.\ \mathsf{E} :^{\mathsf{P}} \Pi x : \mathsf{T}_0.\ ^{\gamma}\mathsf{T}_1}\ (\mathcal{C}/\mathsf{et.fun}) \qquad \frac{\mathsf{B}; \Gamma \vdash_{c} \mathsf{E}_1 :^{\gamma_1} \Pi x : \mathsf{T}_0.\ ^{\gamma_2}\mathsf{T} \qquad \mathsf{B}; \Gamma \vdash_{c} \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0}{\mathsf{B}; \Gamma \vdash_{c} \mathsf{E}_1\, \mathsf{E}_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow_{c} \mathsf{E}_0\}\mathsf{T}}\ (\mathcal{C}/\mathsf{et.app})$$

In addition to the addition of colours, type correction judgements $\mathsf{T}\,\mathsf{ok}$ now become type kinding judgements $\mathsf{T} : *$. Contexts that required an expression of type TYPE now require TYPE*. The complete list of adapted rules is the following:

- all conversion, convertibility and subtyping rules: ($\mathcal{C}$/econv.cong.fun.arg), ($\mathcal{C}$/econv.cong.fun.body), ($\mathcal{C}$/econv.cong.app.fun), ($\mathcal{C}$/econv.cong.app.arg), ($\mathcal{C}$/econv.cong.pair.1), ($\mathcal{C}$/econv.cong.pair.2), ($\mathcal{C}$/econv.cong.field), ($\mathcal{C}$/econv.cong.proj), ($\mathcal{C}$/econv.app), ($\mathcal{C}$/econv.proj), ($\mathcal{C}$/econv.eta.field), ($\mathcal{C}$/econv.eta.fun), ($\mathcal{C}$/econv.eta.pair), ($\mathcal{C}$/tconv.cong.pair.1), ($\mathcal{C}$/tconv.cong.pair.2), ($\mathcal{C}$/tconv.cong.fun.arg), ($\mathcal{C}$/tconv.cong.fun.ret), ($\mathcal{C}$/tconv.cong.sing), ($\mathcal{C}$/tconv.cong.field), ($\mathcal{C}$/tconv.field), ($\mathcal{C}$/tconv.unit), ($\mathcal{C}$/eeq.refl), ($\mathcal{C}$/eeq.sym), ($\mathcal{C}$/eeq.trans), ($\mathcal{C}$/eeq.conv), ($\mathcal{C}$/teq.refl), ($\mathcal{C}$/teq.sym), ($\mathcal{C}$/teq.trans), ($\mathcal{C}$/teq.conv), ($\mathcal{C}$/tsub.trans), ($\mathcal{C}$/tsub.eq), ($\mathcal{C}$/tsub.cong.fun), ($\mathcal{C}$/tsub.cong.pair), ($\mathcal{C}$/tsub.sing);

- most expression typing rules: ($\mathcal{C}$/et.base.unit), ($\mathcal{C}$/et.base.bool), ($\mathcal{C}$/et.base.int), ($\mathcal{C}$/et.fun), ($\mathcal{C}$/et.app), ($\mathcal{C}$/et.pair), ($\mathcal{C}$/et.proj.1), ($\mathcal{C}$/et.proj.2), ($\mathcal{C}$/et.let), ($\mathcal{C}$/et.sub), ($\mathcal{C}$/et.sing).

Appendix A contains a complete list of the rules of system $\mathcal{C}$ in their final form, including inherited rules.

IV.5.5.1 $\boxed{\mathsf{B}; \Gamma \vdash_{c} \mathsf{ok}}$ Environment formation

We describe how to build lexes, environments and colours. These components are built from left to right, both inside lexes and environments (which are built binding by binding from left to right) and in that the lexis is built first, then the environment, then the colour.

Lexis and environment validity follow a similar principle: all entered information must be checked for validity, and a fresh name must be used to label each binding. Note that the colour of a lexis binding may include previous nonces, while that of an environment binding may use any nonce in the lexis as well as previous variables.

$$\frac{}{\mathsf{nil}; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\ \textbf{(}\mathcal{C}\textbf{/envok.nil)} \qquad \frac{\mathsf{B}; \mathsf{nil} \vdash_{c_0} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \text{when } a \notin \mathbf{dom}\, \mathsf{B}}{\mathsf{B}, a = \mathsf{E} :_{c_0} \mathsf{T}; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\ \textbf{(}\mathcal{C}\textbf{/envok.a)}$$

$$\frac{\mathsf{B}; \Gamma \vdash_{c} \mathsf{T} : * \qquad \text{when } x \notin \mathbf{dom}\, \Gamma}{\mathsf{B}; \Gamma, x :_{c} \mathsf{T} \vdash_{\bullet} \mathsf{ok}}\ \textbf{(}\mathcal{C}\textbf{/envok.x)}$$

A colour may contain nonces and variables taken respectively from the lexis and the environment. As a colour is an unordered set, there is no constraint on the order in which a colour is built (other than the validity of intermediate colours when adding a nonce). Nonces may only be added to the colour if their dependencies are already present (or otherwise transparent), while variables may be added at any time — see section IV.5.2.3 and the transparency rules below).

$$\frac{\mathsf{B}; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{when } a = \mathsf{E} :_{c_0} \mathsf{T} \in \mathsf{B} \wedge c_0 \subseteq c'}{\mathsf{B}; \Gamma \vdash_{c' \cup \{a\}} \mathsf{ok}}\ \textbf{(}\mathcal{C}\textbf{/envok.c.a)}$$

$$\frac{\mathsf{B}; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{when } x :_{c_0} \mathsf{T} \in \Gamma}{\mathsf{B}; \Gamma \vdash_{c' \cup \{x\}} \mathsf{ok}}\ \textbf{(}\mathcal{C}\textbf{/envok.c.x)}$$

IV.5.5.2 $\boxed{\mathsf{B}; \Gamma \vdash_{c} \mathsf{T} : \mathsf{K}}$ Type kinding

Type kinding rules refine the type correction rules of system $\mathcal{E}$. Adding colours is straightforward.

As discussed in section IV.5.3.6, a generative functor is always monomorphic (whereas an applicative functor has the same kind as its result type); the rule ($\mathcal{E}$/tok.fun) is split to treat each case correctly.

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{bool} : \wr}\ (\mathcal{C}/\textbf{tok.base.bool}) \qquad \frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{int} : \wr}\ (\mathcal{C}/\textbf{tok.base.int}) \qquad \frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{unit} : \wr}\ (\mathcal{C}/\textbf{tok.base.unit})$$

$$\frac{B;\Gamma \vdash_c E :^{P} \textsc{type}^{K}}{B;\Gamma \vdash_c \mathsf{Typ}\,E : K}\ (\mathcal{C}/\textbf{tok.field})$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Pi x : T'.\,^{P}T'' : K''}\ (\mathcal{C}/\textbf{tok.fun.P})$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Pi x : T'.\,^{I}T'' : \wr}\ (\mathcal{C}/\textbf{tok.fun.I})$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Sigma x : T'.\,T'' : K' \vee K''}\ (\mathcal{C}/\textbf{tok.pair})$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{type}^{K} : *}\ (\mathcal{C}/\textbf{tok.type}) \qquad \frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c S(E) : \wr}\ (\mathcal{C}/\textbf{tok.sing})$$

An extra rule indicates that any monomorphic type is also polymorphic. Similarly a type field containing a monomorphic type can be seen as a type field containing a polymorphic type, so the type of the formed is a subtype of the type of the latter.

$$\frac{B;\Gamma \vdash_c T : K' \qquad \text{when } K' \leqslant K}{B;\Gamma \vdash_c T : K}\ (\mathcal{C}/\textbf{tok.sub}) \qquad \frac{B;\Gamma \vdash_c \mathsf{ok} \qquad \text{when } K_1 \leqslant K_2}{B;\Gamma \vdash_c \textsc{type}^{K_1} <: \textsc{type}^{K_2}}\ (\mathcal{C}/\textbf{tsub.cong.type})$$

Let us also state the rule for forming a type field, which is also modified to account for kinding.

$$\frac{B;\Gamma \vdash_c T : K}{B;\Gamma \vdash_c \langle T\rangle :^{P} \textsc{type}^{K}}\ (\mathcal{C}/\textbf{et.type})$$

IV.5.5.3 $\boxed{B;\Gamma \vdash_c c_0\ \mathsf{transparent}}$ Colour transparency

A primary colour (nonce or variable) can be transparent if it is directly present in the ambient colour. It can also be transparent if it is indirectly made so, via a variable that is present in the ambient colour and whose colour of definition makes the primary colour under consideration transparent.

$$\frac{B;\Gamma \vdash_c \mathsf{ok} \qquad \text{when } \xi \in c}{B;\Gamma \vdash_c \xi\,\mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.in})$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok} \qquad B;\Gamma_0 \vdash_{c_0} \xi\,\mathsf{transparent} \qquad \text{when } \Gamma = (\Gamma_0, x :_{c_0} T, \Gamma_1) \wedge x \in c}{B;\Gamma \vdash_c \xi\,\mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.env})$$

A colour is transparent if and only if all of its elements are transparent.

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \bullet\ \mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.o}) \qquad \frac{B;\Gamma \vdash_c c_1\,\mathsf{transparent} \qquad B;\Gamma \vdash_c c_2\,\mathsf{transparent}}{B;\Gamma \vdash_c c_1 \cup c_2\,\mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.union})$$

Nonce transparency is used in ($\mathcal{C}$/tconv.abs) to justify revealing it. Colour transparency is used in several rules (($\mathcal{C}$/ac.a), ($\mathcal{C}$/et.x)) to express the transparency of the dependencies of a primary colour.

IV.5.5.4 $\boxed{B;\Gamma \vdash_c A \triangleright E : T\ ;\ \ldots}$ Module components

Revelation judgements $B;\Gamma \vdash_c A \triangleright E : T$ assign two pieces of information to a component A: the expression E to which it is revealed, and the apparent signature T mechanically derived from the signature of the underlying nonce in the lexis. The structure of the revelation derivation follows that of this signature.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{c}_0\,\mathsf{transparent} \qquad \text{when } \mathsf{a} = \mathsf{E} :_{\mathsf{c}_0} \mathsf{T} \in \mathsf{B}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{a} \triangleright \mathsf{E} : \mathsf{T}}\ (\mathcal{C}/\textsf{ac.a}) \qquad \frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_1 \mathsf{A} \triangleright \pi_1 \mathsf{E} : \mathsf{T}_1}\ (\mathcal{C}/\textsf{ac.proj.1})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_1 :^{\mathsf{P}} \mathsf{S}(\pi_1 \mathsf{E}) \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_2 \mathsf{A} \triangleright \pi_2 \mathsf{E} : \{x \leftarrow_{\mathsf{c}} \mathsf{E}_1\}\mathsf{T}_2}\ (\mathcal{C}/\textsf{ac.proj.2})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1 \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A}\,\mathsf{E}_0 \triangleright \mathsf{E}\,\mathsf{E}_0 : \{x \leftarrow_{\mathsf{c}} \mathsf{E}_0\}\mathsf{T}_1}\ (\mathcal{C}/\textsf{ac.app})$$

When a component has the apparent signature $\textsc{type}^{\mathsf{K}}$, it can be used to form an abstract type. If the underlying nonce is transparent, this abstract type can be converted to the revealed representation.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \textsc{type}^{\mathsf{K}}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\!|\mathsf{A}|\!) : \mathsf{K}}\ (\mathcal{C}/\textsf{tok.abs})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \textsc{type}^{\mathsf{K}} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathbf{underl}(\mathsf{A})\,\mathsf{transparent}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\!|\mathsf{A}|\!) \longrightarrow \mathsf{Typ}\,\mathsf{E}}\ (\mathcal{C}/\textsf{tconv.abs})$$

A component is almost inert: the only conversion that might significantly affect it is its revelation. Context rules are however needed to enable conversion of embedded expressions.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 \longrightarrow \mathsf{E}_0' \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0 \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A}\,\mathsf{E}_0 \longrightarrow \mathsf{A}\,\mathsf{E}_0'}\ (\mathcal{C}/\textsf{aconv.cong.app.arg})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \longrightarrow \mathsf{A}' \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1 \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A}\,\mathsf{E}_0 \longrightarrow \mathsf{A}'\,\mathsf{E}_0}\ (\mathcal{C}/\textsf{aconv.cong.app.fun})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \longrightarrow \mathsf{A}' \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_i \mathsf{A} \longrightarrow \pi_i \mathsf{A}}\ (\mathcal{C}/\textsf{aconv.cong.proj})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \longrightarrow \mathsf{A}' \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \textsc{type}^{\mathsf{K}}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\!|\mathsf{A}|\!) \longrightarrow (\!|\mathsf{A}'|\!)}\ (\mathcal{C}/\textsf{tconv.cong.abs})$$

Convertibility equivalence for components follows the same model as for types and expressions, with four rules ($\mathcal{C}$/aeq.refl), ($\mathcal{C}$/aeq.sym), ($\mathcal{C}$/aeq.trans) and ($\mathcal{C}$/aeq.conv) following the model of (teq.*) and (eeq.*).

IV.5.5.5 $\boxed{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\gamma} \mathsf{T}}$ Coloration of expressions

The rule for typing variables contains a novel side condition which requires the transparency of the variable in the ambient colour. This ensures that the dependencies of the variable keep being present if the ambient colour is weakened.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} x\,\mathsf{transparent} \qquad \text{when } x :_{\mathsf{c}'} \mathsf{T} \in \Gamma}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} x :^{\mathsf{P}} \mathsf{T}}\ (\mathcal{C}/\textsf{et.x})$$

Coloured brackets surround an expression with a different colour from its surroundings. The type annotation must be valid both in both the outer and inner colours, for which we use the intersection of the two colours.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{E} :^{\gamma} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c} \cap \mathsf{c}'} \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}}\ \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} [\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}} :^{\gamma} \mathsf{T}}\ (\mathcal{C}/\textsf{et.col})$$

The sealing typing rule takes the colour annotation into account. The colour is added when typing the body of the module.

$$\frac{B;\Gamma \vdash_{c \cup c'} E :^{\gamma} T \qquad B;\Gamma \vdash_c T : *}{B;\Gamma \vdash_c (E\ !!_{c'}\ T) :^{I} T}\ (\mathcal{C}\text{/et.seal})$$

IV.5.5.6 $\boxed{B;\Gamma \vdash_c E \longrightarrow E'}$ Conversion and coloured brackets

New conversion rules reflect the new reduction rules concerning brackets: new reduction contexts, and bracket pushing rules.

$$\frac{B;\Gamma \vdash_{c'} E \longrightarrow E' \qquad B;\Gamma \vdash_{c'} E :^{P} T \qquad B;\Gamma \vdash_{c \cap c'} T : \wr \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [E]^{T}_{c'} \longrightarrow [E']^{T}_{c'}}\ (\mathcal{C}\text{/econv.cong.col.e})$$

$$\frac{B;\Gamma \vdash_{c'} E :^{P} T_1 \qquad B;\Gamma \vdash_{c \cap c'} T_1 \longrightarrow T_2 \qquad B;\Gamma \vdash_{c \cap c'} T_1 : \wr \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [E]^{T_1}_{c'} \longrightarrow [E]^{T_2}_{c'}}\ (\mathcal{C}\text{/econv.cong.col.t})$$

$$\frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [()]^{\mathrm{UNIT}}_{c'} \longrightarrow ()}\ (\mathcal{C}\text{/econv.col.base.unit}) \qquad \frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [bv]^{\mathrm{BOOL}}_{c'} \longrightarrow bv}\ (\mathcal{C}\text{/econv.col.base.bool})$$

$$\frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [\underline{n}]^{\mathrm{INT}}_{c'} \longrightarrow \underline{n}}\ (\mathcal{C}\text{/econv.col.base.int})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \qquad B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{P} T_1 \\ B;\Gamma \vdash_c \mathsf{ok} \qquad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_c [\lambda x : T_2.\ E]^{\Pi x : T_0.\, ^{P} T_1}_{c'} \longrightarrow \lambda x : T_0.\ [E]^{T_1}_{c' \cup \{x\}}}\ (\mathcal{C}\text{/econv.col.fun.P})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \qquad B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{I} T_1 \\ B;\Gamma \vdash_c \mathsf{ok} \qquad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_c [\lambda x : T_2.\ E]^{\Pi x : T_0.\, ^{I} T_1}_{c'} \longrightarrow \lambda x : T_0.\ E\ !!_{c' \cup \{x\}}\ T_1}\ (\mathcal{C}\text{/econv.col.fun.I})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} E_1 :^{P} T_1 \qquad B;\Gamma \vdash_{c \cap c'} T_1 : \wr \qquad B;\Gamma, x :_{c'} T_1 \vdash_{c' \cup \{x\}} E_2 :^{P} T_2 \\ B;\Gamma \vdash_c \mathsf{ok} \qquad B;\Gamma, x :_{c \cap c'} T_1 \vdash_{(c \cap c') \cup \{x\}} T_2 : \wr \qquad B;\Gamma \vdash_{c'} E_2 :^{P} \{x \leftarrow_{c'} [E_1]^{T_1}_{c'}\} T_2 \end{array}}{B;\Gamma \vdash_c [(E_1, E_2)]^{\Sigma x : T_1.\, T_2}_{c'} \longrightarrow ([E_1]^{T_1}_{c'}, [E_2]^{\{x \leftarrow_{c'} [E_1]^{T_1}_{c'}\} T_2}_{c'})}\ (\mathcal{C}\text{/econv.col.pair})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c_2} E :^{P} T_2 \qquad B;\Gamma \vdash_{c_1 \cap c_2} T_2 : \wr \\ B;\Gamma \vdash_{c_1} T_2 <: T_1 \qquad B;\Gamma \vdash_{c \cap c_1} T_1 : \wr \qquad B;\Gamma \vdash_c \mathsf{ok} \end{array}}{B;\Gamma \vdash_c [[E]^{T_2}_{c_2}]^{T_1}_{c_1} \longrightarrow [E]^{T_1}_{c_1 \cup c_2}}\ (\mathcal{C}\text{/econv.col.merge})$$

$$\frac{B;\Gamma \vdash_{c'} E' :^{P} S(E) \qquad B;\Gamma \vdash_{c \cap c'} E :^{P} T \qquad B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c [E']^{S(E)}_{c'} \longrightarrow E}\ (\mathcal{C}\text{/econv.col.sing})$$

$$\frac{\begin{array}{c} B;\Gamma, x :_c T_0 \vdash_{c'} E_1 :^{\gamma} T_1 \quad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 \longrightarrow T_1' \\ B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1\ !!_{c'}\ T_1) \longrightarrow (\lambda x : T_0.\ E_1\ !!_{c'}\ T_1')}\ (\mathcal{C}\text{/econv.cong.fun.seal})$$

IV.6 Dynamic typing and distributed programs $\boxed{\mathcal{D}}$

IV.6.1 Dynamic typing

[Sorry, this fragment has not been translated yet.]

IV.6.2 Formalisation

IV.6.2.1 Syntax

We define a new language, system $\mathcal{D}$, which is a conservative extension of system $\mathcal{C}$. The new features are the type DYN and two constructors and a destructor for this type.

T ::=	**type**
...	
DYN	dynamically typed values

E ::=	**expression**
...	
$\mathsf{dyn\ E\ at\ T}$	dynamic
$\mathsf{dynned\ E\ at\ T}$	universal dynamic
$\mathsf{undyn\ E\ at\ T\ else\ E'}$	dynamic type verification

IV.6.2.2 Reduction

The universal dynamic of a value is a value. The new constructors and destructor are evaluation contexts.

V ::=	**quasi-value**
...	
$\mathsf{dynned\ V^{\bullet}\ at\ T}$	universal dynamic

$\mathsf{V^c}$ ::=	**value in** c
...	
$\mathsf{dynned\ V^{\bullet}\ at\ T}$	universal dynamic

$\mathsf{C^{c}_{c'}}$ ::=	**evaluation context with inner colour** $\mathsf{c'}$ **and outer colour** c
...	
$\mathsf{dyn\ \_\ at\ T}$	dynamic
$\mathsf{dynned\ \_\ at\ T}$	universal dynamic, when $\mathsf{c'} = \bullet$
$\mathsf{undyn\ \_\ at\ T\ else\ E'}$	dynamic type verification

We saw in section IV.6.1.4 how to produce a universal dynamic from a dynamic. Evaluating a dynamic type verification can either result in accepting the underlying value if the types are compatible, or evaluating the alternate expression otherwise. A new rule lets a bracket be pushed into a dynamic. A coloured bracket around a universal dynamic can simply be erased, since its contents are already protected.

$$\mathsf{dyn\ V^c\ at\ T} \longrightarrow_{\mathsf{c}} \mathsf{dynned\ [V^c]_c^{\mathbf{conc}_c^B(T)}\ at\ \mathbf{conc}_c^B(T)} \qquad (\mathcal{D}/\mathsf{ered.dyn})$$

$$\mathsf{B} \vdash \mathsf{undyn\ (dyn\ V^c\ at\ T)\ at\ T'\ else\ E'} \longrightarrow_{\mathsf{c}} \mathsf{B} \vdash \begin{cases} \mathsf{V^c} & \text{if } \mathsf{B; nil} \vdash_{\mathsf{c}} \mathsf{T <: T'} \\ \mathsf{E'} & \text{otherwise} \end{cases} \qquad (\mathcal{D}/\mathsf{ered.undyn})$$

$$\mathsf{[dynned\ V^{\bullet}\ at\ T]_{c'}^{DYN}} \longrightarrow_{\mathsf{c}} \mathsf{dynned\ V^{\bullet}\ at\ T} \qquad (\mathcal{D}/\mathsf{ered.col.dynned})$$

IV.6.2.3 Typing

As we saw in section IV.6.1.4, ordinary dynamics $\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$, but universal dynamics $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ (which are only useful with E pure) are pure. We do not wish to have to manage dynamic typing inside a compiler[19], thus we declare that any expression of the form $\mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}\,\mathsf{else}\,\mathsf{E}'$ is impure.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathrm{DYN} : \mathsf{K}}\;(\mathcal{D}/\textbf{tok.base.dyn}) \qquad \frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\Upsilon} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \wr}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} :^{\mathrm{I}} \mathrm{DYN}}\;(\mathcal{D}/\textbf{et.dyn})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\bullet} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} :^{\mathsf{P}} \mathrm{DYN}}\;(\mathcal{D}/\textbf{et.dynned})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\Upsilon} \mathrm{DYN} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}' :^{\Upsilon} \mathsf{T}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}\,\mathsf{else}\,\mathsf{E}' :^{\mathrm{I}} \mathsf{T}}\;(\mathcal{D}/\textbf{et.undyn})$$

Since the language has a new constructor, we need corresponding conversion rules: congruence rules to rewrite the arguments of the constructor, and a bracket pushing rule (reflecting the reduction rule $(\mathcal{D}/\mathsf{ered.col.dynned})$).

$$\frac{\mathsf{B};\Gamma \vdash_{\bullet} \mathsf{E} \longrightarrow \mathsf{E}' \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} \longrightarrow \mathsf{dynned}\,\mathsf{E}'\,\mathsf{at}\,\mathsf{T}}\;(\mathcal{D}/\textbf{econv.cong.dynned.e})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\bullet} \mathsf{T} \longrightarrow \mathsf{T}' \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} \longrightarrow \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}'}\;(\mathcal{D}/\textbf{econv.cong.dynned.t})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\bullet} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\bullet} \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{ok} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} [\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}]_{\mathsf{c}'}^{\mathrm{DYN}} \longrightarrow \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}}\;(\mathcal{D}/\textbf{econv.col.dynned})$$

The dynamisation function Our typing rules let us write any monomorphic dynamisation function $\lambda \mathsf{x}{:}\mathsf{T}.\ \mathsf{dyn}\,\mathsf{x}\,\mathsf{at}\,\mathsf{T}$, with the type $\mathsf{T} \rightarrow^{\mathrm{I}} \mathrm{DYN}$. They also let us write the polymorphic dynamisation function $\lambda \mathsf{t} : \mathrm{TYPE}^{\wr}.\ \lambda \mathsf{x} : \mathsf{Typ}\,\mathsf{t}.\ \mathsf{dyn}\,\mathsf{x}\,\mathsf{at}\,\mathsf{Typ}\,\mathsf{t}$, with the type $\Pi \mathsf{t} : \mathrm{TYPE}^{\wr}.{}^{\mathsf{P}}\,\mathsf{Typ}\,\mathsf{t} \rightarrow^{\mathrm{I}} \mathrm{DYN}$. Applying one of these dynamisation function produces a value of the form $\mathsf{dynned}\,[\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}\ \mathsf{at}\ \mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ where c is the ambient colour.

IV.6.3 Communication inter-machines

IV.6.3.1 Introduction

The present dissertation was motivated by the need for dynamic type-checkin in distributed programs with abstract types. The present chapter has so far mainly dealt with abstract types, and we have now introduced dynamic type-checking. We now add the last ingredient: inter-machine communication.

As in chapter II, we assume the availability of some serialisation mechanism to send values between programs running on different machines. In the present discussion, we deal with networked programs, but many considerations also apply to time- rather than space-separated programs, i.e., a program writing data to persistent storage and another program later reading the data.

In order for a value sent by a machine A to be correctly received and decoded on a machine B, the programs running on the two machines must agree on their interpretations of the bit strings they exchange. We assume that all programs are written in the same language and use the same serialisation library, so that it suffices to ensure that the exchanged values do not depend on any manner of environment that is not shared by the two machines. Our semantics does contain one

[19]Not only would it be useless, it would also work around a putative stratification (see section V.3.1.1).

machine-dependent element: abstract types defined on one machine may not be available on another machine.

As a first step, we will make the soundness-safe assumption that abstract types defined on one machine are distinct from abstract types defined on any other machine, i.e., abstract types are globally fresh. We spent most of chapter II studying how to lift this restriction, and we will see in section IV.6.3.5 how to integrate these ideas into system $\mathcal{D}$.

IV.6.3.2 Communication and colours

In this section, we assume the existence of two primitives `send` and `recv` for respectively sending and receiving a value. Communication can take place on a network, via temporary storage or by any other means. More precisely, since we are working in a typed language, we will assume two type-indexed families of primitives $\mathtt{send}^{\mathsf{T}}$ and $\mathtt{recv}^{\mathsf{T}}$, the type T being that of transmitted values; their types are $\mathtt{send}^{\mathsf{T}} : \mathsf{T} \to^{\mathtt{I}} \textsc{unit}$ et $\mathtt{recv}^{\mathsf{T}} : \textsc{unit} \to^{\mathtt{I}} \mathsf{T}$. In order for communications to respect typing, the communication protocol must ensure that values sent by $\mathtt{send}^{\mathsf{T}}$ will only ever be received by $\mathtt{recv}^{\mathsf{T}'}$ when it can be guaranteed that any value of type T also has the type T', which we model with the constraint $\mathsf{T} <: \mathsf{T}'$.

One thorny issue is that T and T' live in different contexts: the sending locus and the reception locus may have different knowledge of abstract types. In our framework, this means that the ambiant colour might differ between sending and reception. As we saw in section IV.5.3.1, the colour influences both the validity and the semantics of a type. This also applies to the transmitted value, which may have the type T in the sending colour c without having the type T, or indeed any type, in the reception colour c'.

One way to ensure the safety of communication is to index the primitives by a colour as well as a type, i.e., $\mathtt{send}^{\mathsf{T}}_{\mathsf{c}}$ and $\mathtt{recv}^{\mathsf{T}'}_{\mathsf{c}'}$, and require that the communication protocol ensure colour compatibility $\mathsf{c} \subseteq \mathsf{c}'$ (or rather more precisely $\Gamma \vdash_{\mathsf{c}'} \mathsf{c}\,\mathsf{transparent}$) as well as type compatiblity $\Gamma \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T}'$. However adapting a communication protocol to ensure colour compatibility is not straightforward, all the less as the ambiant colour of a value may change as it is passed around (whereas the type annotations T and T' are usually known statically).

One way to prevent any incompatibility from occurring is to require that the sending colour be empty, in other words that the sending type T as well as the transmitted value be *universal*. We will study how to achieve this in section IV.6.3.3.

If we wish to transmit values between arbitrary colours, they need to be protected. We encountered a similar situation in section IV.5.2.2: given a value V and a type T, we need to construct a value that is "equivalent" to V and has a type "equivalent" to T in the empty colour $\bullet$. The solution is to use concretisation and send $[\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}$ instead of V, where $\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ is the type T with uses of c expanded out. The extra coloured bracket protects V by bestowing upon it any type equation that it may need. Having obtained the universal value $[\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}$, we can safely send it to any receiver for the type $\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$.

IV.6.3.3 Universals

[Sorry, this fragment has not been translated yet.]

IV.6.3.4 Nonce sharing

In section IV.6.3.2, we discussed how to send values between different colours. We glossed over the fact that a colour is only defined in a certain lexis — comparing colours defined in different lexes, or transmitting a colour from one lexis to another, does not a priori make sense. However in a network of machines, each machine would have its own lexis.

We modelled the execution of a program (consisting of a single thread running on a given machine) by a reduction relation of the form $B \vdash E \longrightarrow_{\bullet} B' \vdash E'$ (at the top level, outside of any bracket, the ambiant colour is empty). An immediate generalisation to networked programs leads us to consider a family of reductions $B_i \vdash E_i \longrightarrow_{\bullet} B'_i \vdash E'_i$ where the index i represents the machine on which the reduction takes place. In this model, communication must take into account the change of lexis from B_i to B_j as well as the colour change.

Recall that a lexis is a set of nonces (plus some information about these nonces), and each nonce that is added to the lexis is freshly created (by the rule ($\mathcal{C}$/ered.seal) and globally unique). Two lexes B_1 and B_2 formed on different machines are therefore disjoint; it is clear that (B_1, B_2), or indeed any lexis made by interleaving the elements of B_1 with the elements of B_2 is also a well-formed lexis, with the same information stored for each nonce as in B_1 or B_2. In the metatheory, we can just merge lexes and model the evolution of a networked program by a reduction relation of the form

$$B \vdash E_1 \parallel \ldots \parallel E_n \longrightarrow B' \vdash E'_1 \parallel \ldots \parallel E'_n$$

(where $E_1 \parallel \ldots \parallel E_n$ notes the parallel composition of n expressions, each running on its own machine).

This model makes considerations about mixing lexes moot as far as the metatheory is concerned. However, in practice, requiring each nonce to be broadcast as soon as it is created would be prohibitively expensive (and might be impossible in networks with complex dynamic topologies). Fortunately one can easily implement the shared lexis model by considering that each machine only has a partial copy of the global lexis at any time, and requiring every transmission of a value to also contain any information necessary to reconstruct the parts of the lexis that the value depends on (that is, the nonces contained in a value as well as their dependencies). Thus the lexis is spread around lazily. Note that although nonce creation requires the generation of a globally unique name, this does not in practice require synchronisation: it suffices that each machine have a globally unique name that can be included in the nonce, which is the case in most distributed systems.

IV.6.3.5 Static sealing and hashes

Nonces are singularised identities in the sense of section II.6.1.2, as a fresh nonce is generated whenever a new family of abstract types is created by evaluating a dynamic sealing construct $E \mathbin{!!} T$. In section IV.4.4.2, we presented system $\mathcal{W}$, which has another notion of sealing, namely static sealing $E :: T$. Unlike dynamic sealing, static sealing creates a new family of abstract types once and for all at program compile- or initialisation-time, and thus requires an identity to be generated at the corresponding time.

In a distributed environment, there are several choices as to when to generate identities for statically sealed modules. The two main possibilities, compile- and initialisation-time, give different results.

Generating stamps at compile-time [Mac84] is one traditional way of obtaining comparable designations of abstract types. This is not suitable when the identity of a type depends on the run-time behaviour of the program, but this is never the case with our static sealing. Some module systems for distributed programs [Sew01] explicitly allow for abstract type generation at compile-time. This feature has a grave practical defect, namely the impossibility of reconstructing a program from its source alone. If two instances of the same program are deployed, they will only have compatible types if they stem from the same compilation, not if the program was distributed in source form. For this reason, we choose not to support any way to generate module identities at compile-time.

Generating new identities at program-initialisation time allows for less compatibility than at compile-time. However the behaviour is easily predictable and reproducible: any excessive generativity can be spotted in testing. Thus we propose that this is a viable semantics for static sealing.

None of the generation semantics described so far allows sharing abstract types between independently compiled instances of the same program (let alone independently deployed instances of a program component). Yet most cases where static sealing is used — often to enforce data structure invariants — correspond to cases where structural module identities, i.e., hash*hashes* are desired (see section II.3). It is therefore natural to designate statically sealed modules by their hash. The identities are defined by a purely mathematical computation and therefore reproducible at will. As in HAT (see section III.2.7.5), we can see hashes as unifying separate definitions of the "same" module on different machines.

We shall not describe hash formation for system $\mathcal{W}$ formally here. This construction requires that the statically sealed module be lifted from its local potentially-generative context as described in section IV.4.4.5. Note that in system $\mathcal{W}$, unlike in HAT, the identity of a type is an arbitrarily-sized term which may mention more than one hash. For example, if f is a statically sealed module with the signature $\Pi x : T_0.\, \Sigma t : \text{TYPE}.\, T_1$ that is applied to a dynamically sealed module which was given the nonce a, the identity of the type field in the resulting module is $\pi_1 (h\, a)$ where h is the hash of the functor.

IV.7 Conclusion

Summary In the present chapter we presented a description language called TOPHAT for a module system for an ML-like language. The main features of this language are:

- structures and functors, whose types are respectively dependent sums and dependent products;
- a way to test the equivalence of two modules, and propagate knowledge of such an equivalence, using singleton signatures;
- abstract types can be defined by sealing a module, and an effect system determines which expressions remain comparable;
- an reduction abstraction-preserving, thanks to coloured brackets;
- a dynamic type-checking construct that does not depend on the program context.

Soundness The most basic requirement for a type system is that it for the proposed execution mechanism. Appendix B contains a soundness proof for TOPHAT, classically formulated as two theorems: type preservation by reduction (**??** ()) and progress of well-typed expressions (**??** ()).

Decidability of type-checking Another expected property of a type system for a programming language is decidability, i.e., we would like an algorithm for deciding whether a given expression has a given type[20]. In particular, we would need to decide when two types are equivalent. Decision procedures exist for weaker type systems, in particular the one proposed by Dreyer, Crary and Harper [DCH03]. However their algorithm does not easily generalise to our system, and we regretfully leave the question open.

[20]**Type inference** would in fact be desirable. However inference is known to be undecidable in much weaker type systems such as system F. With the type annotations that we require in the syntax, in particular on function arguments, type reconstruction might not be substantially harder than verification.

Chapter V

Conclusion

V.1 Summary

[Sorry, this fragment has not been translated yet.]

V.2 Related work

V.2.1 Theoretic considerations

[[CM88], [OTCP90]]

V.2.2 Programming languages

[Modula-3, Java, .NET, Objective Caml]

V.2.3 Acute and HashCaml

[Sorry, this fragment has not been translated yet.]

V.2.4 Alice ML

Rossberg's work is to my knowledge the only other in-depth treatment of the main topic of this dissertation. Interestingly, my and his independent study of the problem led us towards the same tools.

Rossberg's first step [Ros03] was to use coloured brackets [ZGM99] to keep track of abstract types at run-time and obtain an abstraction-preserving reduction relation, in a manner similar to our HAT [LPSW03]. Our theories differ in that Rossberg's approach is purely generative: abstract types created on different machines are incompatible.

Rossberg also studied the generalisation from simple modules to a full-fledged ML module calculus [Ros07]. His implementation builds on Alice ML [PSL]. Rossberg defines the $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calculus, which models the core of Alice ML. This calculus includes a construct that defines an abstract type (§10.5–10.7), and he shows that this is equivalent to ML-like module sealing (specifically , see my section IV.4.4.2). An abstract type is identified by a type variable α with an *abstraction kind* (§11.3). An abstraction kind $A(\tau)$ is similar to a singleton kind $S(\tau)$ but only the singleton kind allows implicit conversion between α and τ. Abstraction-kinded type variables play the same role as my lexis-stored nonces. The type system of $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calculus allows explicit conversions between an abstract type and its representation type anywhere in the program, whereas I materialise such conversions with coloured brackets.

Rossberg proves an *opacity* property (§12.9): a program that does not contain any explicit conversion between an abstract type and its representation is parametric with respect to said representation. Rossberg also proposes a mechanism to seal functors (§13), allowing for both applicative and generative functors. Given the complexity of both systems, I leave to future work a comparison between the expressivity of $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calculus with functors and that of TOPHAT.

V.3 Future work

V.3.1 Improvements to the theory

V.3.1.1 Stratification

[indexing TYPE *with a universe]*

V.3.1.2 One or two language levels?

[Sorry, this fragment has not been translated yet.]

V.3.1.3 Effect analysis

[Sorry, this fragment has not been translated yet.]

V.3.1.4 Colours and brackets

[Sorry, this fragment has not been translated yet.]

V.3.1.5 Decidability of type-checking

[Sorry, this fragment has not been translated yet.]

V.3.1.6 Parametricity

[Sorry, this fragment has not been translated yet.]

V.3.2 Supplementary features

V.3.2.1 Field names and width subsignaturing

[Sorry, this fragment has not been translated yet.]

V.3.2.2 Towards a programming language

[polymorphism; recursion; libraries]

V.3.2.3 Generic programming

[Sorry, this fragment has not been translated yet.]

V.3.2.4 Security

[Sorry, this fragment has not been translated yet.]

V.3.3 Implementation

V.3.3.1 Hash computation

[Sorry, this fragment has not been translated yet.]

V.3.3.2 Typing TOPHAT

[Sorry, this fragment has not been translated yet.]

V.3.3.3 Integration into Objective Caml: the module system

Adding named structure fields and width subtyping to TOPHAT as described in section V.3.2.1 yields a language that covers all the features of the module calculus of Objective Caml [L$^+$]. But is our language compatible, i.e., is it a conservative extension of Objective Caml?

The answer is a qualified "no". There are programs that Objective Caml accepts and we reject, because Objective Caml treats every functor as applicative, even if its body contains side effects. This is unacceptable in TOPHAT as applications of applicative functors must be able to be statically evaluated. One way to improve compatibility would be to introduce a notion of separation (in the sense of separability [Dre05] as discussed in section IV.4.2.3). It is however debatable whether this is desirable: treating a functor whose application has side effects as applicative does not break structural typing but does not fully respect abstraction. We prefer to treat any functor whose body has side effects as generative because when applicativity is desired, the body is usually pure ([Dre05, RRS]). For example, all the functors in the standard library of Objective Caml have a pure body (mainly consisting of type definitions and immediate functions, as well as a few data structure values).

The existing sealing of Objective Caml should be considered a static sealing (see section IV.4.4.2. It would be desirable to add a dynamic sealing construct. Another necessary extension is syntax to mark a functor as generative (i.e., a purity annotation on functor types), in order for all signatures to be expressible in the source language.

In addition to examining the module language, we need to check for incompatibilities with the core language. We discussed polymorphism in section V.3.2.2. We can freely extend TOPHAT with impure constructs; a safe choice is to make almost all core expressions impure. The main requirements with respect to purity are that projecting a field of a module and immediate functions must be considered pure. In fact, Objective Caml (like any implementation of Standard ML) already performs a suitable purity analysis, in order to check the value restriction for polymorphism [Wri95, Gar04].

V.3.4 Applications of dynamic typing

[Sorry, this fragment has not been translated yet.]

V.3.4.1 The JoCaml name server

The JoCaml "name server" was one of the main motivations of this work. The JoCaml language [FLFS07, MM01] is statically typed, including communications [FLMR97]. However this result only applies inside a single program instance: when two separate instances communicate, the fact that the value sent by one instance has the type expected by the other instance cannot result solely from adherence to a protocol that only allows for verification inside a single instance, which static typechecking is.

The recommended programming methodology for JoCaml keeps unsafe interactions to a minimum: one instance publishes a communication channel of an agreed-upon type, and other instances can send values (including other channels) over this initial channel, all communications but the initial reception of the public channel being type-safe. The JoCaml standard library provides a `Ns` module to assist in equipping depolyed programs with a name server. This name server is a particular program instance which acts as a database for communication channels (the names in question). Participating instances can publish their entry points by uploading them to the name server. A program instance that wants to join the network can query the name server to obtain a channel to send data on. The only type-checking that must take place at run-time is that performed by new participants as they check that the data returned by the name server matches their typing expectations (the actual verification may be performed by the name server itself; in any case the name server must retain typing information for the values that it stores).

Appendix A

Formal definition of TOPHAT

This appendix is a précis of the language TOPHAT, which is identical to $\mathcal{D}$ of chapter IV.

$\mathsf{E} ::=$	**expression**
$x \mid y \mid t \mid \ldots$	variables
$()$	unit value
$\mathsf{false} \mid \mathsf{true}$	boolean (generically bv)
$0 \mid 1 \mid \ldots$	integer (generically $\underline{n}$)
$\langle \mathsf{T} \rangle$	type field
$(\mathsf{E}_1, \mathsf{E}_2)$	pair
$\pi_i\, \mathsf{E}$	projection ($i \in \{1,2\}$)
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction
$\mathsf{E}_1\ \mathsf{E}_2$	application
$\mathsf{let}\ x = \mathsf{E}_0\ \mathsf{in}\ \mathsf{E} : \mathsf{T}$	local binding
$\mathsf{E}\ !!_c\ \mathsf{T}$	sealed and coloured module
$[\mathsf{E}]_c^{\mathsf{T}}$	coloured bracket
$\mathsf{dyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	dynamic
$\mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	universal dynamic
$\mathsf{undyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}\ \mathsf{else}\ \mathsf{E}'$	dynamic type verification
$\mathsf{T} ::=$	**type**
UNIT	unit
BOOL	booleans
INT	integers
$\mathsf{Typ}\ \mathsf{E}$	projection from a type field
$\Sigma x : \mathsf{T}_1.\ \mathsf{T}_2$	dependent sum (also written $\mathsf{T}_1 * \mathsf{T}_2$ when $x \notin \mathbf{fv}\,\mathsf{T}_2$)
$\Pi x : \mathsf{T}_0.\ ^{\gamma}\mathsf{T}_1$	dependent product (also written $\mathsf{T}_1 \rightarrow^{\gamma} \mathsf{T}_2$ when $x \notin \mathbf{fv}\,\mathsf{T}_1$)
$\mathsf{S}(\mathsf{E})$	singleton
TYPE^{K}	abstract type field
$(\!\lvert \mathsf{A} \rvert\!)$	abstract type
DYN	dynamically typed values
$\mathsf{K} ::=$	**kind**
$\wr$	monomorphic (fully specified)
$*$	polymorphic (partially specified)

$A ::=$	**module component**
a	nonce
$A\ E$	application
$\pi_i\, A$	projection ($i \in \{1, 2\}$)
$\gamma ::=$	**effect**
$\mathtt{P}$	pure
$\mathtt{I}$	impure
$\xi ::=$	**primary colour**
a	nonce
x	variable
$c ::=$	**colour**
$\bullet$	empty colour (also written $\{\}$)
$\{a_1, \ldots, a_k,\ x_1, \ldots, x_k\}$	finite set of primary colours
$B ::=$	**lexis**
nil	empty
$B, a = E :_{c_0} T$	nonce a with implemented by E with the signature T
$\Gamma ::=$	**environnement**
nil	empty
$\Gamma, x :_c T$	binding of the variable x
$J ::=$	**local judgement right-hand side**
ok	environment correction
$T : K$	type kinding (generalising $T : *$)
$T \longrightarrow T'$	typing conversion
$T \equiv T'$	convertibility equivalence on types
$E \longrightarrow E'$	expression conversion
$E \equiv E'$	convertibility equivalence on expressions
$T_1 <: T_2$	subtyping
$c_0\ \mathsf{transparent}$	colour transparency
$A \triangleright E : T$	component revelation
$A \longrightarrow A'$	component conversion
$A \equiv A'$	convertibility equivalence on components
$E :^{\gamma} T$	expression typing
$V ::=$	**quasi-value**
$() \mid bv \mid \underline{n}$	constant
$\langle T \rangle$	type field
(V_1, V_2)	pair
$\lambda x : T.\ E$	lambda-abstraction
$[V]_{c'}^{(\!|A^V|\!)}$	potentially abstraction-making coloured bracket
$\mathsf{dynned}\ V^{\bullet}\ \mathsf{at}\ T$	universal dynamic

$V^{c} ::=$	**value in** c
$() \mid bv \mid \underline{n}$	constant
$\langle T \rangle$	type field
(V_1^{c}, V_2^{c})	pair
$\lambda x : T.\ E$	lambda-abstraction
$[V^{c'}]_{c'}^{\llparenthesis A^{V^{c \cap c'}} \rrparenthesis}$	coloured bracket, if $A^{V^{c \cap c'}}$ is abstract in c but concrete in c'
$\mathsf{dynned}\ V^{\bullet}\ \mathsf{at}\ T$	universal dynamic

$A^{V} ::=$	**component value**
a	nonce
$A^{V}\ V$	application to a quasi-value
$\pi_i\, A^{V}$	projection ($i \in \{1, 2\}$)

$A^{V^{c}} ::=$	**abstract component in** c
a	nonce, if opaque in c
$A^{V^{c}}\ V^{c}$	application of a functor to a value
$\pi_i\, A^{V^{c}}$	projection ($i \in \{1, 2\}$)

$C_{c'}^{c} ::=$	**evaluation context with inner colour** c' **and outer colour** c
$E_1\ \_\_$	function argument
$\_\_\ V_2$	applied function
$(\_\_, E_2)$	first component of a pair
$(V_1, \_\_)$	second component of a pair
$\pi_i\ \_\_$	projection ($i \in \{1, 2\}$)
$\mathsf{let}\ x = \_\_\ \mathsf{in}\ E : T$	local bound
$\_\_\ !!_{c_1}\ T$	sealing
$[\_\_]_{c'}^{T}$	coloured bracket
$[V^{c_1}]_{c_1}^{\mathsf{Typ}\,\_\_}$	type field on a bracket, when $c' = c \cap c_1$
$\mathsf{dyn}\ \_\_\ \mathsf{at}\ T$	dynamic
$\mathsf{dynned}\ \_\_\ \mathsf{at}\ T$	universal dynamic, when $c' = \bullet$
$\mathsf{undyn}\ \_\_\ \mathsf{at}\ T\ \mathsf{else}\ E'$	dynamic type verification

$$
\begin{aligned}
\mathbf{self}^{\,BT}(A) &= BT && \text{if } BT \text{ is a base type (}\textsc{unit}, \textsc{bool}, \textsc{int}, \textsc{dyn}\text{)}\\
\mathbf{self}^{\,\Sigma x:T_1.\,T_2}(A) &= \Sigma x : \mathbf{self}^{\,T_1}(\pi_1 A).\ \mathbf{self}^{\,T_2}(\pi_2 A)\\
\mathbf{self}^{\,\Pi x:T_0.\,^{P}T_1}(A) &= \Pi x : T_0.\,^{P}(\mathbf{self}^{\,T_1}(A\,x))\\
\mathbf{self}^{\,\Pi x:T_0.\,^{I}T_1}(A) &= \Pi x : T_0.\,^{I}T_1\\
\mathbf{self}^{\,S(E')}(A) &= S(E)\\
\mathbf{self}^{\,\textsc{type}^{K}}(A) &= S(\langle \llparenthesis A \rrparenthesis \rangle)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{conc}_c^{B}(\llparenthesis A_1 \rrparenthesis) &= \mathsf{Typ}\,\mathbf{reveal}^{B}(A_1) && \text{if } \mathbf{underl}(A_1) \in c\\
\mathbf{conc}_c^{B}(\llparenthesis A_1 \rrparenthesis) &= \llparenthesis A_1 \rrparenthesis && \text{if } \mathbf{underl}(A_1) \notin c\\
\mathbf{conc}_c^{B}([E]_{c'}^{T}) &= [E]_{c'}^{\mathbf{conc}_{c \cap c'}^{B}(T)}
\end{aligned}
$$

(other cases by simple induction)

$$\begin{array}{rcll}
\{x{\leftarrow_{c_0}}E_0\}x &=& E_0 & \\
\{x{\leftarrow_{c_0}}E_0\}y &=& y & \text{si } y \neq x \\
\{x{\leftarrow_{c_0}}E_0\}[E]^{T}_{c} &=& [\{x{\leftarrow_{c_0}}E_0\}E]^{\{x\leftarrow_{c_0}E_0\}T}_{\{x\leftarrow_{c_0}E_0\}c} & \\
\{x{\leftarrow_{c_0}}E_0\}c &=& (c \setminus \{x\}) \cup c_0 & \text{si } x \in c \\
\{x{\leftarrow_{c_0}}E_0\}c &=& c & \text{si } x \notin c
\end{array}$$

(other cases follow the usual notion of capture-avoiding substitution)

$$\begin{array}{rcl}
\mathbf{underl}(a) &=& a \\
\mathbf{underl}(A\ E) &=& \mathbf{underl}(A) \\
\mathbf{underl}(\pi_i\, A) &=& \mathbf{underl}(A)
\end{array}$$

$$\begin{array}{rcll}
\mathbf{reveal}^{B}(a) &=& E & \text{where } a = E :_c T \in B \\
\mathbf{reveal}^{B}(A\ E) &=& (\mathbf{reveal}^{B}(A))\ E & \\
\mathbf{reveal}^{B}(\pi_i\, A) &=& \pi_i\,(\mathbf{reveal}^{B}(A)) &
\end{array}$$

$$\frac{B; \mathsf{nil} \vdash_{c_0} E :^{P} T \qquad \text{when } a \notin \mathbf{dom}\, B}{B, a = E :_{c_0} T; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\,\textbf{(envok.a)}
\qquad
\frac{B; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{when } a = E :_{c_0} T \in B \wedge c_0 \subseteq c'}{B; \Gamma \vdash_{c' \cup \{a\}} \mathsf{ok}}\,\textbf{(envok.c.a)}$$

$$\frac{B; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{when } x :_{c_0} T \in \Gamma}{B; \Gamma \vdash_{c' \cup \{x\}} \mathsf{ok}}\,\textbf{(envok.c.x)}
\qquad
\frac{}{\mathsf{nil}; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\,\textbf{(envok.nil)}
\qquad
\frac{B; \Gamma \vdash_{c} T : * \qquad \text{when } x \notin \mathbf{dom}\, \Gamma}{B; \Gamma, x :_c T \vdash_{\bullet} \mathsf{ok}}\,\textbf{(envok.x)}$$

$$\frac{B; \Gamma \vdash_{c} \mathsf{ok} \qquad B; \Gamma_0 \vdash_{c_0} \xi\, \mathsf{transparent} \qquad \text{when } \Gamma = (\Gamma_0, x :_{c_0} T, \Gamma_1) \wedge x \in c}{B; \Gamma \vdash_{c} \xi\, \mathsf{transparent}}\,\textbf{(vis.env)}
\qquad
\frac{B; \Gamma \vdash_{c} \mathsf{ok} \qquad \text{when } \xi \in c}{B; \Gamma \vdash_{c} \xi\, \mathsf{transparent}}\,\textbf{(vis.in)}
\qquad
\frac{B; \Gamma \vdash_{c} \mathsf{ok}}{B; \Gamma \vdash_{c} \bullet\ \mathsf{transparent}}\,\textbf{(vis.o)}$$

$$\frac{B; \Gamma \vdash_{c} c_1\, \mathsf{transparent} \qquad B; \Gamma \vdash_{c} c_2\, \mathsf{transparent}}{B; \Gamma \vdash_{c} c_1 \cup c_2\, \mathsf{transparent}}\,\textbf{(vis.union)}$$

$$\frac{B; \Gamma \vdash_{c} c_0\, \mathsf{transparent} \qquad \text{when } a = E :_{c_0} T \in B}{B; \Gamma \vdash_{c} a \triangleright E : T}\,\textbf{(ac.a)}
\qquad
\frac{B; \Gamma \vdash_{c} A \triangleright E : \Pi x : T_0.\, ^{P}T_1 \qquad B; \Gamma \vdash_{c} E_0 :^{P} T_0}{B; \Gamma \vdash_{c} A\ E_0 \triangleright E\ E_0 : \{x{\leftarrow_c}E_0\}T_1}\,\textbf{(ac.app)}$$

$$\frac{B; \Gamma \vdash_{c} A \triangleright E : \Sigma x : T_1.\, T_2}{B; \Gamma \vdash_{c} \pi_1 A \triangleright \pi_1 E : T_1}\,\textbf{(ac.proj.1)}
\qquad
\frac{B; \Gamma \vdash_{c} E_1 :^{P} S(\pi_1 E) \qquad B; \Gamma \vdash_{c} A \triangleright E : \Sigma x : T_1.\, T_2}{B; \Gamma \vdash_{c} \pi_2 A \triangleright \pi_2 E : \{x{\leftarrow_c}E_1\}T_2}\,\textbf{(ac.proj.2)}$$

$$\frac{B; \Gamma \vdash_{c} A \triangleright E : \mathrm{TYPE}^{K}}{B; \Gamma \vdash_{c} (\!|A|\!) : K}\,\textbf{(tok.abs)}
\qquad
\frac{B; \Gamma \vdash_{c} \mathsf{ok}}{B; \Gamma \vdash_{c} \mathrm{BOOL} : \wr}\,\textbf{(tok.base.bool)}
\qquad
\frac{B; \Gamma \vdash_{c} \mathsf{ok}}{B; \Gamma \vdash_{c} \mathrm{DYN} : K}\,\textbf{(tok.base.dyn)}$$

$$\frac{B; \Gamma \vdash_{c} \mathsf{ok}}{B; \Gamma \vdash_{c} \mathrm{INT} : \wr}\,\textbf{(tok.base.int)}
\qquad
\frac{B; \Gamma \vdash_{c} \mathsf{ok}}{B; \Gamma \vdash_{c} \mathrm{UNIT} : \wr}\,\textbf{(tok.base.unit)}
\qquad
\frac{B; \Gamma \vdash_{c} E :^{P} \mathrm{TYPE}^{K}}{B; \Gamma \vdash_{c} \mathsf{Typ}\, E : K}\,\textbf{(tok.field)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T' : K' \\ B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''\end{array}}{B;\Gamma \vdash_c \Pi x : T'.\, {}^{I}T'' : \wr}\,\textbf{(tok.fun.I)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c T' : K' \\ B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''\end{array}}{B;\Gamma \vdash_c \Pi x : T'.\, {}^{P}T'' : K''}\,\textbf{(tok.fun.P)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T' : K' \\ B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''\end{array}}{B;\Gamma \vdash_c \Sigma x : T'.\, T'' : K' \vee K''}\,\textbf{(tok.pair)} \qquad \frac{B;\Gamma \vdash_c \ \mathsf{ok}}{B;\Gamma \vdash_c \textsc{type}^{K} : *}\,\textbf{(tok.type)} \qquad \frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c \mathsf{S}(E) : \wr}\,\textbf{(tok.sing)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T : K' \\ \text{when } K' \leqslant K\end{array}}{B;\Gamma \vdash_c T : K}\,\textbf{(tok.sub)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E_0 \longrightarrow E_0' \\ B;\Gamma \vdash_c E_0 :^{P} T_0 \\ B;\Gamma \vdash_c A \rhd E : \Pi x : T_0.\, {}^{P}T_1\end{array}}{B;\Gamma \vdash_c A\, E_0 \longrightarrow A\, E_0'}\,\textbf{(aconv.cong.app.arg)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c A \longrightarrow A' \\ B;\Gamma \vdash_c A \rhd E : \Pi x : T_0.\, {}^{P}T_1 \\ B;\Gamma \vdash_c E_0 :^{P} T_0\end{array}}{B;\Gamma \vdash_c A\, E_0 \longrightarrow A'\, E_0}\,\textbf{(aconv.cong.app.fun)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c A \longrightarrow A' \\ B;\Gamma \vdash_c A \rhd E : \Sigma x : T_1.\, T_2\end{array}}{B;\Gamma \vdash_c \pi_i A \longrightarrow \pi_i A}\,\textbf{(aconv.cong.proj)}$$

$$\frac{B;\Gamma \vdash_c A_1 \longrightarrow A_2}{B;\Gamma \vdash_c A_1 \equiv A_2}\,\textbf{(aeq.conv)} \quad \frac{B;\Gamma \vdash_c A \rhd E : T}{B;\Gamma \vdash_c A \equiv A}\,\textbf{(aeq.refl)} \quad \frac{B;\Gamma \vdash_c A_2 \equiv A_1}{B;\Gamma \vdash_c A_1 \equiv A_2}\,\textbf{(aeq.sym)} \quad \frac{\begin{array}{c}B;\Gamma \vdash_c A_1 \equiv A_2 \\ B;\Gamma \vdash_c A_2 \equiv A_3\end{array}}{B;\Gamma \vdash_c A_1 \equiv A_3}\,\textbf{(aeq.trans)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c A \longrightarrow A' \\ B;\Gamma \vdash_c A \rhd E : \textsc{type}^{K}\end{array}}{B;\Gamma \vdash_c (\!|A|\!) \longrightarrow (\!|A'|\!)}\,\textbf{(tconv.cong.abs)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E :^{P} \textsc{type}^{*}\end{array}}{B;\Gamma \vdash_c \mathsf{Typ}\, E \longrightarrow \mathsf{Typ}\, E'}\,\textbf{(tconv.cong.field)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T_0 \longrightarrow T_0' \\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 : *\end{array}}{B;\Gamma \vdash_c \Pi x : T_0.\, {}^{\gamma}T_1 \longrightarrow \Pi x : T_0'.\, {}^{\gamma}T_1}\,\textbf{(tconv.cong.fun.arg)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T_0 : * \\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 \longrightarrow T_1'\end{array}}{B;\Gamma \vdash_c \Pi x : T_0.\, {}^{\gamma}T_1 \longrightarrow \Pi x : T_0.\, {}^{\gamma}T_1'}\,\textbf{(tconv.cong.fun.ret)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T_1 \longrightarrow T_1' \\ B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 : *\end{array}}{B;\Gamma \vdash_c \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1'.\, T_2}\,\textbf{(tconv.cong.pair.1)}$$

$$\frac{\begin{array}{c}B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 \longrightarrow T_2' \\ B;\Gamma \vdash_c T_1 : *\end{array}}{B;\Gamma \vdash_c \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1.\, T_2'}\,\textbf{(tconv.cong.pair.2)} \qquad \frac{B;\Gamma \vdash_c E \longrightarrow E'}{B;\Gamma \vdash_c \mathsf{S}(E) \longrightarrow \mathsf{S}(E')}\,\textbf{(tconv.cong.sing)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c A \rhd E : \textsc{type}^{K} \\ B;\Gamma \vdash_c \mathbf{underl}(A)\ \mathsf{transparent}\end{array}}{B;\Gamma \vdash_c (\!|A|\!) \longrightarrow \mathsf{Typ}\, E}\,\textbf{(tconv.abs)} \qquad \frac{B;\Gamma \vdash_c T : *}{B;\Gamma \vdash_c \mathsf{Typ}\, \langle T\rangle \longrightarrow T}\,\textbf{(tconv.field)}$$

$$\frac{B;\Gamma \vdash_c \ \mathsf{ok}}{B;\Gamma \vdash_c \mathsf{S}(()) \longrightarrow \textsc{unit}}\,\textbf{(tconv.unit)}$$

$$\frac{B;\Gamma \vdash_c T_1 \longrightarrow T_2}{B;\Gamma \vdash_c T_1 \equiv T_2}\,(\textbf{teq.conv}) \qquad \frac{B;\Gamma \vdash_c T : *}{B;\Gamma \vdash_c T \equiv T}\,(\textbf{teq.refl}) \qquad \frac{B;\Gamma \vdash_c T_2 \equiv T_1}{B;\Gamma \vdash_c T_1 \equiv T_2}\,(\textbf{teq.sym}) \qquad \frac{\begin{array}{c} B;\Gamma \vdash_c T_1 \equiv T_2 \\ B;\Gamma \vdash_c T_2 \equiv T_3 \end{array}}{B;\Gamma \vdash_c T_1 \equiv T_3}\,(\textbf{teq.trans})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E :^{P} T_0 \\ B;\Gamma \vdash_c E_1 :^{P} \Pi x : T_0.\ ^{P}T_1 \end{array}}{B;\Gamma \vdash_c E_1\, E \longrightarrow E_1\, E'}\,(\textbf{econv.cong.app.arg}) \qquad \frac{\begin{array}{c} B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E :^{P} \Pi x : T_0.\ ^{P}T_1 \\ B;\Gamma \vdash_c E_0 :^{P} T_0 \end{array}}{B;\Gamma \vdash_c E\, E_0 \longrightarrow E'\, E_0}\,(\textbf{econv.cong.app.fun})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} E \longrightarrow E' \\ B;\Gamma \vdash_{c'} E :^{P} T \\ B;\Gamma \vdash_{c\cap c'} T : \wr \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c [E]^{T}_{c'} \longrightarrow [E']^{T}_{c'}}\,(\textbf{econv.cong.col.e}) \qquad \frac{\begin{array}{c} B;\Gamma \vdash_{c'} E :^{P} T_1 \\ B;\Gamma \vdash_{c\cap c'} T_1 \longrightarrow T_2 \\ B;\Gamma \vdash_{c\cap c'} T_1 : \wr \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c [E]^{T_1}_{c'} \longrightarrow [E]^{T_2}_{c'}}\,(\textbf{econv.cong.col.t})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{\bullet} E \longrightarrow E' \\ B;\Gamma \vdash_{\bullet} T : \wr \\ B;\Gamma \vdash_{\bullet} E :^{P} T \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c \text{dynned}\, E \text{ at } T \longrightarrow \text{dynned}\, E' \text{ at } T}\,(\textbf{econv.cong.dynned.e})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{\bullet} T \longrightarrow T' \\ B;\Gamma \vdash_{\bullet} T : \wr \\ B;\Gamma \vdash_{\bullet} E :^{P} T \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c \text{dynned}\, E \text{ at } T \longrightarrow \text{dynned}\, E \text{ at } T'}\,(\textbf{econv.cong.dynned.t}) \qquad \frac{B;\Gamma \vdash_c T \longrightarrow T'}{B;\Gamma \vdash_c \langle T \rangle \longrightarrow \langle T' \rangle}\,(\textbf{econv.cong.field})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_c T_0 \longrightarrow T_0' \\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{\gamma} T_1 \end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1)}\,(\textbf{econv.cong.fun.arg})$$

$$\frac{\begin{array}{c} B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E \longrightarrow E' \\ B;\Gamma, x :_c T_0, y :_{c\cup\{x\}} S(E) \vdash_{c\cup\{y\}\cup\{x\}} E_1 :^{\gamma} T_1 \end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ \{y \leftarrow_{c\cup\{x\}} E\}E_1) \longrightarrow (\lambda x : T_0.\ \{y \leftarrow_{c\cup\{x\}} E'\}E_1)}\,(\textbf{econv.cong.fun.body})$$

$$\frac{\begin{array}{c} B;\Gamma, x :_c T_0 \vdash_{c'} E_1 :^{\gamma} T_1 \\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 \longrightarrow T_1' \\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1) \longrightarrow (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1')}\,(\textbf{econv.cong.fun.seal})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E_2 :^{P} T_2 \end{array}}{B;\Gamma \vdash_c (E, E_2) \longrightarrow (E', E_2)}\,(\textbf{econv.cong.pair.1}) \qquad \frac{\begin{array}{c} B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E_1 :^{P} T_1 \end{array}}{B;\Gamma \vdash_c (E_1, E) \longrightarrow (E_1, E')}\,(\textbf{econv.cong.pair.2})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_c E \longrightarrow E' \\ B;\Gamma \vdash_c E :^{P} \Sigma x : T_1.\ T_2 \end{array}}{B;\Gamma \vdash_c \pi_i\, E \longrightarrow \pi_i\, E'}\,(\textbf{econv.cong.proj}) \qquad \frac{\begin{array}{c} B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{P} T_1 \\ B;\Gamma \vdash_c E_0 :^{P} T_0 \end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow_c E_0\}E_1}\,(\textbf{econv.app})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} \text{ ok} \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c [bv]^{\mathrm{BOOL}}_{c'} \longrightarrow bv}\,(\textbf{econv.col.base.bool}) \qquad \frac{\begin{array}{c} B;\Gamma \vdash_{c'} \text{ ok} \\ B;\Gamma \vdash_c \text{ ok} \end{array}}{B;\Gamma \vdash_c [\underline{n}]^{\mathrm{INT}}_{c'} \longrightarrow \underline{n}}\,(\textbf{econv.col.base.int})$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c'} \mathsf{ok}\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [()]_{c'}^{\mathrm{UNIT}} \longrightarrow ()}\;\textbf{(econv.col.base.unit)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{\bullet} E :^{P} T\\ B;\Gamma \vdash_{\bullet} T : \wr\\ B;\Gamma \vdash_{c'} \mathsf{ok}\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [\mathsf{dynned}\,E\,\mathsf{at}\,T]_{c'}^{\mathrm{DYN}} \longrightarrow \mathsf{dynned}\,E\,\mathsf{at}\,T}\;\textbf{(econv.col.dynned)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c'} T_0 <: T_2\\ B;\Gamma, x :_{c'} T_2 \vdash_{c'\cup\{x\}} E :^{I} T_1\\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\; E]_{c'}^{\Pi x:T_0.\,{}^{I}T_1} \longrightarrow \lambda x : T_0.\; E\,!!_{c'\cup\{x\}}\,T_1}\;\textbf{(econv.col.fun.I)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c'} T_0 <: T_2\\ B;\Gamma, x :_{c'} T_2 \vdash_{c'\cup\{x\}} E :^{P} T_1\\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\; E]_{c'}^{\Pi x:T_0.\,{}^{P}T_1} \longrightarrow \lambda x : T_0.\; [E]_{c'\cup\{x\}}^{T_1}}\;\textbf{(econv.col.fun.P)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c_2} E :^{P} T_2\\ B;\Gamma \vdash_{c_1\cap c_2} T_2 : \wr\\ B;\Gamma \vdash_{c_1} T_2 <: T_1\\ B;\Gamma \vdash_{c\cap c_1} T_1 : \wr\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [[E]_{c_2}^{T_2}]_{c_1}^{T_1} \longrightarrow [E]_{c_1\cup c_2}^{T_1}}\;\textbf{(econv.col.merge)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c'} E_1 :^{P} T_1\\ B;\Gamma \vdash_{c\cap c'} T_1 : \wr\\ B;\Gamma, x :_{c'} T_1 \vdash_{c'\cup\{x\}} E_2 :^{P} T_2\\ B;\Gamma \vdash_{c'} E_2 :^{P} \{x\leftarrow_{c'}[E_1]_{c'}^{T_1}\}T_2\\ B;\Gamma, x :_{c\cap c'} T_1 \vdash_{(c\cap c')\cup\{x\}} T_2 : \wr\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [(E_1, E_2)]_{c'}^{\Sigma x:T_1.\,T_2} \longrightarrow ([E_1]_{c'}^{T_1}, [E_2]_{c'}^{\{x\leftarrow_{c'}[E_1]_{c'}^{T_1}\}T_2})}\;\textbf{(econv.col.pair)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c'} E' :^{P} S(E)\\ B;\Gamma \vdash_{c\cap c'} E :^{P} T\\ B;\Gamma \vdash_{c} \mathsf{ok}\end{array}}{B;\Gamma \vdash_{c} [E']_{c'}^{S(E)} \longrightarrow E}\;\textbf{(econv.col.sing)}$$

$$\dfrac{\begin{array}{c}B;\Gamma \vdash_{c} E_1 :^{P} T_1\\ B;\Gamma \vdash_{c} E_2 :^{P} T_2\end{array}}{B;\Gamma \vdash_{c} \pi_i(E_1, E_2) \longrightarrow E_i}\;\textbf{(econv.proj)}$$

$$\dfrac{B;\Gamma \vdash_{c} E :^{P} \mathrm{TYPE}^{*}}{B;\Gamma \vdash_{c} E \longrightarrow \langle \mathsf{Typ}\,E\rangle}\;\textbf{(econv.eta.field)}$$

$$\dfrac{B;\Gamma \vdash_{c} E :^{P} \Pi x : T_0.\,{}^{\gamma}T_1}{B;\Gamma \vdash_{c} E \longrightarrow (\lambda x : T_0.\; E\,x)}\;\textbf{(econv.eta.fun)}$$

$$\dfrac{B;\Gamma \vdash_{c} E :^{P} \Sigma x : T_1.\,T_2}{B;\Gamma \vdash_{c} E \longrightarrow (\pi_1\,E, \pi_2\,E)}\;\textbf{(econv.eta.pair)}$$

$$\frac{B;\Gamma \vdash_c E_1 \longrightarrow E_2}{B;\Gamma \vdash_c E_1 \equiv E_2}\,\textsf{(eeq.conv)} \qquad \frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c E \equiv E}\,\textsf{(eeq.refl)} \qquad \frac{B;\Gamma \vdash_c E_2 \equiv E_1}{B;\Gamma \vdash_c E_1 \equiv E_2}\,\textsf{(eeq.sym)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E_1 \equiv E_2\\ B;\Gamma \vdash_c E_2 \equiv E_3\end{array}}{B;\Gamma \vdash_c E_1 \equiv E_3}\,\textsf{(eeq.trans)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T_0' <: T_0\\ B;\Gamma, x :_c T_0' \vdash_{c\cup\{x\}} T_1 <: T_1'\\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 : *\\ \text{when } \gamma \sqsubseteq \gamma'\end{array}}{B;\Gamma \vdash_c \Pi x : T_0.\ {}^{\gamma}T_1 <: \Pi x : T_0'.\ {}^{\gamma'}T_1'}\,\textsf{(tsub.cong.fun)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c T_1 <: T_1'\\ B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 <: T_2'\\ B;\Gamma, x :_c T_1' \vdash_{c\cup\{x\}} T_2' : *\end{array}}{B;\Gamma \vdash_c \Sigma x : T_1.\ T_2 <: \Sigma x : T_1'.\ T_2'}\,\textsf{(tsub.cong.pair)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c \mathsf{ok}\\ \text{when } K_1 \leqslant K_2\end{array}}{B;\Gamma \vdash_c \mathrm{TYPE}^{K_1} <: \mathrm{TYPE}^{K_2}}\,\textsf{(tsub.cong.type)} \qquad \frac{B;\Gamma \vdash_c T \equiv T'}{B;\Gamma \vdash_c T <: T'}\,\textsf{(tsub.eq)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c T <: T'\\ B;\Gamma \vdash_c T' <: T''\end{array}}{B;\Gamma \vdash_c T <: T''}\,\textsf{(tsub.trans)}$$

$$\frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c S(E) <: T}\,\textsf{(tsub.sing)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E_1 :^{\gamma_1} \Pi x : T_0.\ {}^{\gamma_2}T\\ B;\Gamma \vdash_c E_0 :^{P} T_0\end{array}}{B;\Gamma \vdash_c E_1\, E_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow_c E_0\}T}\,\textsf{(et.app)} \qquad \frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c bv :^{P} \mathrm{BOOL}}\,\textsf{(et.base.bool)} \qquad \frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \underline{n} :^{P} \mathrm{INT}}\,\textsf{(et.base.int)}$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c () :^{P} \mathrm{UNIT}}\,\textsf{(et.base.unit)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_{c'} E :^{\gamma} T\\ B;\Gamma \vdash_{c\cap c'} T : \wr\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c [E]^{T}_{c'} :^{\gamma} T}\,\textsf{(et.col)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E :^{\gamma} T\\ B;\Gamma \vdash_c T : \wr\end{array}}{B;\Gamma \vdash_c \mathsf{dyn}\, E\, \mathsf{at}\, T :^{I} \mathrm{DYN}}\,\textsf{(et.dyn)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_{\bullet} E :^{P} T\\ B;\Gamma \vdash_{\bullet} T : \wr\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c \mathsf{dynned}\, E\, \mathsf{at}\, T :^{P} \mathrm{DYN}}\,\textsf{(et.dynned)} \qquad \frac{B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E :^{\gamma} T_1}{B;\Gamma \vdash_c \lambda x : T_0.\ E :^{P} \Pi x : T_0.\ {}^{\gamma}T_1}\,\textsf{(et.fun)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E_0 :^{I} T_0\\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E :^{I} T\\ B;\Gamma \vdash_c T : *\end{array}}{B;\Gamma \vdash_c (\mathsf{let}\, x = E_0\, \mathsf{in}\, E : T) :^{I} T}\,\textsf{(et.let)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E_1 :^{\gamma} T_1\\ B;\Gamma \vdash_c E_2 :^{\gamma} T_2\end{array}}{B;\Gamma \vdash_c (E_1, E_2) :^{\gamma} T_1 * T_2}\,\textsf{(et.pair)}$$

$$\frac{B;\Gamma \vdash_c E :^{\gamma} \Sigma x : T_1.\ T_2}{B;\Gamma \vdash_c \pi_1 E :^{\gamma} T_1}\,\textsf{(et.proj.1)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E :^{P} \Sigma x : T_1.\ T_2\\ B;\Gamma \vdash_c E_1 :^{P} S(\pi_1 E)\end{array}}{B;\Gamma \vdash_c \pi_2 E :^{P} \{x \leftarrow_c E_1\}T_2}\,\textsf{(et.proj.2)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_{c\cup c'} E :^{\gamma} T\\ B;\Gamma \vdash_c T : *\end{array}}{B;\Gamma \vdash_c (E\ !!_{c'}\ T) :^{I} T}\,\textsf{(et.seal)}$$

$$\frac{B;\Gamma \vdash_c T : K}{B;\Gamma \vdash_c \langle T \rangle :^{P} \mathrm{TYPE}^{K}}\,\textsf{(et.type)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E :^{\gamma} \mathrm{DYN}\\ B;\Gamma \vdash_c E' :^{\gamma} T\end{array}}{B;\Gamma \vdash_c \mathsf{undyn}\, E\, \mathsf{at}\, T\, \mathsf{else}\, E' :^{I} T}\,\textsf{(et.undyn)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c x\, \mathsf{transparent}\\ \text{when } x :_{c'} T \in \Gamma\end{array}}{B;\Gamma \vdash_c x :^{P} T}\,\textsf{(et.x)}$$

$$\frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c E :^{P} S(E)}\,\textsf{(et.sing)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E :^{\gamma} T\\ B;\Gamma \vdash_c T <: T'\\ \text{when } \gamma \sqsubseteq \gamma'\end{array}}{B;\Gamma \vdash_c E :^{\gamma'} T'}\,\textsf{(et.sub)}$$

$$(\lambda x : \mathsf{T}.\ \mathsf{E})\ \mathsf{V}^{c} \longrightarrow_{c} \{x \leftarrow_{c} \mathsf{V}^{c}\}\mathsf{E} \qquad \text{(ered.app)}$$

$$[bv]_{c'}^{\mathrm{BOOL}} \longrightarrow_{c} bv \qquad \text{(ered.col.base.bool)}$$

$$[\underline{n}]_{c'}^{\mathrm{INT}} \longrightarrow_{c} \underline{n} \qquad \text{(ered.col.base.int)}$$

$$[()]_{c'}^{\mathrm{UNIT}} \longrightarrow_{c} () \qquad \text{(ered.col.base.unit)}$$

$$[\mathsf{dynned}\ \mathsf{V}^{\bullet}\ \mathsf{at}\ \mathsf{T}]_{c'}^{\mathrm{DYN}} \longrightarrow_{c} \mathsf{dynned}\ \mathsf{V}^{\bullet}\ \mathsf{at}\ \mathsf{T} \qquad \text{(ered.col.dynned)}$$

$$[\lambda x : \mathsf{T}_2.\ \mathsf{E}]_{c'}^{\Pi x:\mathsf{T}_0.\,{}^{\mathrm{I}}\mathsf{T}_1} \longrightarrow_{c} \lambda x : \mathsf{T}_0.\ (\mathsf{E}\ !!_{c' \cup \{x\}}\ \mathsf{T}_1) \qquad \text{(ered.col.fun.I)}$$

$$[\lambda x : \mathsf{T}_2.\ \mathsf{E}]_{c'}^{\Pi x:\mathsf{T}_0.\,{}^{\mathrm{P}}\mathsf{T}_1} \longrightarrow_{c} \lambda x : \mathsf{T}_0.\ [\mathsf{E}]_{c' \cup \{x\}}^{\mathsf{T}_1} \qquad \text{(ered.col.fun.P)}$$

$$[[\mathsf{V}^{c_2}]_{c_2}^{(\!|A_2|\!)}]_{c_1}^{(\!|A_1|\!)} \longrightarrow_{c} [\mathsf{V}^{c_2}]_{c_1 \cup c_2}^{(\!|A_1|\!)} \qquad \text{(ered.col.merge)}$$

if A_1 et A_2 are both opaque in c_1 but A_2 is concrete in c_2

$$[(\mathsf{V}_1^{c'}, \mathsf{V}_2^{c'})]_{c'}^{\Sigma x:\mathsf{T}_1.\,\mathsf{T}_2} \longrightarrow_{c} ([\mathsf{V}_1^{c'}]_{c'}^{\mathsf{T}_1}, [\mathsf{V}_2^{c'}]_{c'}^{\{x \leftarrow_{c} [\mathsf{V}_1^{c'}]_{c'}^{\mathsf{T}_1}\}\mathsf{T}_2}) \qquad \text{(ered.col.pair)}$$

$$[\mathsf{V}^{c'}]_{c'}^{\mathsf{S}(\mathsf{E})} \longrightarrow_{c} \mathsf{E} \qquad \text{(ered.col.sing)}$$

$$\mathsf{B} \vdash [\mathsf{V}^{c'}]_{c'}^{(\!|A|\!)} \longrightarrow_{c} \mathsf{B} \vdash [\mathsf{V}^{c'}]_{c'}^{\mathsf{Typ}\ \mathbf{reveal}^{\mathsf{B}}(A)} \qquad \text{(ered.colAbs)}$$

if $\mathbf{underl}(A) \in c \cap c'$

$$[\mathsf{V}^{c'}]_{c'}^{\mathsf{Typ}\ \langle \mathsf{T} \rangle} \longrightarrow_{c} [\mathsf{V}^{c'}]_{c'}^{\mathsf{T}} \qquad \text{(ered.colTyp)}$$

$$\frac{\mathsf{E} \longrightarrow_{c'} \mathsf{E}'}{\mathsf{C}_{c'}^{c} \cdot \mathsf{E} \longrightarrow_{c} \mathsf{C}_{c'}^{c} \cdot \mathsf{E}'}\ \textbf{(ered.context)}$$

$$\mathsf{dyn}\ \mathsf{V}^{c}\ \mathsf{at}\ \mathsf{T} \longrightarrow_{c} \mathsf{dynned}\ [\mathsf{V}^{c}]_{c}^{\mathbf{conc}_{c}^{\mathsf{B}}(\mathsf{T})}\ \mathsf{at}\ \mathbf{conc}_{c}^{\mathsf{B}}(\mathsf{T}) \qquad \text{(ered.dyn)}$$

$$\mathsf{let}\ x = \mathsf{V}^{c}\ \mathsf{in}\ \mathsf{E} : \mathsf{T} \longrightarrow_{c} \{x \leftarrow_{c} \mathsf{V}^{c}\}\mathsf{E} \qquad \text{(ered.let)}$$

$$\pi_i\,(\mathsf{V}_1^{c}, \mathsf{V}_2^{c}) \longrightarrow_{c} \mathsf{V}_i \qquad \text{(ered.proj)}$$

$$\mathsf{B} \vdash \mathsf{V}^{c \cup c'}\ !!_{c'}\ \mathsf{T} \longrightarrow_{c} \mathsf{B}, a = \mathsf{V}^{c \cup c'} :_{c \cup c'} \mathsf{T} \vdash [\mathsf{V}^{c \cup c'}]_{c \cup c' \cup \{a\}}^{\mathbf{self}^{\,\mathsf{T}}(a)} \qquad \text{(ered.seal)}$$

where a is fresh (i.e., $a \notin \mathbf{dom}\,\mathsf{B}$)

$$\mathsf{B} \vdash \mathsf{undyn}\,(\mathsf{dyn}\ \mathsf{V}^{c}\ \mathsf{at}\ \mathsf{T})\ \mathsf{at}\ \mathsf{T}'\ \mathsf{else}\ \mathsf{E}' \longrightarrow_{c} \mathsf{B} \vdash \begin{cases} \mathsf{V}^{c} & \text{if } \mathsf{B}; \mathsf{nil} \vdash_{c} \mathsf{T} <: \mathsf{T}' \\ \mathsf{E}' & \text{otherwise} \end{cases} \qquad \text{(ered.undyn)}$$

Bibliography

[AM91] Andrew Appel and David B. MacQueen. Standard ML of new jersey. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings of the 3rd Intn'l Symposium*, volume 528 of *LNCS*, pages 1–13. Springer Verlag, 1991.

[CM88] L. Cardelli and D. MacQueen. Persistence and type abstraction. In Malcolm P. Atkinson, Peter Buneman, and Ronald Morrison, editors, *Data Types and Persistence. Edited Papers from the Proceedings of the First Workshop on Persistent Objects, Appin, Scotland, August 1985*, Topics in Information Systems, pages 31–41. Springer, 1988. Since revised.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–249, New York, NY, USA, 2003. ACM Press.

[Dre02] Derek Dreyer. Moscow ML's higher-order modules are unsound, December 2002. Message on the TYPES forum. Online at `http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg01136.html`.

[Dre05] Derek Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, 2005.

[FLFS07] Cédric Fournet, Fabrice Le Fessant, and Alan Schmitt. *The JoCaml language (beta release)*, 2007.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *CONCUR '97: Proceedings of the 1997 8th International Conference on Concurrency Theory*, pages 196–212. Springer-Verlag, July 1997.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *FLOPS '04: Proceedings of the 7th International Symposium on Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer-Verlag, April 2004.

[Gog05] Healfdene Goguen. A syntactic approach to eta equality in type theory. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–84, 2005.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM Press.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354, New York, NY, USA, 1990. ACM Press.

[Klo80] Jan Willem Klop. *Combinatory reduction systems.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[L$^+$] Xavier Leroy et al. *The Objective Caml system.*

[Ler] Xavier Leroy. Private communication.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, New York, NY, USA, 1994. ACM Press.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 142–153, New York, NY, USA, 1995. ACM Press.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon University, May 1997.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 87–98, New York, NY, USA, 2003. ACM Press.

[Mac84] David MacQueen. Modules for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM Press.

[MM01] Louis Mandel and Luc Maranget. *The JoCaml language*, 2001.

[OTCP90] Atsushi Ohori, Ivan Tabkha, Richard Connor, and Paul Philbrow. Persistence and type abstraction revisited. In *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects, 23-27 September 1990, Martha's Vineyard, MA, USA*, pages 141–153. Morgan Kaufmann, 1990.

[Pie05] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* MIT Press, 2005.

[PS00] Benjamin C. Pierce and Eijiro Sumii. Relating cryptography and polymorphism. Manuscript, July 2000.

[PSL] Programming System Lab, Saarland University. *The Alice ML Language.*

[Ros] Andreas Rossberg. SML vs. Ocaml. Online at `http://www.ps.uni-sb.de/~rossberg/SMLvsOcaml.html`.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 241–252, New York, NY, USA, 2003. ACM Press.

[Ros07] Andreas Rossberg. *Typed Open Programming — A higher-order, typed approach to dynamic modularity and distribution.* PhD thesis, Universität des Saarlandes, January 2007.

[RRS] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Language Overview.*

[Rus98] Claudio Russo. *Types For Modules.* PhD thesis, University of Edinburgh, 1998.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–247, New York, NY, USA, 2001. ACM Press.

[Sha99] Zhong Shao. Transparent modules with fully syntatic signatures. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 220–232, New York, NY, USA, 1999. ACM Press.

[SP04] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–172, New York, NY, USA, 2004. ACM Press.

[Sun] Sun Microsystems, Inc. *Java API Specifications.*

[Wad89] Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM Press.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: a syntactic proof technique. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 197–207, New York, NY, USA, 1999. ACM Press.

Index