Modules, Abstraction, and Parametric Polymorphism

Karl Crary

Carnegie Mellon University

Abstract

Reynolds's Abstraction theorem forms the mathematical foundation for data abstraction. His setting was the polymorphic lambda calculus. Today, many modern languages, such as the ML family, employ rich module systems designed to give more expressive support for data abstraction than the polymorphic lambda calculus, but analogues of the Abstraction theorem for such module systems have lagged far behind.

We give an account of the Abstraction theorem for a modern module calculus supporting generative and applicative functors, higher-order functors, sealing, and translucent signatures. The main issues to be overcome are: (1) the fact that modules combine both types and terms, so they must be treated as both simultaneously, (2) the effect discipline that models the distinction between transparent and opaque modules, and (3) a very rich language of type constructors supporting singleton kinds. We define logical equivalence for modules and show that it coincides with contextual equivalence. This substantiates the folk theorem that modules are good for data abstraction. All our proofs are formalized in Coq.

1 Introduction

In his seminal 1983 paper *Types, Abstraction and Parametric Polymorphism* [25], John Reynolds first stated his Abstraction theorem. The theorem (which also came to be known as the Parametricity theorem) has been put to various purposes—including free theorems [30], and full abstraction for program transformations [1]—but its principal application remains Reynolds's original one: the theoretical foundation for type abstraction. The theorem states that no well-typed program can distinguish between two different implementations of an abstraction, provided the two implementations operate the same way when viewed through the abstraction's interface. The theorem gives programmers license to reimplement components of a larger system without worrying about compromising the system.

Reynolds stated his theorem for the polymorphic lambda calculus, in which the only mechanism for type abstraction arises from parametric polymorphism, that is, functions that take types as arguments. The theorem then says that two different applications of a function are equivalent provided that the arguments are (to use modern terminology) logically related.

This gives an account of *client-side* abstraction: A client of a component, when he or she desires to hold that component abstract, can do so by passing it into a polymorphic function. But note that the burden of abstraction falls on the client of a component. A more robust form of type abstraction is *provider-side,* in which the provider of a component may "seal" it, rendering it abstract to all clients, without requiring any action on the clients' part. It is only provider-side abstraction that gives the provider *carte blanche* to reimplement a component at will.

Later type systems did provide support for provider-side abstraction. First, existential types [20] provided rudimentary support for packages that are made abstract by the provider. They also support a theoretical account of data abstraction [19] similar to that of Reynolds. However, they are clumsy for the client to use. Distinct unpackings of the same package provide incompatible types, so the client must be sure to open a package with a large enough scope to encompass all its uses.

Modular programming languages provide a more elegant account of provider-side abstraction [15, 31, 2]. In them, a *module* groups together abstract type definitions and operations on those types—that is, an entire abstract data type implementation—as a single component, on which an abstract interface is imposed.

A particularly successful form of modular programming is provided by the module language [8] of Standard ML and its cousins, based on the strong sum mechanism of Martin-Löf type theory [17]. In the ML-style module calculi, a module's abstract types can be referenced without opening the module. This facility makes it possible to give a closed type to a module's operations, in contrast to existential types.

The family of ML-style module calculi has enjoyed sustained development at the hands of many researchers and in many different dimensions, mostly directed at providing better facilities for abstraction for various programming-in-the-large settings. Of particular interest here is work on functors [8, 9, 7, 12, 16, 13, 28, 5, 4]. Functors allow the programmer to give an implementation of an abstract datatype that depends parametrically on an unknown implementation of another abstract data type. The parameter is filled in later, and possibly more than once. In ML and its cousins, a functor is seen as a form of function, mapping modules to modules. The most advanced functor calculi even support *higher-order functors*, in which a functor can take another functor as its argument.

Throughout the three-decade history of ML-style module calculi, it has been assumed that sophisticated module calculi are good for providing data abstraction. Informally,

this assumption has been well substantiated, both by illustrative examples, and by large, multi-programmer software projects.

Formally, however, the situation is quite different. In fact, there has never been a published proof of a Reynolds-style abstraction theorem for any ML-style module calculus. The closest is Leroy [13], which posits, but does not carefully prove, an abstraction principle for the applicative-functors module calculus now underlying the OCaml language [14], and discusses some corollaries.

In this paper, we show that a Reynolds-style abstraction theorem does indeed hold for a sophisticated module calculus. All the proofs are formalized in Coq. The calculus we consider supports both applicative and generative functors, higher-order functors, and controlled abstraction (also known as translucency). It also respects the phase distinction [9] between compile- and run-time expressions, meaning that types can be compared for equivalence without executing the modules those types come from.

Of course, this result is ultimately unsurprising. At a high-level, we confirm the folk theorem that ML modules are good for data abstraction. Nevertheless, the details are instructive: it is not obvious even how to state the abstraction theorem for our module calculus of interest, and the proof is non-trivial.

To see why, we outline the standard story of an abstraction theorem, and then look at how modules complicate the story.

The abstraction story, in abstract First one defines *contextual equivalence,* which we write $e \approx e' : \tau$. It says that whenever e and e' are embedded into a hole in a larger program, the resulting programs produce the same observable result. Equivalently, we can define contextual equivalence as the coarsest congruence such that equivalent terms produce the same observable result. Contextual equivalence is the gold standard in operational equivalence because it relates all terms that cannot be effectively distinguished, but it is difficult to work with because showing a contextual equivalence requires quantifying over all possible closing contexts.

Next, one defines *logical equivalence,* which we write $e \Leftrightarrow e' : \tau$. Logical equivalence is the central technical definition. It defines equivalence not in terms of closing contexts, but in terms of the operational behavior of e and e' themselves. It says that e and e' are equivalent when the operations that can be performed on them (according to their common type τ) produce results that are logically equivalent.

With these two definitions in hand, we prove that contextual and logical equivalence coincide. Thus we may show that two terms are contextually equivalent (*i.e.,* indistinguishable) by showing that they are logically equivalent.

The application to data abstraction comes from logical equivalence's treatment of quantifiers. In one of those serendipities that make type theory a joy, it almost falls out from the technical device (Girard's method [6]) needed to define logical equivalence over impredicative polymorphism. Consider the type $\exists\alpha.\tau$ of an abstract data type. In order to show that two implementations $\mathsf{pack}\,[\rho, e]$ and $\mathsf{pack}\,[\rho', e']$ are logically equivalent (and hence indistinguishable), it is not necessary to have $\rho = \rho'$. Rather, we exhibit a relation R between values of ρ and ρ', such that e and e' are logically equivalent, where all appearances of α in τ are interpreted using R. A similar story, with a bit more convolution, can be told for the universal quantifier.

For example, we may show that $\mathsf{pack}\,[\mathsf{bool}, \langle \texttt{true}, \lambda x.x \rangle]$ and $\mathsf{pack}\,[\mathsf{int}, \langle 0, \texttt{isEven}? \rangle]$ are logically equivalent at $\exists\alpha.\alpha \times (\alpha \to \mathsf{bool})$ by choosing R to be the relation that relates true to even numbers. Then `true` and 0 are R-related, and $\lambda x.x$ and `isEven`? take R-related arguments to the same boolean.

With modules In the presence of modules, the definition of logical equivalence becomes more complicated, for several reasons:

- *Mixed-phase objects:* The standard definition treats ordinary functions (which take terms as arguments) one way, while treating polymorphic functions (which take types as arguments) another way. But modules contain both terms and types, so they must be treated both ways at once.
- *Purity:* Our module calculus distinguishes between pure and impure modules. Roughly speaking, these correspond to unsealed and sealed modules, so pure modules are the ones whose type components may be considered well-determined by their clients. This gives rise to two different notions of logical equivalence, a coarser notion for impure modules, and a finer one for pure ones.
- *Expressive type constructors:* To allow programmers to express precise sharing constraints between modules, our module calculus is built on top of the singleton kind calculus [29]. It contains not only higher-order type constructors, but dependent kinds and singleton kinds (that is, kinds containing exactly one type), and it employs a subkinding judgement. Consequently, we need a full story of logical equivalence for type constructors as well as for terms and modules, and the two must cohere with each other.

We develop our theory in four steps: First, we summarize our module calculus and state some preliminary results regarding it (Section 2). Second, we define contextual equivalence (Section 3). Third, we adapt biorthogonality [22] to our module calculus (Section 4), as a technical device to deal with admissibility arising from recursion. Fourth, we define logical equivalence and prove that it coincides with contextual equivalence (Section 5).

All our proofs are formalized in Coq, and are available at:

`www.cs.cmu.edu/~crary/papers/2016/mapp.tgz`

2 The Module Calculus

Our module calculus is adapted from that of Dreyer [4], and is very nearly a fragment of the internal language used by Lee, *et al.* [11]. The module calculus is not a research contribution of this paper; our purpose here is just to lay the foundation for our development of the Abstraction theorem.

The module calculus, by itself, is not intended to account for all the functionality of ML modules, such as named fields, sealed functor arguments, or `open` declarations. We assume that the full language is defined by elaboration into the module calculus, as suggested by recent work on formalizing SML [10, 5, 4, 11] (but in contrast to earlier work [18]).

The module calculus can be broken into the static expressions: kinds (k), type constructors (c), and signatures (σ)— and the dynamic expressions: terms (e) and modules (M).

k	::=	1	unit kind
	|	T	types
	|	$\mathsf{S}(c)$	singleton kind
	|	$\Pi\alpha{:}k.k$	dependent functions
	|	$\Sigma\alpha{:}k.k$	dependent pairs
$c,$	::=	α	
τ	|	$\star$	unit constructor
	|	$\lambda\alpha{:}k.c \mid c\,c$	lambda, application
	|	$\langle c, c\rangle$	pair
	|	$\pi_1 c \mid \pi_2 c$	projection
	|	unit	unit type
	|	$\tau_1 \to \tau_2$	functions
	|	$\tau_1 \times \tau_2$	products
	|	$\forall\alpha{:}k.\tau$	universals
	|	$\exists\alpha{:}k.\tau$	existentials
e	::=	x	
	|	$\star$	unit term
	|	$\lambda x{:}\tau.e \mid e\,e$	lambda, application
	|	$\langle e, e\rangle$	pair
	|	$\pi_1 e \mid \pi_2 e$	projection
	|	$\Lambda\alpha{:}k.e$	polymorphic fun.
	|	$e[c]$	polymorphic app.
	|	$\mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau$	existential package
	|	$\mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,e$	unpack
	|	$\mathsf{fix}_\tau\, e$	recursion
	|	$\mathsf{let}\, x = e\,\mathsf{in}\,e$	term binding
	|	$\mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e$	module binding
	|	$\mathsf{Ext}\,M$	extraction
M	::=	m	
	|	$\star$	unit module
	|	$(\!\vert\, c \,\vert\!)$	atomic module
	|	$\{\!\vert\, e \,\vert\!\}$	atomic module
	|	$\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M$	generative functor
	|	$M\,M$	generative app.
	|	$\lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M$	applicative functor
	|	$M \cdot M$	applicative app.
	|	$\langle M, M\rangle$	pair
	|	$\pi_1 M \mid \pi_2 M$	projection
	|	$\mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(M : \sigma)$	unpack
	|	$\mathsf{let}\, x = e\,\mathsf{in}\,M$	term binding
	|	$\mathsf{let}\,\alpha/m = M\,\mathsf{in}\,(M : \sigma)$	module binding
	|	$M :> \sigma$	sealing
σ	::=	1	unit signature
	|	$(\!\vert\, k \,\vert\!)$	atomic signature
	|	$\{\!\vert\, \tau \,\vert\!\}$	atomic signature
	|	$\Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma$	generative functors
	|	$\Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma$	applicative functors
	|	$\Sigma\alpha{:}\sigma.\sigma$	pairs
Γ	::=	ϵ	empty context
	|	$\Gamma, \alpha{:}k$	constr. hypothesis
	|	$\Gamma, x{:}\tau$	term hypothesis
	|	$\Gamma, \alpha/m{:}\sigma$	module hypothesis

Figure 1: Syntax

Signatures serve as the types for modules. The calculus is designed to respect the *phase distinction* [9], meaning that the meaning and equivalence of static expressions can be determined without executing any dynamic expressions. The full syntax is given in Figure 1.

Constructors and kinds The constructor and kind portion is the singleton kind calculus of Stone and Harper [29], supplemented with type operators. The type constructors are all standard. The kind T contains constructors that happen to be types. We use the metavariable τ for constructors that are intended to be types. The kind 1 contains the unit constructor $\star$.

The *singleton kind* [29], $\mathsf{S}(c)$, classifies the constructors that are definitionally equivalent to the constructor c. It is used to model type definitions and type sharing specifications. Singletons create dependencies of kinds on constructors, so function and product kinds take dependent form, $\Pi\alpha{:}k_1.k_2$ and $\Sigma\alpha{:}k_1.k_2$, respectively. As usual, when α does not appear free in k_2, we write $\Pi\alpha{:}k_1.k_2$ as $k_1 \to k_2$, and $\Sigma\alpha{:}k_1.k_2$ as $k_1 \times k_2$.

Constructor equivalence is induced by beta and extensionality rules, together with rules pertaining to singleton kinds: Singleton introduction says c belongs to $\mathsf{S}(c)$, provided $c : \mathsf{T}$. (Singletons for higher kinds are definable using primitive singletons and dependent kinds [29].) The singleton elimination rule conversely says $c : \mathsf{S}(c')$ implies $c \equiv c' : \mathsf{T}$. Consequently, constructor equivalence is context-sensitive. For example, we have $\alpha{:}\mathsf{S}(\mathsf{int}) \vdash \alpha \equiv \mathsf{int} : \mathsf{T}$. Constructor equivalence also depends on the kind at which constructors are compared. For example, $\lambda\alpha{:}\mathsf{T}.\alpha$ and $\lambda\alpha{:}\mathsf{T}.\mathsf{int}$ are non-equivalent at $\mathsf{T} \to \mathsf{T}$, but are equivalent at the superkind $\mathsf{S}(\mathsf{int}) \to \mathsf{T}$.

Terms and modules The syntax for terms is mostly standard. The recursion form $\mathsf{fix}_\tau\, e$ has the type τ, provided e has the type $(\mathsf{unit} \to \tau) \to \tau$.[1]

The module language contains static atomic modules ($(\!\vert\, c \,\vert\!)$), which contain a single constructor, dynamic atomic modules ($\{\!\vert\, e \,\vert\!\}$), which contain a single term, generative and applicative functors, and pairs. Application of generative and applicative functors are syntactically distinguished. For example, the ML module:

```
struct
  type t = int
  val x = 12
end
```

could be elaborated $\langle (\!\vert\, \mathsf{int} \,\vert\!), \{\!\vert\, 12 \,\vert\!\}\rangle$.

Abstraction is introduced by *sealing:* In the module $M :> \sigma$, access to the components of M is limited to the interface σ. In particular, none of the type components are exposed outside M unless σ reveals them. To make this work, the calculus must prevent type components from being extracted from any sealed module, and to do so the calculus views sealing as a computational effect [5]. Generative functor application also induces a sealing effect, but applicative functor application does not. Sealing effects can always be mitigated by binding the sealed module to a variable; variables are always pure.

[1] This unusual typing is more convenient for call-by-value execution, as it allows us to expand $\mathsf{fix}_\tau\, e$ to $e(\lambda\_{:}\mathsf{unit}.\mathsf{fix}_\tau\, e)$.

$$
\begin{array}{lcl}
\mathsf{Fst}(1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\langle\!|\, k \,|\!\rangle) & \stackrel{\text{def}}{=} & k \\
\mathsf{Fst}(\langle\!|\, \tau \,|\!\rangle) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\,\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2) \\
\mathsf{Fst}(\Sigma\alpha{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & \Sigma\alpha{:}\,\mathsf{Fst}(\sigma_1).\,\mathsf{Fst}(\sigma_2)
\end{array}
$$

$$
\frac{\alpha/m \in \mathrm{Dom}(\Gamma)}{\Gamma \vdash \mathsf{Fst}(m) \gg \alpha} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\star) \gg \star} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\langle\!|\, c \,|\!\rangle) \gg c}
$$

$$
\frac{}{\Gamma \vdash \mathsf{Fst}(\langle\!|\, e \,|\!\rangle) \gg \star} \qquad \frac{}{\Gamma \vdash \mathsf{Fst}(\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M) \gg \star}
$$

$$
\frac{\Gamma, \alpha/m{:}\sigma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M) \gg \lambda\alpha{:}\,\mathsf{Fst}(\sigma).c}
$$

$$
\frac{\Gamma \vdash \mathsf{Fst}(M_1) \gg c_1 \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\begin{array}{c}\Gamma \vdash \mathsf{Fst}(M_1 \cdot M_2) \gg c_1 c_2 \\ \Gamma \vdash \mathsf{Fst}(\langle M_1, M_2\rangle) \gg \langle c_1, c_2\rangle\end{array}}
$$

$$
\frac{\Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\pi_i M) \gg \pi_i c} \qquad \frac{\Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash \mathsf{Fst}(\mathsf{let}\, x = e\, \mathsf{in}\, M) \gg c}
$$

Figure 2: Static portions

For technical reasons related to the avoidance problem [4, section 4.2.6], a module that let-binds another module or unpacks an existential must provide a signature annotation, and that annotation induces an implicit seal.

Extraction and twinned variables A term is extracted from a dynamic atomic module using the $\mathsf{Ext}\, M$ form. However, we do not include a syntactic form for extracting a constructor from a static atomic module. Instead, we adopt Dreyer's device wherein constructor extraction is a meta-operation, not a syntactic form. This has the virtue that it disentangles the kind/constructor language from the term/module language, and consequently the metatheoretic results of the singleton kind calculus [29] can be taken off-the-shelf.

This meta-operation is the judgement $\mathsf{Fst}(M) \gg c$, meaning that c is the the static portion of M.[2] For example, $\mathsf{Fst}\,\langle\!|\,\mathsf{int}\,|\!\rangle \gg \mathsf{int}$. The rules appear in Figure 2. As discussed above, a constructor can be extracted only from pure modules, so there is no case for forms, such as sealing, that are never pure. When the context is empty, we will sometimes write $\mathsf{Fst}(M)$ for that c such that $\mathsf{Fst}(M) \gg c$.

Notable is the treatment of variables. Every module variable is associated with a constructor variable that represents its static portion. Harper, *et al.* [9] and Dreyer [4] maintain this correspondence using a naming convention. However, this complicates the key principle of alpha-convertibility. Instead, we follow Lee, *et al.* [11] and maintain the correspondence explicitly. Module variable in binding positions and in the context appear in *twinned* form $\alpha/m : \sigma$, meaning that m has signature σ and its static portion is α.

[2]The name Fst is motivated by the construction in Harper, *et al.* [9], which explicitly renders modules as a pair of static and dynamic components.

$\vdash \Gamma\ \mathsf{ok}$	context formation
$\Gamma \vdash k : \mathsf{kind}$	kind formation
$\Gamma \vdash k \equiv k' : \mathsf{kind}$	kind equivalence
$\Gamma \vdash k \leq k' : \mathsf{kind}$	subkind
$\Gamma \vdash c : k$	constructor formation
$\Gamma \vdash c \equiv c' : k$	constructor equivalence
$\Gamma \vdash \sigma : \mathsf{sig}$	signature formation
$\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}$	signature equivalence
$\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}$	subsignature
$\Gamma \vdash e : \tau$	term formation
$\Gamma \vdash_\kappa M : \sigma$	module formation
$\Gamma \vdash \mathsf{Fst}(M) \gg c$	static portion

Figure 3: Static semantic judgements

Signatures Signatures include signatures for atomic modules, generative and application functor signatures, and signatures for pairs. The signatures for functors and for pairs are dependent, so one might expect to write them with a twinned left-hand-side, like $\Pi^{\mathsf{gn}}\alpha/m{:}\sigma_1.\sigma_2$. However, a module can never appear within any static syntax, including signatures, so the m variable will never be used.

In a twinned binding $\alpha/m{:}\sigma$, or in the binding $\alpha{:}\sigma$ within a dependent signature, α stands for the static portion of some module belonging to σ. Thus, α will have kind $\mathsf{Fst}(\sigma)$, which stands for the static portion of σ. The definition of Fst appears in Figure 2. Whenever $m : \sigma$ and $\mathsf{Fst}(m) \gg c$, we have $c : \mathsf{Fst}(\sigma)$.

Semantics The static semantics is given by several judgements: formation and equivalence for kinds, constructors, and signatures; subkind and subsignature relations; typing judgements for terms and modules; the static portion judgement discussed above; and a well-formedness judgement for contexts. The notation is summarized in Figure 3. The full rules are given in Figure 2 and Appendix A.

Of particular note is the typing judgement for modules, written $\Gamma \vdash_\kappa M : \sigma$. Here κ is a purity class: either P, indicating that the module is pure (unsealed), or I, indicating that it is impure (sealed). A "forget" rule allows pure modules to be viewed as impure.

The dynamic semantics is given by a standard, call-by-value, structured operational semantics, denoted by $\Gamma \vdash e \mapsto e'$ and $\Gamma \vdash M \mapsto M'$. (This allows us to evaluate open terms; in the typical case in which Γ is empty, we will omit the turnstile.) The value forms are:

$$
\begin{array}{rcl}
v & ::= & x \mid \star \mid \lambda x{:}\tau.e \mid \langle v, v\rangle \mid \Lambda\alpha{:}k.e \\
 & \mid & \mathsf{pack}\,[c, v]\,\mathsf{as}\,\exists\alpha{:}k.\tau \\
V & ::= & m \mid \star \mid \langle\!|\, c \,|\!\rangle \mid \langle\!|\, v \,|\!\rangle \mid \langle V, V\rangle \\
 & \mid & \lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M \mid \lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M
\end{array}
$$

We write $e{\downarrow}$ or $M{\downarrow}$ to mean that e or M evaluates to a value. We say that e diverges when there exists an infinite evaluation $\Gamma \vdash e \mapsto e_1 \mapsto e_2 \mapsto \cdots$.

Two rules illustrate issues that arise in the dynamic semantics of modules. The evaluation rule for sealed modules immediately removes the seal:

$$
\frac{}{\Gamma \vdash (M :> \sigma) \mapsto M}
$$

In essence, this step is the computational effect that the type system tracks. Rules that substitute for module variables, such as the beta rules for functors, also must substitute for the twinned constructor variable, which is obtained using the Fst judgement:

$$\frac{\Gamma \vdash \mathsf{Fst}(V_2) \gg c_2}{\Gamma \vdash (\lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M_1)V_2 \mapsto [c_2, V_2/\alpha, m]M_1}$$

This rule works because well-formed module values are always pure, so their static portion can always be obtained.

We can now state some preliminary results:

Lemma 2.1

- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *then* $\Gamma \vdash \mathsf{Fst}(M) \gg c$ *and* $\Gamma \vdash c : \mathsf{Fst}(\sigma)$.
- *If* $\Gamma \vdash_{\mathsf{I}} V : \sigma$ *then* $\Gamma \vdash_{\mathsf{P}} V : \sigma$.

Theorem 2.2 (Type preservation) *If* $\vdash \Gamma\ \mathsf{ok}$ *then:*

- *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e \mapsto e'$ *then* $\Gamma \vdash e' : \tau$.
- *If* $\Gamma \vdash_\kappa M : \sigma$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash_\kappa M' : \sigma$.
- *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$ *and* $\Gamma \vdash \mathsf{Fst}(M) \gg c$ *and* $\Gamma \vdash M \mapsto M'$ *then* $\Gamma \vdash \mathsf{Fst}(M') \gg c'$ *and* $\Gamma \vdash c \equiv c' : \mathsf{Fst}(\sigma)$.

Theorem 2.3 (Progress) *If* $\vdash e : \tau$ *then either* e *is a value or takes a step. If* $\vdash_\kappa M : \sigma$ *then either* M *is a value or takes a step.*

Theorem 2.4 (Determinism)

- *If* $\Gamma \vdash e \mapsto e_1$ *and* $\Gamma \vdash e \mapsto e_2$ *then* $e_1 = e_2$.
- *If* $\Gamma \vdash M \mapsto M_1$ *and* $\Gamma \vdash M \mapsto M_2$ *then* $M_1 = M_2$.
- *If* $\Gamma \vdash \mathsf{Fst}(M) \gg c_1$ *and* $\Gamma \vdash \mathsf{Fst}(M) \gg c_2$ *then* $c_1 = c_2$.

3 Contextual Approximation and Equivalence

The most common way to define contextual equivalence is first to define syntactic contexts ($\mathcal{C}$), which are terms with a hole. (To avoid confusion, we will reserve the unadorned word "context" for typing contexts Γ.) For example:

$$\mathcal{C} ::= [\,] \mid \lambda x{:}\tau.\mathcal{C} \mid e\,\mathcal{C} \mid \mathcal{C}\,e \mid \ldots etc \ldots$$

One writes $\mathcal{C}[e]$ to refer to filling the hole in $\mathcal{C}$ with e. Then one defines a typing judgement for contexts, $\mathcal{C} : (\Gamma \triangleright \tau) \to (\Gamma' \triangleright \tau')$, meaning that when the hole is filled with a term having type τ in context Γ, the resulting term has type τ' in context Γ'.

Finally, one says e and e' are contextually equivalent at τ in Γ (written $\Gamma \vdash e \approx e' : \tau$), if for any context $\mathcal{C} : (\Gamma \triangleright \tau) \to (\epsilon \triangleright \mathsf{unit})$, $\mathcal{C}[e]$ halts if and only if $\mathcal{C}[e']$ halts.

However, this definition is impractical for the module calculus. To begin with, one would need four different sorts of syntactic context (term with hole for a term, term with a hole for a module, etc.). The quadratic explosion worsens for the typing judgement, where pure and impure modules must be treated differently, resulting in nine different judgements. Furthermore, when typing a syntactic context with a hole for a pure module, one must know the static portion of the term that will fill the hole, further complicating the three judgements that deal with pure-module holes.

Thus, we employ an alternative definition that avoids syntactic contexts entirely. It is not hard to show that contextual equivalence is uniquely the coarsest congruence that is consistent with execution (in a sense that we will make precise shortly). We use this property as our definition.

Definition 3.1 An *indexed dynamic relation* R is a pair of relations: a relation on terms indexed by a context and type; and a relation on modules indexed by a context, signature, and purity class; such that if $\vdash \Gamma\ \mathsf{ok}$ then:

- if $\Gamma \vdash e\ R\ e' : \tau$ then $\Gamma \vdash e, e' : \tau$, and
- if $\Gamma \vdash_\kappa M\ R\ M' : \sigma$ then $\Gamma \vdash_\kappa M, M' : \sigma$, and
- if $\Gamma \vdash_{\mathsf{P}} M\ R\ M' : \sigma$ then $\Gamma \vdash c \equiv c' : \mathsf{Fst}(\sigma)$, where $\Gamma \vdash \mathsf{Fst}(M) \gg c$ and $\Gamma \vdash \mathsf{Fst}(M') \gg c'$.

Definition 3.2 Suppose R is an indexed dynamic relation.

- R is *compatible* if it respects the rules in Figure 4. (Informally, R is compatible if R-related expressions can be built from R-related subexpressions.)
- R is *substitutive* if it respects substitution and weaking for constructor, term, and module hypotheses. (For example, if $\Gamma, \alpha/m{:}\sigma, \Gamma' \vdash e\ R\ e' : \tau$ and $\Gamma \vdash_{\mathsf{P}} M : \sigma$ and $\Gamma \vdash \mathsf{Fst}(M) \gg c$ then $\Gamma, [c/\alpha]\Gamma' \vdash [c, M/\alpha, m]e\ R\ [c, M/\alpha, m]e' : [c/\alpha]\tau$.)
- R is a *quasicongruence* if it is compatible, substitutive, and reflexive.
- R is a *semicongruence* if it is compatible, substitutive, reflexive, and transitive.
- R is a *congruence* if it is compatible, substitutive, reflexive, transitive, and symmetric.
- R is *consistent* if it preserves module termination. (That is, if $\vdash_{\mathsf{I}} M\ R\ M' : \sigma$ and M halts then M' halts.)

We are interested in congruences for equivalences and in semicongruences for orders. Quasicongruences are useful for a technical device (Lemma 3.7). Although we define consistency in terms of module termination, it is easy to show that a consistent, compatible relation also preserves term termination.

Definition 3.3 *Contextual approximation* (written $\Gamma \vdash e \preceq e' : \tau$ and $\Gamma \vdash_\kappa M \preceq M' : \sigma$) is defined as the union of all consistent semicongruences.

Theorem 3.4 *Contextual approximation is the coarsest consistent semicongruence.*

Proof Sketch

By definition, contextual approximation includes all consistent semicongruences. It remains to show it is itself a consistent semicongruence. Let $\preceq^+$ be the transitive closure of contextual approximation. Then $\preceq^+$ is clearly substitutive, reflexive, transitive, and consistent, and we may show that it is compatible as well. Therefore $\preceq^+$ is a consistent semicongruence, so $\preceq^+ \subseteq \preceq$. Also $\preceq \subseteq \preceq^+$, so $\preceq = \preceq^+$.

We could define contextual equivalence as the union of consistent congruences, but it is convenient to define it as the symmetrization of contextual approximation:

Definition 3.5 Two expressions are *contextually equivalent* (written $\Gamma \vdash e \approx e' : \tau$ and $\Gamma \vdash_\kappa M \approx M' : \sigma$) when each expression contextually approximates the other.

Theorem 3.6 *Contextual equivalence is the coarsest consistent congruence.*

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma, x{:}\tau_1 \vdash e\ R\ e' : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e\ R\ \lambda x{:}\tau_1.e' : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1\ R\ e_1' : \tau \to \tau' \quad \Gamma \vdash e_2\ R\ e_2' : \tau}{\Gamma \vdash e_1 e_2\ R\ e_1' e_2' : \tau'} \qquad \frac{\Gamma \vdash e_1\ R\ e_1' : \tau_1 \quad \Gamma \vdash e_2\ R\ e_2' : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle\ R\ \langle e_1', e_2'\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e\ R\ e' : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e\ R\ \pi_1 e' : \tau_1} \qquad \frac{\Gamma \vdash e\ R\ e' : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 e\ R\ \pi_2 e' : \tau_2} \qquad \frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k; \Gamma \vdash e\ R\ e' : \tau}{\Gamma \vdash \Lambda\alpha{:}k.e\ R\ \Lambda\alpha{:}k.e' : \forall\alpha{:}k.\tau} \qquad \frac{\Gamma \vdash e\ R\ e' : \forall\alpha{:}k.\tau \quad \Gamma \vdash c : k}{\Gamma \vdash e[c]\ R\ e'[c] : [c/\alpha]\tau}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash e\ R\ e' : [c/\alpha]\tau \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists\alpha{:}k.\tau\ R\ \mathsf{pack}\,[c, e']\,\mathsf{as}\,\exists\alpha{:}k.\tau : \exists\alpha{:}k.\tau} \qquad \frac{\Gamma \vdash e_1\ R\ e_1' : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2\ R\ e_2' : \tau' \quad \Gamma \vdash \tau' : \mathsf{T}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2\ R\ \mathsf{unpack}\,[\alpha, x] = e_1'\,\mathsf{in}\,e_2' : \tau'}$$

$$\frac{\Gamma \vdash \tau : \mathsf{T} \quad \Gamma \vdash e\ R\ e' : (\mathsf{unit} \to \tau) \to \tau}{\Gamma \vdash \mathsf{fix}_\tau\ e\ R\ \mathsf{fix}_\tau\ e' : \tau} \qquad \frac{\Gamma \vdash e_1\ R\ e_1' : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2\ R\ e_2' : \tau_2}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2\ R\ \mathsf{let}\,x = e_1'\,\mathsf{in}\,e_2' : \tau_2}$$

$$\frac{\Gamma \vdash M\ R\ M' : \sigma \quad \Gamma, \alpha/m{:}\sigma \vdash e\ R\ e' : \tau \quad \Gamma \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e\ R\ \mathsf{let}\,\alpha/m = M'\,\mathsf{in}\,e' : \tau} \qquad \frac{\Gamma \vdash M\ R\ M' : \langle\!|\,\tau\,|\!\rangle}{\Gamma \vdash \mathsf{Ext}\,M\ R\ \mathsf{Ext}\,M' : \tau} \qquad \frac{\Gamma \vdash e\ R\ e' : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash e\ R\ e' : \tau'}$$

$$\frac{\Gamma \vdash c : k}{\Gamma_{\mathsf{P}} \vdash (\!|\,c\,|\!)\ R\ (\!|\,c\,|\!) : (\!|\,k\,|\!)} \qquad \frac{\Gamma \vdash e\ R\ e' : \tau}{\Gamma \vdash_{\mathsf{P}} \langle\!|\,e\,|\!\rangle\ R\ \langle\!|\,e'\,|\!\rangle : \langle\!|\,\tau\,|\!\rangle} \qquad \frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{I}} M\ R\ M' : \sigma_2}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M\ R\ \lambda^{\mathsf{gn}}\alpha/m{:}\sigma_1.M' : \Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1\ R\ M_1' : \Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma' \quad \Gamma \vdash_{\mathsf{P}} M_2\ R\ M_2' : \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_{\mathsf{I}} M_1 M_2\ R\ M_1' M_2' : [c_2/\alpha]\sigma'} \qquad \frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M\ R\ M' : \sigma_2}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M\ R\ \lambda^{\mathsf{ap}}\alpha/m{:}\sigma_1.M' : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_\kappa M_1\ R\ M_1' : \Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma' \quad \Gamma \vdash_{\mathsf{P}} M_2\ R\ M_2' : \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_\kappa M_1 \cdot M_2\ R\ M_1' \cdot M_2' : [c_2/\alpha]\sigma'} \qquad \frac{\Gamma \vdash_\kappa M_1\ R\ M_1' : \sigma_1 \quad \Gamma \vdash_\kappa M_2\ R\ M_2' : \sigma_2}{\Gamma \vdash_\kappa \langle M_1, M_2\rangle\ R\ \langle M_1', M_2'\rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M\ R\ M' : \Sigma\alpha{:}\sigma_1.\sigma_2}{\Gamma \vdash_{\mathsf{P}} \pi_1 M\ R\ \pi_1 M' : \sigma_1} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M\ R\ M' : \Sigma\alpha{:}\sigma_1.\sigma_2 \quad \Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash_{\mathsf{P}} \pi_2 M\ R\ \pi_2 M' : [\pi_1 c/\alpha]\sigma_2}$$

$$\frac{\Gamma \vdash e\ R\ e' : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash_{\mathsf{I}} M\ R\ M' : \sigma \quad \Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash_{\mathsf{I}} \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(M : \sigma)\ R\ \mathsf{unpack}\,[\alpha, x] = e'\,\mathsf{in}\,(M' : \sigma) : \sigma} \qquad \frac{\Gamma \vdash e\ R\ e' : \tau \quad \Gamma, x{:}\tau \vdash_\kappa M\ R\ M' : \sigma}{\Gamma \vdash_\kappa \mathsf{let}\,x = e\,\mathsf{in}\,M\ R\ \mathsf{let}\,x = e'\,\mathsf{in}\,M' : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1\ R\ M_1' : \sigma \quad \Gamma, \alpha/m{:}\sigma \vdash_{\mathsf{I}} M_2\ R\ M_2' : \sigma' \quad \Gamma \vdash \sigma' : \mathsf{sig}}{\Gamma \vdash_{\mathsf{I}} \mathsf{let}\,\alpha/m = M_1\,\mathsf{in}\,(M_2 : \sigma')\ R\ \mathsf{let}\,\alpha/m = M_1'\,\mathsf{in}\,(M_2' : \sigma') : \sigma'} \qquad \frac{\Gamma \vdash_{\mathsf{I}} M\ R\ M' : \sigma}{\Gamma \vdash_{\mathsf{I}} (M :> \sigma)\ R\ (M' :> \sigma) : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M\ R\ M' : \sigma}{\Gamma \vdash_{\mathsf{I}} M\ R\ M' : \sigma} \qquad \frac{\Gamma \vdash_\kappa M\ R\ M' : \sigma \quad \Gamma \vdash \sigma \le \sigma'}{\Gamma \vdash_\kappa M\ R\ M' : \sigma'}$$

Figure 4: Compatibility

Proof Sketch

Suppose R is a consistent congruence. Then R and its transpose are consistent semicongruences, so they are both contained in $\preceq$. Hence $R \subseteq \approx$. It remains to show that contextual equivalence is itself a consistent congruence. Clearly it is consistent and symmetric. We may show that semicongruences are closed under transpose and intersection, so contextual equivalence is a semicongruence as well.

We can use the transitive-closure and transpose devices from Theorems 3.4 and 3.6 to prove a very useful lemma:

Lemma 3.7 *If R is a consistent quasicongruence, then R is contained in contextual approximation. If, in addition, R's transpose is consistent, then R is contained in contextual equivalence.*

Proof Sketch

Suppose R is a consistent quasicongruence. Let R^+ be the transitive closure of R. Clearly R^+ is substitutive, reflexive, transitive, and consistent, and we may show that it is compatible as well. Therefore R^+ is a consistent semicongruence. Thus $R \subseteq R^+ \subseteq \preceq$.

Suppose R's transpose is also consistent. Then the transpose is a consistent semicongruence, so also $R^{\mathrm{op}} \subseteq \preceq$. Hence $R \subseteq \approx$.

In the sequel, we will use two key properties of contextual equivalence and approximation without explicit reference:

Lemma 3.8 *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e \mapsto e'$ then $\Gamma \vdash e \approx e' : \tau$, and similarly for modules.*

Proof Sketch

Let β_{v} be the compatible, reflexive, and transitive closure of evaluation (restricted to well-formed expressions). Clearly β_{v} is a quasicongruence. We may show using the Standardization Theorem [23, 21, 3] that β_{v} and its transpose are consistent. By Lemma 3.7, $\beta_{\mathsf{v}} \subseteq \approx$.

Lemma 3.9 *If $\Gamma \vdash e, e' : \tau$ and e diverges then $\Gamma \vdash e \preceq e' : \tau$.*

Proof Sketch

Let R be the smallest quasicongruence containing the rule:

$$\frac{\Gamma \vdash e, e' : \tau \quad e \text{ diverges}}{\Gamma \vdash e\ R\ e' : \tau}$$

By Lemma 3.7, it is sufficient to show that R is consistent. It is easy to show that if $\Gamma \vdash v\ R\ e$ then e is a value. We may show by induction that if $\Gamma \vdash e_1\ R\ e_2 : \tau$ and $\Gamma \vdash e_1 \mapsto e_1'$ then there exists e_2' such that $\Gamma \vdash e_1'\ R\ e_2' : \tau$ and $\Gamma \vdash e_2 \mapsto^* e_2'$. It follows that if $\Gamma \vdash e_1\ R\ e_2 : \tau$ and $\Gamma \vdash e_1 \mapsto^* v_1$ then $\Gamma \vdash e_2 \mapsto^* v_2$.

3.1 Compactness

In support of recursion, we will need a compactness lemma. As usual, we define $\bot_\tau$ to be any divergent term of type τ, and define:

$$\begin{array}{lcl} \mathsf{fix}_\tau^0\ f & \stackrel{\text{def}}{=} & \bot_\tau \\ \mathsf{fix}_\tau^{i+1}\ f & \stackrel{\text{def}}{=} & f(\lambda\_{:}\,\mathsf{fix}_\tau^i\ f) \\ \mathsf{fix}_\tau^\omega\ f & \stackrel{\text{def}}{=} & \mathsf{fix}_\tau\ f \end{array}$$

Observe that $\mathsf{fix}_\tau^i\ f \preceq \mathsf{fix}_\tau\ f$, and if $i \le j$ then $\mathsf{fix}_\tau^i\ f \preceq \mathsf{fix}_\tau^j\ f : \tau$. We write $e^{f[o]}$ to mean $[\lambda\_{:}\mathsf{unit}.\,\mathsf{fix}_\tau^o\ f/x]e$ (when the type τ is clear from context). Note that this notation designates a particular free variable (namely x) to receive an approximation.

Lemma 3.10 (Compactness) *Suppose $\vdash f : (\mathsf{unit} \to \rho) \to \rho$ and $x{:}(\mathsf{unit} \to \rho) \vdash e : \tau$. If $e^{f[\omega]}$ halts then there exists i such that for all $j \ge i$, $e^{f[j]}$ halts.*

Proof Sketch

The result is a corollary of a stronger simulation result. For terms, the simulation result says: Suppose $\vdash f : (\mathsf{unit} \to \rho) \to \rho$ and $x{:}(\mathsf{unit} \to \rho) \vdash e : \tau$. If $e^{f[\omega]} \mapsto^* v$, then v can be written in the form $v'^{f[\omega]}$, such that there exists i such that for all $j \ge i$, $e^{f[j]} \succeq v'^{f[j-i]} : \tau$. Since $v'^{f[j-i]}$ certainly halts, it follows that $e^{f[j]}$ halts.

The proof is by complete induction on the number of evaluation steps, with an inner induction on the typing derivation of e. The number i is how many times the evaluation unrolls any instance of the fixed point. The proof builds a sequence of contextual approximations that simulates the evaluation sequence, but ensures that all instances of the fixed point are uniformly unrolled exactly i times, whether they need to be or not.

4 Biorthogonality

One tricky issue dealing with recursion is the problem of admissibility. Informally, a relation is admissible if a recursive term belongs to the relation whenever its finite approximations belong to the relation. We can prove membership in an admissible relation using fixed point induction.

Most type operators can be shown to preserve admissibility, but some cannot. In fact, the ones that fail are the ones that interest us most: those that provide type abstraction such as existential types and dependent-sum signatures.

To deal with this issue, we employ Pitts' biorthogonality technique [22]. The technique provides a closure condition that is stronger than admissibility but easier to work with. Furthermore, it provides a closure operator that can be applied in the cases that are not already closed. In passing, biorthogonality also provides closure with respect to contextual equivalence.

Let us write $\mathrm{Rel}_{\tau,\tau'}$ for the set of binary relations between closed values of type τ and τ'. Given such a relation R, R^{s} is its dual space, containing the continuations that agree on everything related by R. Conversely, given a relation S on continuations, S^{t} is *its* dual space, containing the terms everything related by S agree on.

Definition 4.1 Suppose $R \in \mathrm{Rel}_{\tau,\tau'}$. Then:

$$\begin{array}{rl} R^{\mathsf{s}} \stackrel{\text{def}}{=} \{(v : \tau \to \mathsf{unit}, v' : \tau' \to \mathsf{unit}) \mid & \\ \qquad \forall (w, w') \in R,\ v\,w{\downarrow} \Leftrightarrow v'\,w'{\downarrow}\} & \end{array}$$

Definition 4.2 Suppose S is a binary relation between closed values of type $\tau \to \mathsf{unit}$ and $\tau' \to \mathsf{unit}$. Then:

$$S^{\mathsf{t}} \stackrel{\text{def}}{=} \{(w : \tau, w' : \tau') \mid \forall (v, v') \in S,\ v\,w{\downarrow} \Leftrightarrow v'\,w'{\downarrow}\}$$

Observe that R^{st} belongs to $\mathrm{Rel}_{\tau,\tau'}$ whenever R does. We can show that $-^{\mathsf{st}}$ is indeed a closure operator:

Lemma 4.3 *Suppose $R, R' \in \mathrm{Rel}_{\tau,\tau'}$. Then:*

- $R \subseteq R^{\mathsf{st}}$
- *If $R \subseteq R'$ then $R^{\mathsf{st}} \subseteq R'^{\mathsf{st}}$.*
- $R^{\mathsf{ststst}} = R^{\mathsf{st}}$

We make similar definitions for modules, using $\sigma \stackrel{\mathsf{gn}}{\to} 1$ (that is, $\Pi^{\mathsf{gn}}\_{:}\sigma.1$) as the signature of continuations for σ, and $-^{\mathsf{st}}$ for modules is likewise a closure operator.

Finally, we say that R is *Pitts closed* if $R = R^{\mathsf{st}}$. Note that, by idempotence, R^{st} is always Pitts closed. Pitts-closed relations are closed under contextual equivalence, and are admissible:

Lemma 4.4 *Suppose $R \in \mathrm{Rel}_{\tau_1,\tau_2}$ is Pitts closed. Then R is closed under contextual equivalence. (And similarly for modules.)*

Proof Sketch

Suppose $(v_1, v_2) \in R$ and $v_1 \approx v_1' : \tau_1$ and $v_2 \approx v_2' : \tau_2$. Since R is Pitts closed, it suffices to show that $(v_1', v_2') \in R^{\mathsf{st}}$. Let $(f_1, f_2) \in R^{\mathsf{s}}$. Then $f_1 v_1{\downarrow} \Leftrightarrow f_2 v_2{\downarrow}$, but the former is contextually equivalent to $f_1 v_1'$ and the latter to $f_2 v_2'$, so $f_1 v_1'{\downarrow} \Leftrightarrow f_2 v_2'{\downarrow}$ as required. The proof for modules is similar.

Lemma 4.5 (Admissibility) *Suppose $R \in \mathrm{Rel}_{\tau_1,\tau_2}$ is Pitts closed. Suppose further that (for $i = 1, 2$) $\vdash f_i : (\mathsf{unit} \to \rho_i) \to \rho_i$ and $x{:}(\mathsf{unit} \to \rho_i) \vdash v_i : \tau_i$. If $(v_1^{f_1[j]}, v_2^{f_2[j]}) \in R$ for all j, then $(v_1^{f_1[\omega]}, v_2^{f_2[\omega]}) \in R$.*

Proof Sketch

Let $(g_1, g_2) \in R^{\mathsf{s}}$. Since R is Pitts closed, it suffices to show that $g_1(v_1^{f_1[\omega]}){\downarrow} \Leftrightarrow g_2(v_2^{f_2[\omega]}){\downarrow}$. Suppose $g_1(v_1^{f_1[\omega]})$ halts. By compactness, there exists j such that $g_1(v_1^{f_1[j]})$ halts. Recall that $(v_1^{f_1[j]}, v_2^{f_2[j]}) \in R$. Since $(g_1, g_2) \in R^{\mathsf{s}}$, $g_2(v_2^{f_2[j]})$ halts. Finally, $g_2(v_2^{f_2[j]}) \preceq g_2(v_2^{f_2[\omega]}) : \mathsf{unit}$, so $g_2(v_2^{f_2[\omega]})$ halts. The reverse is similar.

4.1 Evaluation Closure

We lift value relations to terms using evaluation closure:

Definition 4.6 Suppose $R \in \mathrm{Rel}_{\tau,\tau'}$. Then:

$$R^{\mathsf{ev}} \stackrel{\text{def}}{=} \{(e:\tau, e':\tau') \mid \\ e\downarrow \Leftrightarrow e'\downarrow \text{and} \\ \forall v, v'.(e \mapsto^* v) \Rightarrow (e' \mapsto^* v') \Rightarrow (v, v') \in R\}$$

Again, we make a similar definition for modules. Observe that if R is closed under contextual equivalence, then so is R^{ev}.

The beauty of stev closure is it behaves something like a monad. If we wish to show that a pair of terms are related by R^{stev} and they depend on a pair of terms related by Q^{stev}, we may assume without loss of generality that the latter pair are simply related by Q.

Definition 4.7 Suppose $x{:}\rho \vdash e : \tau$. We say that x is *active* in e if, for all closed $e' : \rho$, $[e'/x]e\downarrow$ implies $e'\downarrow$.

Lemma 4.8 (Monadic stev) *Suppose $Q \in \mathrm{Rel}_{\rho_1,\rho_2}$ and $R \in \mathrm{Rel}_{\tau_1,\tau_2}$ and (for $i = 1, 2$), $x{:}\rho_i \vdash e_i : \tau_i$. Suppose further that x is active in e_1 and e_2. If:*

$$\forall (v_1, v_2) \in Q.\, ([v_1/x]e_1, [v_2/x]e_2) \in R^{\mathsf{stev}}$$

then:

$$\forall (e_1', e_2') \in Q^{\mathsf{stev}}.\, ([e_1'/x]e_1, [e_2'/x]e_2) \in R^{\mathsf{stev}}$$

Proof Sketch

Suppose $(e_1', e_2') \in Q^{\mathsf{stev}}$. For the first condition, suppose $[e_1'/x]e_1\downarrow$. Then $e_1'\downarrow$ so $e_2'\downarrow$. Let $e_i' \mapsto^* v_i$. Then $(v_1, v_2) \in Q^{\mathsf{st}}$. Let $f_i = \lambda x{:}\rho_i.\mathsf{let}\ \_ = e_i\ \mathsf{in}\ \star$. It is easy to show that $(f_1, f_2) \in Q^{\mathsf{s}}$. It follows that $f_1 v_1\downarrow \Leftrightarrow f_2 v_2\downarrow$, but $f_i v_i\downarrow \Leftrightarrow [v_i/x]e_i\downarrow$ and $[v_i/x]e_i \approx [e_i'/x]e_i$. Thus $[e_2'/x]e_2\downarrow$. The reverse is similar.

For the second condition, suppose $[e_1'/x]e_1 \mapsto^* w_1$ and $[e_2'/x]e_2 \mapsto^* w_2$. Then $e_1'\downarrow$ so $e_2'\downarrow$. Again, let $e_i' \mapsto^* v_i$. Again $(v_1, v_2) \in Q^{\mathsf{st}}$. Suppose $(g_1, g_2) \in R^{\mathsf{s}}$. Let $h_i = \lambda x{:}\rho_i.g_i e_i$. We claim that $(h_1, h_2) \in Q^{\mathsf{s}}$. It follows that $h_1 v_1\downarrow \Leftrightarrow h_2 v_2$, but $h_i v_i \approx g_i([v_i/x]e_i) \approx g_i([e_i'/x]e_i) \approx g_i w_i$. Hence $(w_1, w_2) \in R^{\mathsf{st}}$.

To prove the claim, suppose that $(u_1, u_2) \in Q$. Then $([u_1/x]e_1, [u_2/x]e_2) \in R^{\mathsf{stev}}$. Suppose $h_1 u_1\downarrow$. Then $[u_1/x]e_1\downarrow$ so $[u_2/x]e_2\downarrow$. Let $[u_i/x]e_i \mapsto^* u_i'$. Then $(u_1', u_2') \in R^{\mathsf{st}}$. It follows that $g_1 u_1'\downarrow \Leftrightarrow g_2 u_2'\downarrow$. But $g_i u_i' \approx g_i([u_i/x]e_i) \approx h_i u_i$. Thus $h_2 u_2\downarrow$. The reverse is similar. Hence $(h_1, h_2) \in Q^{\mathsf{s}}$.

We may also establish similar lemmas for modules that depend on modules, for modules that depend on terms, and for terms that depend on modules.

5 Logical Equivalence

Unlike contextual equivalence, logical equivalence defines the equivalence of expressions based on the operational behavior of those expressions themselves. Two expressions are equivalent when the operations that can be performed on them (according to their type) produce equivalent results.

As usual, we define logical equivalence using a variation of Girard's methods of candidates [6]. To show that two polymorphic functions e_1 and e_2 of type $\forall\alpha{:}\mathsf{T}.\tau$ are equivalent, we quantify over all type arguments τ_1 and τ_2 and all *candidates* R, and show that $e_1[\tau_1]$ and $e_2[\tau_2]$ are equivalent at τ, wherein the free occurrences of α are interpreted using R.

The candidate R is a relation between closed values of type τ_1 and τ_2. If $\tau_1 = \tau_2$, R might happen to be the logical interpretation of τ_1, but in general it can be any relation satisfying some closure properties. The license to pick a relation *other* than the logical interpretation of a type is what gives the Abstraction theorem its power.

In the presence of higher-order type constructors, the notion of a candidate must be generalized. Suppose c_1 and c_2 belong to $\mathsf{T} \to \mathsf{T}$. A candidate relating c_1 and c_2 is (more or less) a function that, for any τ_1 and τ_2, maps relations between τ_1 and τ_2 to relations between $c_1\tau_1$ and $c_2\tau_2$.

We say a *simple kind* is a kind containing no singletons. Observe that every kind can be erased (written k°) to a simple kind by replacing singletons with T. We define *pre-candidates* indexed by simple kinds:

$$\begin{array}{ll}
\mathrm{Val} & \stackrel{\text{def}}{=} \{v \mid \vdash v : \tau\} \\
\mathrm{Con} & \stackrel{\text{def}}{=} \{c \mid \vdash c : k\} \\
\mathrm{PreCand}_{\mathsf{T}} & \stackrel{\text{def}}{=} \mathcal{P}(\mathrm{Val} \times \mathrm{Val}) \\
\mathrm{PreCand}_{k_1 \to k_2} & \stackrel{\text{def}}{=} \mathrm{Con} \times \mathrm{Con} \times \mathrm{PreCand}_{k_1} \to \mathrm{PreCand}_{k_2} \\
\mathrm{PreCand}_{k_1 \times k_2} & \stackrel{\text{def}}{=} \mathrm{PreCand}_{k_1} \times \mathrm{PreCand}_{k_2} \\
\mathrm{PreCand}_{1} & \stackrel{\text{def}}{=} \{\star\}
\end{array}$$

Here $\to$ refers to the set-theoretic partial function space. We will use Q to range over pre-candidates in general, Φ for pre-candidates over function kinds, and R for pre-candidates over T (*i.e.,* relations). A *candidate* will be a pre-candidate that belongs to the interpretation of a kind.

An *environment* is a mapping from constructor variables to triples (c, c', Q) (where Q is intended to relate c and c'), from term variables to pairs (v, v'), and from module variables to pairs (V, V'). We write η_{L} and η_{R} for the substitutions that maps every variable to the first and second constituent of the triple or pair that η maps that variable to.

5.1 The Logical Interpretation

Kinds, constructors, and signatures are given a logical interpretation relative to a environment, which we denote by $[\![k]\!]_\eta$, $[\![c]\!]_\eta$, and $[\![\sigma]\!]_\eta$. The full definition is given in Figures 5 and 6. Note that η's mappings for term and modules variables are irrelevant in the logical interpretation; they will be used in logical equivalence itself.

The logical interpretation of a constructor is a candidate. In the case of a type, that candidate (belonging to $\mathrm{PreCand}_{\mathsf{T}}$) will be a relation over closed values, which we will use for logical equivalence of terms.

The logical interpretation of a kind is a set of quadruples (c, c', Q, Q'), in which c and c' are constructors belonging to the kind, and Q and Q' are two candidates that relate c to c'. This induces a partial equivalence relation on candidates: when $(c, c', Q, Q') \in [\![k]\!]_\eta$, we say that Q and Q' are semantically equivalent candidates relating c to c'. As usual with PERs, we use self-equivalence for membership, so Q relates c to c' (written $(c, c', Q) \in [\![k]\!]_\eta^{\mathsf{set}}$) when $(c, c', Q, Q) \in [\![k]\!]_\eta$.

Lemma 5.1 *Suppose $(c, c', Q, Q') \in [\![k]\!]_\eta$. Then $\vdash c : \eta_{\mathsf{L}}(k)$ and $\vdash c' : \eta_{\mathsf{R}}(k)$ and $Q, Q' \in \mathrm{PreCand}_{k^\circ}$.*

$$
\begin{array}{lcl}
[\![\mathsf{T}]\!]_\eta & \stackrel{\text{def}}{=} & \{(\tau,\tau',R,R) \mid \\
& & \quad \vdash \tau,\tau' : \mathsf{T} \\
& & \quad \text{and } R \in \mathrm{Rel}_{\tau,\tau'} \text{ is Pitts closed}\} \\
[\![\mathsf{S}(c)]\!]_\eta & \stackrel{\text{def}}{=} & \{(\tau,\tau',[\![c]\!]_\eta,[\![c]\!]_\eta) \mid \\
& & \quad \vdash \tau \equiv \eta_{\mathsf{L}}(c) : \mathsf{T} \text{ and } \vdash \tau' \equiv \eta_{\mathsf{R}}(c) : \mathsf{T} \\
& & \quad [\![c]\!]_\eta \in \mathrm{Rel}_{\tau,\tau'} \text{ is Pitts closed}\} \\
[\![\Pi\alpha{:}k_1.k_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(c,c',\Phi,\Phi') \mid \\
& & \quad \vdash c,c' : \eta_{\mathsf{L},\mathsf{R}}(\Pi\alpha{:}k_1.k_2) \text{ and} \\
& & \quad \Phi,\Phi' \in \mathrm{PreCand}_{\Pi\alpha:k_1.k_2} \text{ and} \\
& & \quad \forall(d,d',Q,Q') \in [\![k_1]\!]_\eta.\, \forall d'',d'''. \\
& & \qquad \vdash d \equiv d'' : \eta_{\mathsf{L}}(k_1) \\
& & \qquad \Rightarrow \vdash d' \equiv d''' \in \eta_{\mathsf{R}}(k_1) \\
& & \qquad \Rightarrow (c\,d, c'\,d', \Phi(d,d',Q), \Phi'(d'',d''',Q')) \\
& & \qquad\quad \in [\![k_2]\!]_{\eta,\alpha\mapsto(d,d',Q)}\} \\
[\![\Sigma\alpha{:}k_1.k_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(c,c',P,P') \mid \\
& & \quad \vdash c,c' : \eta_{\mathsf{L},\mathsf{R}}(\Sigma\alpha{:}k_1.k_2) \text{ and} \\
& & \quad P,P' \in \mathrm{PreCand}_{\Sigma\alpha:k_1.k_2} \text{ and} \\
& & \quad (\pi_1 c,\pi_1 c',\pi_1 P,\pi_1 P') \in [\![k_1]\!]_\eta \text{ and} \\
& & \quad (\pi_2 c,\pi_2 c',\pi_2 P,\pi_2 P') \\
& & \qquad \in [\![k_2]\!]_{\eta,\alpha\mapsto(\pi_1 c,\pi_1 c',\pi_1 P)}\} \\
[\![1]\!]_\eta & \stackrel{\text{def}}{=} & \{(c,c',\langle\rangle,\langle\rangle) \mid \vdash c,c' : 1\} \\
[\![k]\!]^{\mathsf{set}}_\eta & \stackrel{\text{def}}{=} & \{(c,c',Q) \mid (c,c',Q,Q) \in [\![k]\!]_\eta\} \\
\\
[\![\alpha]\!]_\eta & \stackrel{\text{def}}{=} & Q \text{ where } \eta(\alpha) = (c,c',Q) \\
[\![\lambda\alpha{:}k.c]\!]_\eta & \stackrel{\text{def}}{=} & \underline{\lambda}(d,d',Q) \in [\![k]\!]^{\mathsf{set}}_\eta.\, [\![c]\!]_{\eta,\alpha\mapsto(d,d',Q)} \\
[\![c_1\,c_2]\!]_\eta & \stackrel{\text{def}}{=} & [\![c_1]\!]_\eta(\eta_{\mathsf{L}}(c_2),\eta_{\mathsf{R}}(c_2),[\![c_2]\!]_\eta) \\
[\![\langle c_1,c_2\rangle]\!]_\eta & \stackrel{\text{def}}{=} & \langle [\![c_1]\!]_\eta,[\![c_2]\!]_\eta\rangle \\
[\![\pi_i c]\!]_\eta & \stackrel{\text{def}}{=} & \pi_i [\![c]\!]_\eta \\
[\![\star]\!] & \stackrel{\text{def}}{=} & \langle\rangle \\
[\![\mathsf{unit}]\!]_\eta & \stackrel{\text{def}}{=} & \{(\star,\star)\} \\
[\![\tau_1 \to \tau_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(v,v') \mid \vdash v,v' : \eta_{\mathsf{L},\mathsf{R}}(\tau_1 \to \tau_2) \text{ and} \\
& & \quad \forall(w,w') \in [\![\tau_1]\!]_\eta.\, (v\,w, v'\,w') \in [\![\tau_2]\!]^{\mathsf{ev}}_\eta\} \\
[\![\tau_1 \times \tau_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(v,v') \mid \\
& & \quad \vdash v,v' : \eta_{\mathsf{L},\mathsf{R}}(\tau_1 \times \tau_2) \text{ and} \\
& & \quad \exists v_1,v_1',v_2,v_2'. \\
& & \qquad v = \langle v_1,v_2\rangle \text{ and } v' = \langle v_1',v_2'\rangle \text{ and} \\
& & \qquad (v_1,v_1') \in [\![\tau_1]\!]_\eta \text{ and } (v_2,v_2') \in [\![\tau_2]\!]_\eta\} \\
[\![\forall\alpha{:}k.\tau]\!]_\eta & \stackrel{\text{def}}{=} & \{(v,v') \mid \\
& & \quad \vdash v,v' : \eta_{\mathsf{L},\mathsf{R}}(\forall\alpha{:}k.\tau) \text{ and} \\
& & \quad \forall(c,c',Q) \in [\![k]\!]^{\mathsf{set}}_\eta. \\
& & \qquad (v[c],v'[c']) \in [\![\tau]\!]^{\mathsf{ev}}_{\eta,\alpha\mapsto(c,c',Q)}\} \\
[\![\exists\alpha{:}k.\tau]\!]_\eta & \stackrel{\text{def}}{=} & \{(v,v') \mid \\
& & \quad \vdash v,v' : \eta_{\mathsf{L},\mathsf{R}}(\exists\alpha{:}k.\tau) \text{ and} \\
& & \quad \exists(c,c',Q) \in [\![k]\!]^{\mathsf{set}}_\eta.\, \exists v_0,v_0',k',k'',\tau',\tau''. \\
& & \qquad v = \mathsf{pack}\,[c,v_0]\,\mathsf{as}\,\exists\alpha{:}k'.\tau' \text{ and} \\
& & \qquad v' = \mathsf{pack}\,[c',v_0']\,\mathsf{as}\,\exists\alpha{:}k''.\tau'' \text{ and} \\
& & \qquad (v_0,v_0') \in [\![\tau]\!]_{\eta,\alpha\mapsto(c,c',Q)}\}^{\mathsf{st}}
\end{array}
$$

Figure 5: Logical Interpretation (Kinds and Constructors)

$$
\begin{array}{lcl}
[\![1]\!]_\eta & \stackrel{\text{def}}{=} & \{(\star,\star,\langle\rangle)\} \\
[\![(\!| k |\!)]\!]_\eta & \stackrel{\text{def}}{=} & \{((\!| c |\!),(\!| c' |\!),Q) \mid (c,c'Q) \in [\![k]\!]^{\mathsf{set}}_\eta\} \\
[\![(\!| \tau |\!)]\!]_\eta & \stackrel{\text{def}}{=} & \{((\!| v |\!),(\!| v' |\!),\langle\rangle) \mid \\
& & \quad \vdash v,v' : \eta_{\mathsf{L},\mathsf{R}}(\tau) \text{ and } (v,v') \in [\![\tau]\!]_\eta\} \\
[\![\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(V,V',\langle\rangle) \mid \\
& & \quad \vdash V,V' : \eta_{\mathsf{L},\mathsf{R}}(\Pi^{\mathsf{gn}}\alpha{:}\sigma_1.\sigma_2) \text{ and} \\
& & \quad \forall(W,W',Q) \in [\![\sigma_1]\!]_\eta. \\
& & \qquad (VW,V'W') \in [\![\sigma_2]\!]^{\mathsf{iev}}_{\eta,\alpha\mapsto(\mathsf{Fst}(W),\mathsf{Fst}(W'),Q)}\} \\
[\![\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(V,V',\Phi) \mid \\
& & \quad \vdash V,V' : \eta_{\mathsf{L},\mathsf{R}}(\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2) \text{ and} \\
& & \quad (\mathsf{Fst}(V),\mathsf{Fst}(V'),\Phi) \in [\![\mathsf{Fst}(\Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2)]\!]^{\mathsf{set}}_\eta \\
& & \quad \text{and } \forall(W,W',Q) \in [\![\sigma_1]\!]_\eta. \\
& & \qquad (V\cdot W,V'\cdot W',\Phi(c,c',Q)) \\
& & \qquad\quad \in [\![\sigma_2]\!]^{\mathsf{pev}}_{\eta,\alpha\mapsto(\mathsf{Fst}(W),\mathsf{Fst}(W'),Q)}\} \\
[\![\Sigma\alpha{:}\sigma_1.\sigma_2]\!]_\eta & \stackrel{\text{def}}{=} & \{(V,V',P) \mid \\
& & \quad \vdash V,V' : \eta_{\mathsf{L},\mathsf{R}}(\Sigma\alpha{:}\sigma_1.\sigma_2) \text{ and} \\
& & \quad \exists V_1,V_1',V_2,V_2'. \\
& & \qquad V = \langle V_1,V_2\rangle \text{ and } V' = \langle V_1',V_2'\rangle \text{ and} \\
& & \qquad (V_1,V_1',\pi_1 P) \in [\![\sigma_1]\!]_\eta \text{ and} \\
& & \qquad (V_2,V_2',\pi_2 P) \\
& & \qquad\quad \in [\![\sigma_2]\!]_{\eta,\alpha\mapsto(\mathsf{Fst}(V_1),\mathsf{Fst}(V_1'),\pi_1 P)}\} \\
[\![\sigma]\!]^{\mathsf{pev}}_\eta & \stackrel{\text{def}}{=} & \{(M,M',Q) \mid \\
& & \quad \vdash_{\mathsf{P}} M,M' : \eta_{\mathsf{L},\mathsf{R}}(\sigma) \text{ and } M{\downarrow} \Leftrightarrow M'{\downarrow} \text{ and} \\
& & \quad \forall V,V'.\, (M \mapsto^* V) \Rightarrow (M' \mapsto^* V') \\
& & \qquad\qquad \Rightarrow (V,V',Q) \in [\![\sigma]\!]_\eta\} \\
[\![\sigma]\!]^{\mathsf{i}}_\eta & \stackrel{\text{def}}{=} & \{(V,V') \mid \exists Q.\, (V,V',Q) \in [\![\sigma]\!]_\eta\}^{\mathsf{st}}
\end{array}
$$

Figure 6: Logical Interpretation (Signatures)

Kinds A candidate for kind T can be any Pitts-closed relation. However, as one might expect, at a singleton kind $\mathsf{S}(c)$ the only candidate is the interpretation of c. For technical reasons, it is convenient to require explicitly that that interpretation is Pitts closed, even though that requirement will be redundant in the end.

A candidate of Π kind must be functional in all three of its arguments. In particular, it must map definitionally equal constructors to semantically equivalent candidates. This is the reason that we require a notion of semantic equivalence at all. We cannot simply use set-theoretic identity for semantic equivalence, because candidates can be equivalent at one kind but not another, just as constructors can.

A candidate of Σ kind is a pair of candidates, and a candidate of 1 is trivial.

Type constructors As usual, the interpretation of a constructor variable is obtained by looking the variable up in the environment. The interpretations of the intro and elim forms for Π and Σ are straightforward computations, but note that the interpretation $[\![\lambda\alpha{:}k.c]\!]_\eta$ is a partial function defined only for arguments that belong to $[\![k]\!]_\eta$.

The interpretations of unit, arrow, and product types are standard. The interpretation of universal and existential types are also conventional, given all the foregoing infrastructure for higher kinds. However, note that the interpretation of existentials—unlike other types—explicitly invokes st closure. Without st closure, it would not respect

contextual equivalence (see Section 6).

It is interesting to note that applying st closure in existentials produces an interpretation similar to the well-known encoding of existentials using universals (*i.e.,* $\exists\alpha.\tau \simeq \forall\beta.(\forall\alpha.\tau \to \beta) \to \beta$). In each case, two packages are equivalent exactly when they cannot be distinguished by certain continuations. In the former case, it is the continuations in R^{s} (where R is the interpretation of existentials prior to st closure); in the latter it is continuations in $[\![\forall\alpha.\tau \to \beta]\!]$. But the latter is very similar to a curried version of the former.

Signatures The main novelty of this work is the logical equivalence of modules, which arises from the interpretation of signatures. There are two main issues arising in modules that do not arise from terms: the fact that modules are mixed-phase objects, and the purity/impurity distinction.

To address the mixed-phase issue, the form of the logical interpretation of signatures is a set of triples (V, V', Q). This is a hybrid between the interpretations of kinds and terms: The static portions of V and V' are related by the candidate Q, while the dynamic portions are related in a manner similar to terms.

The interpretation uses triples as opposed to quadruples, because, unlike for kinds, we do not need the interpretation to serve as a PER. Although we do need to check in the Π^{ap} case that function candidates are functional (as discussed above), the kind interpretation can serve for that purpose.

For pure (transparent) modules, the signature interpretation is employed more or less directly, but for impure (sealed) modules we hide the candidate—reflecting the abstraction—and also apply st closure.

The definition is given in Figure 6. It employs two auxiliary definitions: pev performs evaluation closure while preserving the candidate; i hides the candidate and applies st closure.

Contexts We give two logical interpretation of contexts. The first ($[\![\Gamma]\!]$) is the static interpretation; it ignores terms and modules and defines a PER on substitutions. The second ($[\![\Gamma]\!]^{\mathsf{full}}$) is the full interpretation; it considers both static and dynamic expressions and defines a set of substitutions. Note that if $\eta \in [\![\Gamma]\!]^{\mathsf{full}}$ then $(\eta, \eta) \in [\![\Gamma]\!]$.

Definition 5.2

- We say that (η, η') belongs to $[\![\Gamma]\!]$ if whenever $\Gamma(\alpha) = k$ there exists $\eta(\alpha) = (c_1, c_2, Q)$ and $\eta'(\alpha) = (c_1', c_2', Q')$ such that $\vdash c_1 \equiv c_1' : \eta_{\mathsf{L}}(k)$ and $\vdash c_2 \equiv c_2' : \eta_{\mathsf{R}}(k)$ and $(c_1, c_2, Q, Q') \in [\![k]\!]_\eta$.
- We say that η belongs to $[\![\Gamma]\!]^{\mathsf{full}}$ if $(\eta, \eta) \in [\![\Gamma]\!]$, and additionally:
  - For all $x{:}\tau \in \Gamma$ there exists $\eta(x) = (v_1, v_2)$ such that $(v_1, v_2) \in [\![\tau]\!]_\eta$.
  - For all $\alpha/m{:}\sigma \in \Gamma$ there exists $\eta(m) = (V_1, V_2)$ and $\eta(\alpha) = (\mathsf{Fst}(V_1), \mathsf{Fst}(V_2), Q)$ such that $(V_1, V_2, Q) \in [\![\sigma]\!]_\eta$.

The static interpretation is used in several lemmas about the semantic model. The full interpretation is used in the definition of logical equivalence, which we can now give:

Definition 5.3 (Logical equivalence)

- Terms e and e' are *logically equivalent* at τ in Γ (written $\Gamma \vdash e \Leftrightarrow e' : \tau$) if $\vdash \Gamma\ \mathsf{ok}$ implies:
  - $\Gamma \vdash e, e' : \tau$, and
  - for all $\eta \in [\![\Gamma]\!]^{\mathsf{full}}$, $(\eta_{\mathsf{L}}(e), \eta_{\mathsf{R}}(e')) \in [\![\tau]\!]^{\mathsf{ev}}_\eta$.
- Modules M and M' are *impurely logically equivalent* at σ in Γ (written $\Gamma \vdash_{\mathsf{I}} M \Leftrightarrow M' : \sigma$) if $\vdash \Gamma\ \mathsf{ok}$ implies:
  - $\Gamma \vdash_{\mathsf{I}} M, M' : \sigma$, and
  - for all $\eta \in [\![\Gamma]\!]^{\mathsf{full}}$, $(\eta_{\mathsf{L}}(M), \eta_{\mathsf{R}}(M')) \in [\![\sigma]\!]^{\mathsf{iev}}_\eta$.
- Modules M and M' are *purely logically equivalent* at σ in Γ (written $\Gamma \vdash_{\mathsf{P}} M \Leftrightarrow M' : \sigma$) if $\vdash \Gamma\ \mathsf{ok}$ implies:
  - $\Gamma \vdash_{\mathsf{P}} M, M' : \sigma$, and
  - $\Gamma \vdash \mathsf{Fst}(M) \gg c$ and $\Gamma \vdash \mathsf{Fst}(M') \gg c'$ and $\Gamma \vdash c \equiv c' : \mathsf{Fst}(\sigma)$, and
  - for all $\eta \in [\![\Gamma]\!]^{\mathsf{full}}$, $(\eta_{\mathsf{L}}(M), \eta_{\mathsf{R}}(M'), [\![c]\!]_\eta) \in [\![\sigma]\!]^{\mathsf{pev}}_\eta$.

Observe that logical equivalence is an indexed dynamic relation, like contextual equivalence. We will show that they coincide.

5.2 Model Regularity

Our proof begins with the usual substitution lemma:

Lemma 5.4 (Substitution) *Suppose E is a kind, constructor, or signature. Then $[\![E]\!]_{\eta, \alpha \mapsto (\eta_{\mathsf{L}}(c), \eta_{\mathsf{R}}(c), [\![c]\!]_\eta)} = [\![[c/\alpha]E]\!]_\eta$.*

Next we wish to show that the kind interpretation works properly. For this, we prove a technical property we call model regularity. It says that, for all kinds up to a given kind, the kind interpretation is functional with respect to the environment, symmetric, and transitive.

Definition 5.5 We say *the model is regular up to k in context Γ* (written $\Gamma \models k$) if, for all $(\eta, \eta') \in [\![\Gamma]\!]$:

- $\Gamma \vdash k : \mathsf{kind}$
- $[\![k]\!]_\eta = [\![k]\!]_{\eta'}$
- For all $(c, c', Q_1, Q_2) \in [\![k]\!]_\eta$, we have $(c, c', Q_2, Q_1) \in [\![k]\!]_\eta$.
- For all $(c, c', Q_1, Q_2) \in [\![k]\!]_\eta$ and $(c, c', Q_2, Q_3) \in [\![k]\!]_\eta$, we have $(c, c', Q_1, Q_3) \in [\![k]\!]_\eta$.
- If k has the form $\Pi\alpha{:}k_1.k_2$ or $\Sigma\alpha{:}k_1.k_2$ then $\Gamma \models k_1$ and $\Gamma, \alpha{:}k_1 \models k_2$.

Definition 5.6 *We say* the model is regular up to Γ*, (written $\models \Gamma$) if for every $\alpha \in \mathrm{Dom}(\Gamma)$, $\Gamma \models \Gamma(\alpha)$.*

Model regularity is sufficient condition for the context interpretation to function as a PER:

Lemma 5.7 *If $\models \Gamma$ then $[\![\Gamma]\!]$ is symmetric and transitive.*

Model regularity respects the kind formation rules:

Lemma 5.8 *Suppose* $\models \Gamma$. *Then:*

- $\Gamma \models \mathsf{T}$ *and* $\Gamma \models 1$.
- *If* $\Gamma \vdash c : \mathsf{T}$ *and for all* $(\eta, \eta') \in [\![\Gamma]\!]$, $[\![c]\!]_\eta = [\![c]\!]_{\eta'}$, *then* $\Gamma \models \mathsf{S}(c)$.
- *If* $\Gamma \models k_1$ *and* $\Gamma, \alpha{:}k_1 \models k_2$ *then* $\Gamma \models \Pi\alpha{:}k_1.k_2$ *and* $\Gamma \models \Sigma\alpha{:}k_1.k_2$.

The main induction for kinds and constructors shows simultaneously that the model is regular up to the relevant kind, and that it respects the judgements in an appropriate way:

Lemma 5.9 *Suppose* $\models \Gamma$ *and* $(\eta, \eta') \in [\![\Gamma]\!]$. *Then:*

1. *If* $\Gamma \vdash k : \mathsf{kind}$ *then* $\Gamma \models k$.
2. *If* $\Gamma \vdash k_1 \equiv k_2 : \mathsf{kind}$ *then* $\Gamma \models k_1, k_2$ *and* $[\![k_1]\!]_\eta = [\![k_2]\!]_\eta$.
3. *If* $\Gamma \vdash k_1 \leq k_2 : \mathsf{kind}$ *then* $\Gamma \models k_1, k_2$ *and* $[\![k_1]\!]_\eta \subseteq [\![k_2]\!]_\eta$.
4. *If* $\Gamma \vdash c : k$ *then* $\Gamma \models k$ *and* $(\eta_{\mathsf{L}}(c), \eta_{\mathsf{R}}(c), [\![c]\!]_\eta, [\![c]\!]_{\eta'}) \in [\![k]\!]_\eta$.
5. *If* $\Gamma \vdash c \equiv c' : k$ *then* $\Gamma \models k$ *and* $(\eta_{\mathsf{L}}(c), \eta_{\mathsf{R}}(c), [\![c]\!]_\eta, [\![c']\!]_{\eta'}) \in [\![k]\!]_\eta$.

Proof Sketch

By induction on the derivation.

At this point, model regularity is proved for all well-formed contexts and kinds. Next we prove that the model respects the signature judgements:

Lemma 5.10 *Suppose* $\vdash \Gamma\,\mathsf{ok}$ *and* $(\eta, \eta') \in [\![\Gamma]\!]$. *Then:*

1. *If* $\Gamma \vdash \sigma : \mathsf{sig}$ *then* $[\![\sigma]\!]_\eta = [\![\sigma]\!]_{\eta'}$. *If, in addition,* $(V, V', Q) \in [\![\sigma]\!]_\eta$, *then* $(\mathsf{Fst}(V), \mathsf{Fst}(V'), Q, Q) \in [\![\mathsf{Fst}(\sigma)]\!]_\eta$.
2. *If* $\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}$ *then* $[\![\sigma]\!]_\eta = [\![\sigma']\!]_\eta$.
3. *If* $\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}$ *then* $[\![\sigma]\!]_\eta \subseteq [\![\sigma']\!]_\eta$.

Proof Sketch

By induction on the derivation.

5.3 The Fundamental Theorem

The logical interpretation is designed to respect contextual equivalence. For terms we have this already: the interpretation of any type belongs to $[\![\mathsf{T}]\!]$, and is therefore Pitts closed, and therefore respects contextual equivalence. However, the interpretation of a signature is not necessarily closed, so we need an additional lemma:

Lemma 5.11 *If* $\vdash \Gamma\,\mathsf{ok}$ *and* $\Gamma \vdash \sigma : \mathsf{sig}$ *and* $(\eta, \eta) \in [\![\Gamma]\!]$ *and* $(V_1, V_2, Q) \in [\![\sigma]\!]_\eta$ *and* $\vdash_{\mathsf{P}} V_1 \approx V_1' : \eta_{\mathsf{L}}(\sigma)$ *and* $\vdash_{\mathsf{P}} V_2 \approx V_2' : \eta_{\mathsf{R}}(\sigma)$ *then* $(V_1', V_2', Q) \in [\![\sigma]\!]_\eta$.

From this it follows that the logical equivalence respects contextual equivalence:

Corollary 5.12

- *If* $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau$ *and* $\Gamma \vdash e_1 \approx e_1' : \tau$ *and* $\Gamma \vdash e_2 \approx e_2' : \tau$ *then* $\Gamma \vdash e_1' \Leftrightarrow e_2' : \tau$.
- *If* $\Gamma \vdash_\kappa M_1 \Leftrightarrow M_2 : \sigma$ *and* $\Gamma \vdash_\kappa M_1 \approx M_1' : \sigma$ *and* $\Gamma \vdash_\kappa M_2 \approx M_2' : \sigma$ *then* $\Gamma \vdash M_1' \Leftrightarrow M_2' : \sigma$.

Proof Sketch

Observe that $[\![\tau]\!]^{\mathsf{ev}}_\eta$, $[\![\sigma]\!]^{\mathsf{iev}}_\eta$ and $[\![\sigma]\!]^{\mathsf{pev}}_\eta$ all respect contextual equivalence; the first because $[\![\tau]\!]_\eta$ is Pitts closed, the second because iev explicitly applies st closure, and the third because of Lemma 5.11. The result is then immediate, using substitutivity of contextual equivalence.

Now we can prove our main result. We wish to establish (for Lemma 3.7) that logical equivalence is a consistent quasicongruence, and its transpose is also consistent. Consistency is obvious because logical equivalence is defined using evaluation closure. It is also not hard to show it is substitutive. Thus, the fundamental theorem states:

Theorem 5.13 *Logical equivalence is compatible and reflexive.*

Proof Sketch

We prove compatibility by analysis of the cases that define it. (This looks almost exactly like an inductive proof.) Then we prove reflexivity by induction over the typing derivation $\Gamma \vdash e : \tau$ or $\Gamma \vdash_\kappa M : \sigma$. Most of the cases follow immediate from compatibility. The few remaining cases are those typing rules without corresponding compatibility rules: variables, unit, and extensionality.

Corollary 5.14 *Logical and contextual equivalence coincide.*

Proof

As above, logical equivalence is a consistent quasicongruence, and its transpose is also consistent. Thus logical equivalence is contained in contextual equivalence. For the converse, suppose $\Gamma \vdash e \approx e' : \tau$. Then $\Gamma \vdash e : \tau$. By reflexivity, $\Gamma \vdash e \Leftrightarrow e : \tau$. By Corollary 5.12, $\Gamma \vdash e \Leftrightarrow e' : \tau$. The case for modules is similar.

6 Discussion

To simplify our examples, let us suppose we have a few base types. Consider the signature $\sigma = \Sigma\alpha{:}(\!|\,\mathsf{T}\,|\!).\, (\!|\,\alpha\,|\!) \times (\!|\,\alpha \to \mathsf{bool}\,|\!)$ and three modules belonging to it:

$$\begin{array}{lcl} M_1 & = & \langle (\!|\,\mathsf{bool}\,|\!), \langle (\!|\,\mathtt{true}\,|\!), (\!|\,\lambda x.x\,|\!)\rangle\rangle \\ M_2 & = & \langle (\!|\,\mathsf{int}\,|\!), \langle (\!|\,0\,|\!), (\!|\,\mathtt{isEven?}\,|\!)\rangle\rangle \\ M_3 & = & \langle (\!|\,\mathsf{int}\,|\!), \langle (\!|\,0\,|\!), (\!|\,\mathtt{isZero?}\,|\!)\rangle\rangle \end{array}$$

In what follows, it is useful to keep in mind that $\vdash \mathsf{Fst}(M_1) \gg \langle \mathsf{bool}, \langle\star,\star\rangle\rangle$ and $\vdash \mathsf{Fst}(M_2), \mathsf{Fst}(M_3) \gg \langle \mathsf{int}, \langle\star,\star\rangle\rangle$.

All three terms are contextually equivalent when considered as *impure* modules. We can prove this by showing that they are logically equivalent. To show M_1 and M_2 are equivalent, we use the relation R_{12} that relates true to even numbers. It is then easy to show that $(\mathsf{bool}, \mathsf{int}, R_{12}) \in [\![\mathsf{T}]\!]^{\mathsf{set}}$

and $(\texttt{true}, 0) \in [\![\alpha]\!]_{\alpha \mapsto (\mathsf{bool}, \mathsf{int}, R_{12})}$ and $(\lambda x.x, \texttt{isEven?}) \in [\![\alpha \to \mathsf{bool}]\!]_{\alpha \mapsto (\mathsf{bool}, \mathsf{int}, R_{12})}$. Consequently:

$$(M_1, M_2, \langle R_{12}, \langle \star, \star \rangle \rangle) \in [\![\sigma]\!]$$

(Note that the trailing $\langle \star, \star \rangle$ is the vestigial candidate for the dynamic fields.) It then follows that $(M_1, M_2) \in [\![\sigma]\!]^{\mathsf{iev}}$, so $\vdash_! M_1 \Leftrightarrow M_2 : \sigma$.

To show M_2 and M_3 are equivalent, we use the relation R_{23} that relates even numbers to zero. We can then similarly show that:

$$(M_2, M_3, \langle R_{23}, \langle \star, \star \rangle \rangle) \in [\![\sigma]\!]$$

Thus $\vdash_! M_2 \Leftrightarrow M_3 : \sigma$.

However, none of the three modules are contextually equivalent as *pure* modules. For example, they can all be distinguished by the syntactic context $\mathcal{C} = \mathsf{Ext}(\pi_2\pi_2[\,])\,10$, which projects out the third field and applies it to 10. Note that this relies on the hole being instantiated with a pure module. Observe that $\mathcal{C}[M_2]$ returns true, $\mathcal{C}[M_3]$ returns false, and $\mathcal{C}[M_1]$ isn't even well-typed.

In logical terms, we see that M_1 and M_2 do not have equivalent static portions ($\langle \mathsf{bool}, \langle \star, \star \rangle \rangle$ versus $\langle \mathsf{int}, \langle \star, \star \rangle \rangle$) so they cannot be purely logically equivalent.

The other case is more interesting: M_2 and M_3 do have equivalent (indeed, identical) static portions. They fail pure logical equivalence because the candidate that relates them is $\langle R_{23}, \langle \star, \star \rangle \rangle$, which is not the logical interpretation of their static portion, namely $[\![\langle \mathsf{int}, \langle \star, \star \rangle \rangle]\!] = \langle [\![\mathsf{int}]\!], \langle \star, \star \rangle \rangle$. The equivalence relies on a different relation than the full equivalence of ints.

This precisely captures our intuition: M_1, M_2, and M_3 are indistinguishable only when they are sealed (*i.e.,* impure). Unsealed, their type fields are exposed, making them easy to distinguish.

Another instructive example is suggested by Pitts [22, example 7.7.4]. Suppose we augment our language with an empty void type. Let:

$$\begin{array}{lcl}
\tau & = & (\alpha \to \mathsf{bool}) \to \mathsf{unit} \\
f_1 & = & \lambda g{:}(\mathsf{void} \to \mathsf{bool}).\, \bot \\
f_2 & = & \lambda g{:}(\mathsf{bool} \to \mathsf{bool}).\, \texttt{if}\ g\ \texttt{false}\ \texttt{then}\ \bot \\
 & & \qquad \texttt{else if}\ g\ \texttt{true}\ \texttt{then}\ \star\ \texttt{else}\ \bot \\
N_1 & = & \langle (\!|\, \mathsf{void} \,|\!), \langle\!|\, f_1 \,|\!\rangle \rangle \\
N_2 & = & \langle (\!|\, \mathsf{bool} \,|\!), \langle\!|\, f_2 \,|\!\rangle \rangle
\end{array}$$

With some difficulty, we can show that N_1 and N_2 are contextually equivalent (as impure modules) at $\Sigma\alpha{:}(\!|\,\mathsf{T}\,|\!).\langle\!|\,\tau\,|\!\rangle$. Informally, this is because any client of these modules must produce an argument $\alpha \to \mathsf{bool}$ without any knowledge of α, so the only possible arguments are the two constant functions and the everywhere divergent function. Both f_1 and f_2 agree on those arguments.

However, there exists no relation R such that $(f_1, f_2) \in [\![\tau]\!]_{\alpha \mapsto (\mathsf{void}, \mathsf{bool}, R)}$. Indeed, since void is empty, the only R to consider is the empty relation. We may observe that $(\lambda x.\bot, \lambda x.x) \in [\![\alpha \to \mathsf{bool}]\!]_{\alpha \mapsto (\mathsf{void}, \mathsf{bool}, \emptyset)}$. But $f_1(\lambda x.\bot)$ diverges while $f_2(\lambda x.x)$ halts.

This shows that the logical interpretation of signatures, by itself, is not complete for contextual equivalence. We require biorthogonality—or some other mechanism—to close it. For impure modules, we do exactly that. But for pure modules we need not do so. Lemma 5.11 shows that the logical interpretation of signatures does respect *pure* contextual equivalences. Here, N_1 and N_2 are contextually equivalent only as impure modules.

7 Formalization

All the results in this paper are formalized using Coq (version 8.4). We implemented binding using deBruijn indices and explicit substitutions. To make reasoning about relations cleaner, we used the axioms of functional and propositional extensionality.

The full development is about 63k lines, much of which is definitions and preliminaries. Of our main results, Section 3 takes about 6.5k lines, Section 4 takes about 4k lines, and Section 5 takes about 12k lines. (All these counts include comments and whitespace.)

8 Related Work

The most similar work to this is Leroy [13]. His main purpose is the development of a module calculus with transparent applicative functors, the calculus that now underlies the OCaml module language [14]. The main technical advance is to allow functor applications to be part of paths. (Paths are a syntactic class that play the role that pure modules play in our calculus.)

He also wishes to show that allowing functor application in paths does not weaken data abstraction. To do so, he posits an abstraction principle for his module calculus. The principle is plausible, but the details of the proof are not published. He then examines some corollaries showing that the language provides representation independence.

Our results differ in two important ways. First, our language, being more recent, is much more expressive: It supports both generative and applicative functors, while Leroy supports only the latter. We use a semantic notion of purity based on effects [5], which subsumes Leroy's syntactic condition. We provide the full singleton calculus [29], while Leroy supports no higher-order type constructors. Finally, we support recursion, which Leroy's system (but certainly not OCaml) omits.

Second, our dynamic semantics is based on structured operational semantics [24], while Leroy's is denotational. This allows Leroy to sidestep myriad syntactic details. As a very basic example, we consider $(\lambda x.e)v$ and $[v/x]e$ to be two different terms, but in a denotational setting they denote the same semantic object.

An alternative approach to module formalisms is proposed by Rossberg, *et al.* [27, 26], who suggest that the meaning of ML modules—both static and operational semantics—can be defined by elaboration into F_ω. They thus argue that the ML module system is as elementary as F_ω. In so doing, one could argue they inherit a theory of abstraction from the underlying F_ω.

However, for our purposes we find such an inherited account unsatisfying, as it relies on Rossberg, *et al.'s* implicit claim that their elaboration is a faithful interpretation of ML modules, which is not proven. It certainly seems to be true, in much the same way as ML modules have always seemed to be good for data abstraction. But to prove it rigorously, one would need an independent account of abstraction for the source language—such as the one we provide here—and then one could prove that their elaboration preserves abstraction.

9 Conclusion

This work represents a first step toward substantiating the data abstraction claims that researchers into module calculi have been making for years. An important next step is to extend the core language with support for state. Unlike other real-world features that could be added to the core language, there is an important nexus of interaction between modules and state: An important use for generative functors is to ensure that multiple instances of a abstract data type, each with distinct local state, do not share types (for example, symbol tables [5]).

Another direction is to develop an account of abstraction for *weak sealing* [5], which seems necessary for abstraction within applicative functors, and gives rise to an intermediate state between purity and impurity.

A Static Semantics

$\boxed{\vdash \Gamma\ \mathsf{ok}}$

$$\frac{}{\vdash \epsilon\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash k : \mathsf{kind}}{\vdash \Gamma, \alpha{:}k\ \mathsf{ok}}$$

$$\frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \tau : \mathsf{T}}{\vdash \Gamma, x{:}\tau\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash \sigma : \mathsf{sig}}{\vdash \Gamma, \alpha/m{:}\sigma\ \mathsf{ok}}$$

$\boxed{\Gamma \vdash k : \mathsf{kind}}$

$$\frac{}{\Gamma \vdash \mathsf{T} : \mathsf{kind}} \qquad \frac{\Gamma \vdash c : \mathsf{T}}{\Gamma \vdash \mathsf{S}(c) : \mathsf{kind}} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 : \mathsf{kind}}$$

$\boxed{\Gamma \vdash k \equiv k' : \mathsf{kind}}$

$$\frac{\Gamma \vdash k : \mathsf{kind}}{\Gamma \vdash k \equiv k : \mathsf{kind}} \qquad \frac{\Gamma \vdash k' \equiv k : \mathsf{kind}}{\Gamma \vdash k \equiv k' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma \vdash k' \equiv k'' : \mathsf{kind}}{\Gamma \vdash k \equiv k'' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash c \equiv c' : \mathsf{T}}{\Gamma \vdash \mathsf{S}(c) \equiv \mathsf{S}(c') : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \equiv k_2' : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 \equiv \Pi\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \equiv k_2' : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 \equiv \Sigma\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$\boxed{\Gamma \vdash k \leq k' : \mathsf{kind}}$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind}}{\Gamma \vdash k \leq k' : \mathsf{kind}} \qquad \frac{\Gamma \vdash k \leq k' : \mathsf{kind} \quad \Gamma \vdash k' \leq k'' : \mathsf{kind}}{\Gamma \vdash k \leq k'' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash c : \mathsf{T}}{\Gamma \vdash \mathsf{S}(c) \leq \mathsf{T} : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1' \leq k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1' \vdash k_2 \leq k_2' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \Pi\alpha{:}k_1.k_2 : \mathsf{kind} \leq \Pi\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$$\frac{\Gamma \vdash k_1 \leq k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash k_2 \leq k_2' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1' \vdash k_2' : \mathsf{kind}}{\Gamma \vdash \Sigma\alpha{:}k_1.k_2 \leq \Sigma\alpha{:}k_1'.k_2' : \mathsf{kind}}$$

$\boxed{\Gamma \vdash c : k}$

$$\frac{\Gamma(\alpha) = k}{\Gamma \vdash \alpha : k}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash c : k_2}{\Gamma \vdash \lambda\alpha{:}k_1.c : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c_1 : \Pi\alpha{:}k_1.k_2 \quad \Gamma \vdash c_2 : k_1}{\Gamma \vdash c_1 c_2 : [c_2/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : [c_1/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_1 c : k_1} \qquad \frac{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_2 c : [\pi_1 c/\alpha]k_2}$$

$$\frac{}{\Gamma \vdash \star : 1} \qquad \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{T}}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma \vdash \tau_2 : \mathsf{T}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{T}} \qquad \frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma \vdash \tau_2 : \mathsf{T}}{\Gamma \vdash \tau_1 \times \tau_2 : \mathsf{T}}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \forall\alpha{:}k.\tau : \mathsf{T}}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \exists\alpha{:}k.\tau : \mathsf{T}}$$

$$\frac{\Gamma \vdash c : \mathsf{T}}{\Gamma \vdash c : \mathsf{S}(c)}$$

$$\frac{\Gamma \vdash c : \Pi\alpha{:}k_1.k_2' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha : k_2}{\Gamma \vdash c : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash \pi_1 c : k_1 \quad \Gamma \vdash \pi_2 c : [\pi_1 c/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash c : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash c : k'}$$

$\boxed{\Gamma \vdash c \equiv c' : k}$

$$\frac{\Gamma \vdash c : k}{\Gamma \vdash c \equiv c : k} \qquad \frac{\Gamma \vdash c' \equiv c : k}{\Gamma \vdash c \equiv c' : k}$$

$$\frac{\Gamma \vdash c \equiv c' : k \quad \Gamma \vdash c' \equiv c'' : k}{\Gamma \vdash c \equiv c'' : k}$$

$$\frac{\Gamma \vdash k_1 \equiv k_1' : \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash c \equiv c' : k_2}{\Gamma \vdash \lambda\alpha{:}k_1.c \equiv \lambda\alpha{:}k_1'.c' : \Pi\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \Pi\alpha{:}k_1.k_2 \quad \Gamma \vdash c_2 \equiv c_2' : k_1}{\Gamma \vdash c_1 c_2 \equiv c_1' c_2' : [c_2/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : k_1 \quad \Gamma \vdash c_2 \equiv c_2' : [c_1/\alpha]k_2 \quad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle : \Sigma\alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_1 c \equiv \pi_1 c' : k_1}$$

$$\frac{\Gamma \vdash c \equiv c' : \Sigma\alpha{:}k_1.k_2}{\Gamma \vdash \pi_2 c \equiv \pi_2 c' : [\pi_1 c/\alpha]k_2}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \mathsf{T} \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \mathsf{T}}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \mathsf{T}}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \mathsf{T} \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \mathsf{T}}{\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \mathsf{T}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash \forall \alpha{:}k.\tau \equiv \forall \alpha{:}k'.\tau' : \mathsf{T}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash \exists \alpha{:}k.\tau \equiv \exists \alpha{:}k'.\tau' : \mathsf{T}}$$

$$\frac{\Gamma \vdash c \equiv c' : \mathsf{T}}{\Gamma \vdash c \equiv c' : \mathsf{S}(c)} \qquad \frac{\Gamma \vdash c : \mathsf{S}(c')}{\Gamma \vdash c \equiv c' : \mathsf{T}}$$

$$\frac{\Gamma \vdash c : \Pi \alpha{:}k_1.k_2' \quad \Gamma \vdash c' : \Pi \alpha{:}k_1.k_2'' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha \equiv c'\alpha : k_2}{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2' \quad \Gamma, \alpha{:}k_1 \vdash c\alpha \equiv c'\alpha : k_2}{\Gamma \vdash c \equiv c' : \Pi \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash \pi_1 c \equiv \pi_1 c' : k_1 \qquad \Gamma \vdash \pi_2 c \equiv \pi_2 c' : [\pi_1 c/\alpha]k_2 \qquad \Gamma, \alpha{:}k_1 \vdash k_2 : \mathsf{kind}}{\Gamma \vdash c \equiv c' : \Sigma \alpha{:}k_1.k_2}$$

$$\frac{\Gamma \vdash c : 1 \quad \Gamma \vdash c' : 1}{\Gamma \vdash c \equiv c' : 1}$$

$$\frac{\Gamma \vdash c \equiv c' : k \quad \Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash c \equiv c' : k'}$$

$$\frac{\Gamma, \alpha{:}k_1 \vdash c_2 : k_2 \quad \Gamma \vdash c_1 : k_1}{\Gamma \vdash (\lambda \alpha{:}k_1.c_2)c_1 \equiv [c_1/\alpha]c_2 : [c_1/\alpha]k_2}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_1\langle c_1, c_2\rangle \equiv c_1 : k_1}$$

$$\frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_2\langle c_1, c_2\rangle \equiv c_2 : k_2}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \star : \mathsf{unit}}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{T} \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 e : \tau_2}$$

$$\frac{\Gamma \vdash k : \mathsf{kind} \quad \Gamma, \alpha{:}k \vdash e : \tau}{\Gamma \vdash \Lambda \alpha{:}k.e : \forall \alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e : \forall \alpha{:}k.\tau \quad \Gamma \vdash c : k}{\Gamma \vdash e[c] : [c/\alpha]\tau}$$

$$\frac{\Gamma \vdash c : k \quad \Gamma \vdash e : [c/\alpha]\tau \quad \Gamma, \alpha{:}k \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{pack}\,[c, e]\,\mathsf{as}\,\exists \alpha{:}k.\tau : \exists \alpha{:}k.\tau}$$

$$\frac{\Gamma \vdash e_1 : \exists \alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash e_2 : \tau' \quad \Gamma \vdash \tau' : \mathsf{T}}{\Gamma \vdash \mathsf{unpack}\,[\alpha, x] = e_1\,\mathsf{in}\,e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : (\mathsf{unit} \to \tau) \to \tau}{\Gamma \vdash \mathsf{fix}_\tau\, e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1\,\mathsf{in}\,e_2 : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma \quad \Gamma, \alpha/m{:}\sigma \vdash e : \tau \quad \Gamma \vdash \tau : \mathsf{T}}{\Gamma \vdash \mathsf{let}\,\alpha/m = M\,\mathsf{in}\,e : \tau}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \langle\!|\, \tau \,|\!\rangle}{\Gamma \vdash \mathsf{Ext}\, M : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash \sigma : \mathsf{sig}}$

$$\frac{}{\Gamma \vdash 1 : \mathsf{sig}} \qquad \frac{\Gamma \vdash k : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) : \mathsf{sig}} \qquad \frac{\Gamma \vdash \tau : \mathsf{T}}{\Gamma \vdash \langle\!|\, \tau \,|\!\rangle : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{gn}} \alpha{:}\sigma_1.\sigma_2 : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{ap}} \alpha{:}\sigma_1.\sigma_2 : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 : \mathsf{sig}}$$

$\boxed{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}$

$$\frac{\Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash \sigma \equiv \sigma : \mathsf{sig}} \qquad \frac{\Gamma \vdash \sigma' \equiv \sigma : \mathsf{sig}}{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig} \quad \Gamma \vdash \sigma' \equiv \sigma'' : \mathsf{sig}}{\Gamma \vdash \sigma \equiv \sigma'' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash k \equiv k' : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) \equiv (\!|\, k' \,|\!) : \mathsf{sig}} \qquad \frac{\Gamma \vdash \tau \equiv \tau' : \mathsf{T}}{\Gamma \vdash \langle\!|\, \tau \,|\!\rangle \equiv \langle\!|\, \tau' \,|\!\rangle : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \equiv \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{gn}} \alpha{:}\sigma_1.\sigma_2 \equiv \Pi^{\mathsf{gn}} \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \equiv \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{ap}} \alpha{:}\sigma_1.\sigma_2 \equiv \Pi^{\mathsf{ap}} \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' : \mathsf{sig} \quad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \equiv \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 \equiv \Sigma \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$\boxed{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}$

$$\frac{\Gamma \vdash \sigma \equiv \sigma' : \mathsf{sig}}{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma \leq \sigma' : \mathsf{sig} \quad \Gamma \vdash \sigma' \leq \sigma'' : \mathsf{sig}}{\Gamma \vdash \sigma \leq \sigma'' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash k \leq k' : \mathsf{kind}}{\Gamma \vdash (\!|\, k \,|\!) \leq (\!|\, k \,|\!) : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_1 : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \leq \sigma_2' : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{gn}} \alpha{:}\sigma_1.\sigma_2 \leq \Pi^{\mathsf{gn}} \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_1 : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2 \leq \sigma_2' : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 : \mathsf{sig}}{\Gamma \vdash \Pi^{\mathsf{ap}} \alpha{:}\sigma_1.\sigma_2 \leq \Pi^{\mathsf{ap}} \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$$\frac{\Gamma \vdash \sigma_1 \leq \sigma_1' : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1) \vdash \sigma_2 \leq \sigma_2' : \mathsf{sig} \qquad \Gamma, \alpha{:}\,\mathsf{Fst}(\sigma_1') \vdash \sigma_2' : \mathsf{sig}}{\Gamma \vdash \Sigma \alpha{:}\sigma_1.\sigma_2 \leq \Sigma \alpha{:}\sigma_1'.\sigma_2' : \mathsf{sig}}$$

$\boxed{\Gamma \vdash_\kappa M : \sigma}$

$$\frac{\Gamma(m) = \sigma}{\Gamma \vdash_{\mathsf{P}} m : \sigma} \qquad \frac{}{\Gamma \vdash_{\mathsf{P}} \star : 1}$$

$$\frac{\Gamma \vdash c : k}{\Gamma \vdash_{\mathsf{P}} (\!|\, c \,|\!) : (\!|\, k \,|\!)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{\mathsf{P}} \langle\!|\, e \,|\!\rangle : \langle\!|\, \tau \,|\!\rangle}$$

$$\frac{\Gamma \vdash \sigma : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma \vdash_{\mathsf{I}} M : \sigma'}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{gn}}\alpha/m{:}\sigma.M : \Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma'}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1 : \Pi^{\mathsf{gn}}\alpha{:}\sigma.\sigma' \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_{\mathsf{I}} M_1 M_2 : [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash \sigma : \mathsf{sig} \quad \Gamma, \alpha/m{:}\sigma \vdash_{\mathsf{P}} M : \sigma'}{\Gamma \vdash_{\mathsf{P}} \lambda^{\mathsf{ap}}\alpha/m{:}\sigma.M : \Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma'}$$

$$\frac{\Gamma \vdash_{\kappa} M_1 : \Pi^{\mathsf{ap}}\alpha{:}\sigma.\sigma' \quad \Gamma \vdash_{\mathsf{P}} M_2 : \sigma \quad \Gamma \vdash \mathsf{Fst}(M_2) \gg c_2}{\Gamma \vdash_{\kappa} M_1 \cdot M_2 : [c_2/\alpha]\sigma'}$$

$$\frac{\Gamma \vdash_{\kappa} M_1 : \sigma_1 \quad \Gamma \vdash_{\kappa} M_2 : \sigma_2 \quad \alpha \notin FV(\sigma_2)}{\Gamma \vdash_{\kappa} \langle M_1, M_2 \rangle : \Sigma\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Sigma\alpha{:}\sigma_1.\sigma_2}{\Gamma \vdash_{\mathsf{P}} \pi_1 M : \sigma_1}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Sigma\alpha{:}\sigma_1.\sigma_2 \quad \Gamma \vdash \mathsf{Fst}(M) \gg c}{\Gamma \vdash_{\mathsf{P}} \pi_2 M : [\pi_1 c/\alpha]\sigma_2}$$

$$\frac{\Gamma \vdash e : \exists\alpha{:}k.\tau \quad \Gamma, \alpha{:}k, x{:}\tau \vdash_{\mathsf{I}} M : \sigma \quad \Gamma \vdash \sigma : \mathsf{sig}}{\Gamma \vdash_{\mathsf{I}} \mathsf{unpack}\,[\alpha, x] = e\,\mathsf{in}\,(M : \sigma) : \sigma}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash_{\kappa} M : \sigma}{\Gamma \vdash_{\kappa} \mathsf{let}\, x = e\,\mathsf{in}\, M : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M_1 : \sigma \quad \Gamma, \alpha/m{:}\sigma \vdash_{\mathsf{I}} M_2 : \sigma' \quad \Gamma \vdash \sigma' : \mathsf{sig}}{\Gamma \vdash_{\mathsf{I}} \mathsf{let}\,\alpha/m = M_1\,\mathsf{in}\,(M_2 : \sigma') : \sigma'}$$

$$\frac{\Gamma \vdash_{\mathsf{I}} M : \sigma}{\Gamma \vdash_{\mathsf{I}} (M :> \sigma) : \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : (\!| k' |\!) \quad \Gamma \vdash \mathsf{Fst}(M) \gg c \quad \Gamma \vdash c : k}{\Gamma \vdash_{\mathsf{P}} M : (\!| k |\!)}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2' \quad \Gamma, \alpha/m{:}\sigma_1 \vdash_{\mathsf{P}} M \cdot m : \sigma_2}{\Gamma \vdash_{\mathsf{P}} M : \Pi^{\mathsf{ap}}\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} \pi_1 M : \sigma_1 \quad \Gamma \vdash_{\mathsf{P}} \pi_2 M : \sigma_2 \quad \alpha \notin FV(\sigma_2)}{\Gamma \vdash_{\mathsf{P}} M : \Sigma\alpha{:}\sigma_1.\sigma_2}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma}{\Gamma \vdash_{\mathsf{I}} M : \sigma} \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma \quad \Gamma \vdash \sigma \leq \sigma' : \mathsf{sig}}{\Gamma \vdash_{\kappa} M : \sigma'}$$

References

[1] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *2008 ACM International Conference on Functional Programming*, Victoria, Canada, 2008.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation, Systems Research Center, November 1989.

[3] Karl Crary. A simple proof of call-by-value standardization. Technical Report CMU-CS-09-137, Carnegie Mellon University, School of Computer Science, 2009.

[4] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 2005.

[5] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 236–249, New Orleans, Louisiana, January 2003.

[6] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[7] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[8] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[9] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[10] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 2000. Extended version published as CMU technical report CMU-CS-97-147.

[11] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Thirty-Fourth ACM Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[12] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[13] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[14] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 4.03, Documentation and user's manual*. Institut National de Recherche en Informatique et Automatique (INRIA), 2016.

[15] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[16] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1994.

[17] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[18] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[19] John C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[20] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[21] Luca Paolini and Simona Ronchi Della Rocca. Parametric parameter passing lambda-calculus. *Information and Computation*, 189(1):87–106, 2004.

[22] Andrew Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

[23] Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[24] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[25] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[26] Andreas Rossberg. 1ML — core and modules united. In *2015 ACM International Conference on Functional Programming*, Vancouver, Canada, 2015.

[27] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5), September 2014.

[28] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, March 1998.

[29] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4), October 2006. An earlier version appeared in the 2000 Symposium on Principles of Programming Languages.

[30] Philip Wadler. Theorems for free! In *Fourth Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[31] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer, 1983.

Revision 1.1.