Towards a Simpler Account of Modules and Generativity: Abstract Types have *Open* Existential Types.

Revised version, March 2008

Benoît Montagu
INRIA
Benoit.Montagu@inria.fr

Didier Rémy
INRIA
Didier.Remy@inria.fr

Abstract

We present a variant of the explicitly-typed second-order polymorphic λ-calculus with primitive open existential types*,* i.e. *a collection of more atomic constructs for introduction and elimination of existential types. We equip the language with a call-by-value small-step reduction semantics that enjoys the subject reduction property. Traditional* closed *existential types can be defined as syntactic sugar. Conversely, programs with no free existential types can be rearranged to only use closed existential types, but the translation is not compositional. We argue that open existential types model abstract types and modules with generativity. We also introduce a new notion of paths at the level of types instead of terms that allows for writing type annotations more concisely and modularly.*

Introduction

Modularity has always been the key to robust, manageable, and maintainable large software. Modular programming requires good discipline from programmers but also good support from programming languages. Unsurprisingly, module systems and type systems for modules have been an area of intensive research in the programming language community for more than two decades.

The module system for the language ML first proposed by MacQueen [12] in the mid 80's and independently improved and simplified in the mid 90's by Harper and Lillibridge [6] and by Leroy [9] is still the one in use in all dialects of ML, with relatively minor differences. Abstract types, higher-order functors, and sharing *a posteriori*, are key ingredients of its expressiveness and success.

However, there is still a discrepancy between the simplicity of the concepts necessary to understand and write simple modular programs on the one hand and on the other hand the complexity of the theoretical systems that have been proposed to give them a meaning and the heaviness of their use in some larger scale but realistic situations.

Furthermore, previous works have highlighted the difficulty to define a reduction semantics at the level of modules and circumvented the problem by giving a semantics via a non-trivial elaboration into an internal language, thus breaking the close correspondence between programs and logic.

In this paper, we present $\mathsf{F}^{\curlyvee}$ (read F-zip), a variant of System F with primitive *open existential types*, along with a call-by-value operational semantics, that allows to write programs more modularly, enjoys the *subject reduction property*, and whose underlying concepts are strongly and directly related to logic. More precisely, we decompose constructs for existentials into more atomic ones. We argue that this is the key to model abstract types and generativity.

Thus, we claim that the notion of *path* is useless in the sole purpose of type abstraction. However, we also claim that paths are useful to write *compact* programs. For this purpose, we introduce a new notion of *path* at the level of types, whereas it was usually defined at the level of terms.

Our present goal is not to increase the expressiveness of the module language, but instead to simplify the underlying concepts and to bridge the gap between the complexity of the state-of-the-art meta-theory of modules and the intuitive simplicity of the underlying mechanisms. We think this is a necessary step towards the design of a language with first-class modules that is conceptually economical yet more expressive and flexible.

The rest of the paper is organized as follows. We review previous approaches to abstract types in the next section. We present open existential types in §2 and formal results in §3 and §4. We describe our notion of paths in §5. We discuss related works in §6 before concluding remarks.

1. Previous approaches

Existing works on modules and type abstraction can be sorted in two categories. In earlier works, abstract types

were usually identified with existential types. However, it has been realized in the mid 80's that existential types do not adequately model type abstraction. Since then, abstract types have been considered as types whose definition has been forgotten and that are accessible through a *path*, *i.e.* modeled by strong sums and dependent types.

Our argument is that dependent types are actually too strong for modeling abstract types, making the meta-theory of modules unnecessarily involved. We instead argue that we can use existential types, once we "opened" them up.

We now review both approaches. For pedagogical purposes, we do not follow the chronological order.

1.1. Paths-based module systems

Paths are at the foundation of the majority of recent module systems [10, 18, 15]. They arise when a type is made abstract. Since the definition of the type has been forgotten, one cannot refer to it by *how* it is defined, instead one designates it by *where* it is defined: an abstract type is referred to as a projection path from a value variable, which makes types depend on values. In fact, types only depend on value variables, and therefore only require a decidable fragment of dependent types. However, this fragment is not stable under term substitution. This is a serious problem for the definition of a small-step reduction semantics. Different solutions have been proposed in the literature. Yet, none of them is quite satisfactory. A quick review follows.

Courant [2] designed a module system for the Coq proof assistant, together with a substitution semantics. To achieve this, he used the full power of dependent types along with the strong normalization property of Coq terms. This approach is not applicable to general purpose languages that allow non terminating programs.

Lillibridge [11] designed a module calculus in which paths are extended so that they are stable under value substitution. He managed to define a substitution semantics and to prove the soundness of his system, but at the expense of a substantial technical complexity.

Leroy [9] designed a module system and implemented it in the Objective Caml compiler [10], but did not give a direct semantics. Instead he only gave a *translation semantics* of his system, using an untyped λ-calculus with records as the target, and proved the soundness of his system.

In the context of the definition of a formal specification for SML [18], Dreyer *et al* [4] defined a module type system along with a typed internal language. They defined an indirect semantics of their module system through a global non trivial elaboration phase towards their internal language and proved the soundness of the internal language.

1.2. Abstract types as existential types

Much earlier, Mitchell and Plotkin [14] had shown that abstract types could be understood as existential types. However, they also noticed that existential types do not accurately model type abstraction in modules, especially the notion of generativity, and lack some modular properties.

In System F, existential types are introduced by the pack construct. Provided the term M has some type $\tau'\,[\alpha \leftarrow \tau]$, the expression $\mathsf{pack}\ \langle \tau, M\rangle\ \mathsf{as}\ \exists\alpha.\,\tau'$ hides the type information τ, called the *witness* of the existential, from the type of M so that the resulting type is $\exists\alpha.\,\tau'$.

$$\textsc{Pack}\ \frac{\Gamma \vdash M\ :\ \tau'\,[\alpha \leftarrow \tau]}{\Gamma \vdash \mathsf{pack}\ \langle \tau, M\rangle\ \mathsf{as}\ \exists\alpha.\,\tau'\ :\ \exists\alpha.\,\tau'}$$

Existential types are eliminated by the unpack construct: provided M has type $\exists\alpha.\,\tau$, the expression $\mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M'$ binds the type variable α to the witness of the existential and the value variable x to the *unpacked* term M in the body of M'. The resulting type is the one of M', in which α must not appear free. The reason for this restriction is that otherwise α, which is bound in M', would escape its scope.

$$\textsc{Unpack}\ \frac{\Gamma \vdash M\ :\ \exists\alpha.\,\tau \qquad \Gamma, \alpha, x : \tau \vdash M'\ :\ \tau' \qquad \alpha \notin \mathit{ftv}(\tau')}{\Gamma \vdash \mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M'\ :\ \tau'}$$

From now on we will consider System F is equipped with the above constructs, as they can be recovered as a well-known syntactic sugar.

2. Open existential types

Both of the pack and unpack constructs have modularity problems, but in different ways.

The problem with pack is *verbosity*: it requires to completely specify the resulting type, thus duplicating type information in the parts that have not been abstracted away. This duplication happens, for instance, when hiding the type of a single field of a large record, or maybe worse, when hiding some type information deeply inside a record. It is caused by the lack of separation between the introduction of an existential quantifier, and the description of which parts of the type must be made hidden.

The problem with unpack is *non-locality*: it imposes the same scope to the type variable α and the value variable x, which is emphasized by the non escaping condition on α. As a result, all uses of the unpacked term must be anticipated. In other words, the only way to make the variable α available in the whole program is to put unpack early enough in the program, which is a non local, hence non modular, program transformation. Again, the reason for it is that unpack is doing too many things at the same time.

In both cases, the lack of *modularity* is related to the lack of *atomicity* of the constructs. Therefore, we propose to split both of them into more atomic constructs, recovering modularity while preserving expressiveness of existen-

tial types.

2.1. Atomic constructs for existential types

To achieve the decomposition of existential types into more atomic constructs, we first need to enrich the context of typing judgments.

Richer contexts for typing judgments. The contexts of typing judgments in System F are sequences of items. An item is either a binding $x : \tau$ from a value variable to a type, which is introduced while typing functions, or a universal type variable $\forall\alpha$, which is introduced while typing polymorphic expressions.

We augment typing environments with two new items: existential type variables to keep track of the scope of (open) abstract types, and type definitions $\alpha = \tau$ to more concisely mediate between the abstract and concrete views of types. That is, typing environments are as follows:

$$\begin{array}{lll} \Gamma & ::= & \varepsilon \quad | \quad \Gamma, b \qquad\qquad\qquad\qquad\quad \text{(Environments)} \\ \theta & ::= & \alpha \\ b & ::= & x : \tau \quad | \quad \forall\theta \quad | \quad \theta = \tau \quad | \quad \exists\theta \quad \text{(Bindings)} \end{array}$$

Notice that θ is a binding occurrence in type-definitions, as $\forall$- and $\exists$-bindings. Wellformedness of typing environments will ensure that no variable is ever bound twice.

We shall see below that existential variables have to be treated linearly. The syntactic category θ is introduced to facilitate the extension of $\mathsf{F}^{\forall}$ with paths in §5.

Splitting pack. We replace pack with two orthogonal constructs: *existential introduction*, which creates an existential type, and *coercion*, which determines which parts of types are to be hidden.

The *existential introduction* $\exists(\alpha = \tau)\, M$ introduces an existential type variable α with witness τ (more precisely, the definition $\alpha = \tau$) in the environment while typing M, and makes α existentially bound in the resulting type.

$$\textsc{Exists}\ \frac{\Gamma, \alpha = \tau \vdash M : \tau'}{\Gamma \vdash \exists(\alpha = \tau)\, M : \exists\alpha.\, \tau'}$$

The *coercion* $(M : \tau)$ replaces the type of M with some *compatible* type τ. The compatibility relation $\approx$ is the smallest congruence that contains all type-definitions occurring in the environment. A coercion is typically employed to specify where some abstract types should be used instead of their witnesses in the typing of M.

$$\textsc{Coerce}\ \frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \approx \tau}{\Gamma \vdash (M : \tau) : \tau}$$

The expressiveness of pack is retained, since it can be provided as the following syntactic sugar:

$$\mathsf{pack}\ \langle \tau, M \rangle\ \mathsf{as}\ \exists\alpha.\, \tau' \ \triangleq\ \exists(\alpha = \tau)\, (M : \tau')$$

However, the description of what is being hidden can now be separated from the action of hiding, which avoids repetition of type information. Hence it makes the creation of existential values, shorter, thus easier, and more maintainable. Examples can be found in Appendix §A.1.

Splitting unpack. We also replace unpack with two orthogonal constructs, *opening* and *restriction*, that implement *scopeless unpacking* of existential values and *scope restriction* of abstract types, respectively.

The *opening* $\mathsf{open}^{\alpha}\ M$ expects M to have an existential type $\exists\alpha.\, \tau$ and *opens* it under the name α, which is *tracked* in the typing environment by the existential item $\exists\alpha$. The rule can also be read bottom-up, treating the item $\exists\alpha$ as a *linear* resource that is consumed by the opening.

$$\textsc{Open}\ \frac{\Gamma \vdash M : \exists\alpha.\, \tau \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \exists\alpha \vdash \mathsf{open}^{\alpha}\ M : \tau}$$

The *restriction* $\nu\alpha.\, M$ implements the non-escaping condition of Rule UNPACK. First, it requires α not to appear free in the type of M, thus enforcing a limited scope. Second, it provides an existential resource $\exists\alpha$ in the environment, that ought to be consumed by some $\mathsf{open}^{\alpha}\ M'$ expression occurring within M.

$$\textsc{Nu}\ \frac{\Gamma, \exists\alpha \vdash M : \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\, M : \tau}$$

Here again, we may provide unpack as syntactic sugar:

$$\mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M' \ \triangleq\ \nu\alpha.\ (\mathsf{let}\ x = \mathsf{open}^{\alpha}\ M\ \mathsf{in}\ M')$$

This makes explicit all the simultaneous operations performed by unpack: first, it defines a scope for the name α of the witness of the existential type of M; then, it opens M under the name α; finally, it binds the resulting value to x in the remaining expression M'.

The main flaw of unpack, *i.e.* the scope restriction for the abstract name, is essentially captured by the *restriction* construct. However, since the scope restriction has been separated from the unpack, it need not (always) be used anymore. The abstract type α may now be introduced at the outermost level or given by the typing context and be freely made available to the whole program. Interestingly, this closely models type abstraction in modules: the following program, written in an ML-like syntax, defines an abstract module of integers.

```
module X : sig type t val z : t val s : t → t end =
struct
  type t = int val z = 0 val s = λ(x : int) x+1
end
```

It provides the zero constant z and the successor function s. The type $X.t$ is abstract and available in the whole program.

Its counterpart in $\mathsf{F}^{\curlyvee}$ is defined hereafter:

$$\mathsf{open}^{\beta}\ \exists(\alpha = \mathit{int})\,(\{z = 0\ ;\ s = \lambda(x\!:\!\mathit{int})\,x{+}1\} :\ \{z : \alpha\ ;\ s : \alpha \to \alpha\})$$

The two pieces of code look similar, except for the fact that the signature ascription has been replaced with the opening of an existential value. The counterpart of the signature is the type in the coercion. Note that no type component, hence no name for the module, is needed: the counterpart of $X.t$ is the abstract type β, which is present in the typing context. It is available in the whole program and does not refer to a value variable.

Linearity to control openings. As openings use abstract names given by the environment, one must be careful to avoid "abstraction capture", as in the following (ill-typed) example.

$$\mathsf{let}\ f = \mathsf{open}^{\alpha}\ \exists(\beta = \mathit{int})\,(\lambda(z\!:\!\mathit{int})\,z + 1 : \beta \to \beta)\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{open}^{\alpha}\ \exists(\beta = \mathit{bool})\,(\mathit{true} : \beta)\ \mathsf{in}\ f\ x$$

Here, f and x result from two different openings under the same name α. Hence, f and x are assigned types $\alpha \to \alpha$ and α, respectively, using the *same* abstract name α. However, each branch uses a different witness for α (*int* in the case of f and *bool* in the case of x). This yields to the unsound application $f\ x$, which evaluates to $1 + \mathit{true}$.

To prevent abstraction capture, it suffices that *every name α be used in exactly one opening under α*. This may be achieved by treating the existential items of the typing environment in a *linear* way. Linearity can easily be enforced in typing rules by a *zipping* operation that describes how typing environments of the premises must be combined to form the one of the conclusion. Zipping is a binary operation $(\cdot \curlyvee \cdot)$ that proceeds by zipping individual bindings pointwise. For all items but existential type variables, zipping requires the two facing items to be identical, as usual. The interesting case is when one of the two items is an existential variable $\exists\alpha$: the (not final) intuition is that, in this case, the other item must be the universal variable $\forall\alpha$, hence the *zipper* image. This ensures that an existential variable in the conclusion can only be used up in one of the premises. Zipping can also be explained in terms of internal and external choice: the side that makes use of $\exists\alpha$ will make an internal choice by giving internally the witness. Therefore the other side *must* consider the choice of the witness as external. This is why it is given the item $\forall\alpha$.

The above idea is unfortunately too generous: it makes recursive types appear naturally, as will be explained in the next paragraph. The presence of two zipping operators $(\cdot \curlyvee \cdot)$ and $(\cdot \barwedge \cdot)$ will also be justified hereafter.

Avoiding the birth of recursive types. The decomposition of unpack into opening and restriction opens up the way to recursive types, because it allows to use an abstract type variable before its witness has been defined. Recursive types can appear through openings, *i.e.* through type abstraction, in two ways.

We call *internal recursivity* the first one. It is highlighted by the following example:

$$\mathsf{let}\ x = \exists(\alpha = \beta \to \beta)\,M\ \mathsf{in}\ \mathsf{open}^{\beta}\ x$$

The abstract type variable β is used in a witness to define x which is then opened under the name β. By reducing this expression we get:

$$\mathsf{open}^{\beta}\ \exists(\alpha = \beta \to \beta)\,M$$

This leads us to the recursive equation $\beta = \beta \to \beta$. We prevent this misuse by making the zipping non-symmetric $(\cdot \barwedge \cdot)$ for the let construct (see last case of Fig. 1): if an existential resource goes in the second branch of the let then the corresponding type variable cannot appear in the first one. This avoids the premature use of an abstract type variable in a let-binding.

We call *external recursivity* the second way, which is hereafter exemplified:

$$\{\ell_1 = \mathsf{open}^{\beta_1}\ \exists(\alpha_1 = \beta_2 \to \beta_2)\,M_1\ ;\ \ell_2 = \mathsf{open}^{\beta_2}\ \exists(\alpha_2 = \beta_1 \to \beta_1)\,M_2\}$$

The above code is a pair whose components have been abstracted away and the witnesses are mutually defined. If we remove the type abstractions we get the recursive equation system $\beta_1 = \beta_2 \to \beta_2$ and $\beta_2 = \beta_1 \to \beta_1$. Here again we also could make the zipping non-symmetric, but this would enforce a non-symmetric reduction strategy. Instead, we keep it symmetric (see of Fig. 1): if an existential resource goes in one branch then the corresponding type variable cannot appear in the other. Although this might look over-restrictive, it has the advantage of keeping the system independent from evaluation order. Moreover the system remains at least as expressive as System F:

$$\nu\alpha.\,C_2[\mathsf{let}\ x = C_1[\mathsf{open}^{\alpha}\ M_1]\ \mathsf{in}\ M_2]$$

If we consider this program as an unpack, then the contexts C_1 and C_2 are empty. Consequently, α cannot occur free in C_1 or C_2. This is almost what our restriction implements: α cannot occur in C_1 thanks to the restriction on *external* recursivity; α cannot occur in C_2 thanks to the restriction on *internal* recursivity, unless $\mathsf{open}^{\alpha}\ M_1$ is itself bound by a let.

A less restrictive system is currently being studied.

2.2. Syntax of $\mathsf{F}^{\curlyvee}$

The language $\mathsf{F}^{\curlyvee}$ is based on the explicitly typed version of System F with records and is extended with constructs of §2.1. Types and terms are described in Fig. 2.

As open existentials do not introduce new forms of types, types of $\mathsf{F}^{\curlyvee}$ are type variables, arrow types, record types, universal types, and existential types. The notation $(\ell_i : \tau_i)^{i\in 1..n}$ stands for a sequence of n pairs, each composed of

$$
\begin{array}{rcl}
\varepsilon \curlyvee \varepsilon & = & \varepsilon \\
(\Gamma_1, b) \curlyvee (\Gamma_2, b) & = & (\Gamma_1 \curlyvee \Gamma_2), b \text{ if } b \neq \exists\alpha \\
(\Gamma_1, \exists\alpha) \curlyvee \Gamma_2 & = & (\Gamma_1 \curlyvee \Gamma_2), \exists\alpha \text{ if } \alpha \notin \mathit{dom}\,\Gamma_2 \\
\Gamma_1 \curlyvee (\Gamma_2, \exists\alpha) & = & (\Gamma_1 \curlyvee \Gamma_2), \exists\alpha \text{ if } \alpha \notin \mathit{dom}\,\Gamma_1
\end{array}
$$

$$
\begin{array}{rcl}
\varepsilon \curlyvee' \varepsilon & = & \varepsilon \\
(\Gamma_1, b) \curlyvee' (\Gamma_2, b) & = & (\Gamma_1 \curlyvee' \Gamma_2), b \text{ if } b \neq \exists\alpha \\
(\Gamma_1, \exists\alpha) \curlyvee' \Gamma_2 & = & (\Gamma_1 \curlyvee' \Gamma_2), \exists\alpha \text{ if } \alpha \notin \mathit{dom}\,\Gamma_2 \\
\Gamma_1 \curlyvee' (\Gamma_2, \exists\alpha) & = & (\Gamma_1 \curlyvee' \Gamma_2), \exists\alpha \text{ if } \alpha \notin \mathit{dom}\,\Gamma_1 \\
(\Gamma_1, \exists\alpha) \curlyvee' (\Gamma_2, \forall\alpha) & = & (\Gamma_1 \curlyvee' \Gamma_2), \exists\alpha
\end{array}
$$

Figure 1. Zippings of two contexts.

$$
\begin{array}{rcl}
\tau ::= & \alpha \mid \tau \to \tau \mid \{(\ell_i : \tau_i)^{i\in 1..n}\} \\
& \mid \forall\theta.\,\tau \mid \exists\theta.\,\tau \\
M ::= & x \mid \lambda(x{:}\tau)\,M \mid M\ M \\
& \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid \Lambda\theta.\,M \mid M\ [\tau] \\
& \mid \exists(\theta = \tau)\,M \mid (M : \tau) \mid \nu\theta.\,M \\
& \mid \mathsf{open}^{\alpha}\ M \mid \{r\} \mid M.\ell \\
r ::= & \epsilon \mid \ell = M\ ;\ r \mid \ell : \tau = M\ ;\ r
\end{array}
$$

Figure 2. Types and terms.

a label and a type. Type and environment wellformedness are defined as usual (see Fig. 13 in the Appendix).

Terms of $\mathsf{F}^{\curlyvee}$ are variables, functions (whose arguments are explicitly typed), applications, let-bindings, type generalizations and applications, introductions and projections of records, and the four constructs for open existentials described before.

Record fields are pairs $\ell = M$ of a label name ℓ and a term M. The label name is used to access the field externally, as usual with records. In addition, we provide an optional type annotation on record fields, that behaves as a coercion. This construct[1] is theoretically unimportant, but practically quite useful on real large examples.

For conciseness, we also use the following syntactic sugar in technical developments:

$$\Sigma^{\beta}(\alpha = \tau)\,M \triangleq \mathsf{open}^{\beta}\ \exists(\alpha = \tau)\,M$$

This can be understood as the creation of an abstract type without seeing the intermediate existential type—which is a common programming pattern in modular programs.

2.3. Typing rules

Typing rules for open existentials have already been presented in §2.1. A few more selected typing rules are given on Fig. 3. The full type system can be found in Appendix.

[1] In fact, this construct is slightly more general than coercion: expressing $\{\ell_1 : \tau_1 = M_1; \ell_2 = M_2\}$ in terms of coercions requires the knowledge of the type of M_2. Conversely, the coercion $(M{:}\tau)$ can be replaced with $\{\ell{:}\tau{=}M\}.\ell$. The latter actually reduces to the former. Though coercions need not be exposed to the user, they must remain in the syntax of terms.

$$\textsc{Var}\ \frac{\Gamma \vdash \mathrm{wf} \quad \Gamma\ \mathit{pure}}{\Gamma \vdash x : \Gamma(x)} \qquad \textsc{App}\ \frac{\Gamma_1 \vdash M_1 : \tau_2 \to \tau \quad \Gamma_2 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2 : \tau}$$

$$\textsc{Let}\ \frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee' \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

$$\textsc{Shift}\ \frac{\Gamma' \vdash M : \tau \qquad \Gamma \Vdash \Gamma'}{\Gamma \vdash M : \tau}$$

Figure 3. More typing rules.

Rule SHIFT states that if a term is typable in an environment Γ then it is also typable in every stronger environment Γ', *i.e.* $\Gamma' \Vdash \Gamma$. Environment entailment is the smallest pre-order satisfying the following commutation:

$$\frac{\alpha \notin \mathit{ftv}(b)}{\Gamma_1, \exists\alpha, b, \Gamma_2 \Vdash \Gamma_1, b, \exists\alpha, \Gamma_2}$$

The *raison d'être* of Rule SHIFT is to shift existential items to the right of the environment (whenever this is sound): indeed Rule OPEN needs the resource it consumes to be the right-most item of the environment. Soundness of entailment is ensured by the natural interpretation of environments as logical formulæ.

The remaining typing rules are as in System F with two small differences described.

First, as mentioned above, typing rules with several typing judgments as premises use zipping instead of equality to relate their typing environments. This is exemplified by Rule APP for the symmetric zipping and by Rule LET for the non-symmetric one.

Typing rules must also ensure that values can be substituted without breaking linearity, which is the case when the typing environment does not contain any existential items. When Γ is such an environment, we say that Γ is pure and write $\Gamma\ \mathit{pure}$. This condition appears as an additional premise of typing rules of expressions that are also values as exemplified by Rule VAR. Purity will be used and explained in more details in §2.4.

We consider now the typing derivation of term $\Sigma^{\beta}(\alpha = \tau)\,M$: this term abstracts (the type definition for) α away in two steps: first, the equation is forgotten returning the existential value $\exists(\alpha = \tau)\,M$ of type $\exists\alpha.\,\tau'$, which is then opened under the name β. In a nutshell, type abstraction is the composition of information *erasure* and *generativity*.

$$\textsc{Open}\ \frac{\textsc{Exists}\ \dfrac{\Gamma, \alpha = \tau \vdash M : \tau'}{\Gamma \vdash \exists(\alpha = \tau)\,M : \exists\alpha.\,\tau'}}{\Gamma, \exists\beta \vdash \mathsf{open}^{\beta}\ \exists(\alpha = \tau)\,M : \tau'\,[\alpha \leftarrow \beta]}$$

$$
\begin{array}{rcll}
v & ::= & u \mid (u : \tau) & \text{(Values)} \\
u & ::= & x \mid \lambda(x{:}\tau)\, M & \text{(Pre-values)} \\
 & \mid & \Lambda\alpha.\, M \mid \exists(\alpha = \tau)\, v \mid \{r_v\} & \\
r_v & ::= & \epsilon \mid \ell : \tau^? = v\ ;\ r_v & \text{(Value-fields)} \\
w & ::= & v \mid \Sigma^\beta(\alpha = \tau)\, w & \text{(Results)} \\
E & ::= & [] \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid E\ M & \text{(Contexts)} \\
 & \mid & w\ E \mid E.\ell \mid \exists(\alpha = \tau)\, E & \\
 & \mid & \mathsf{open}^\alpha\ E \mid (E : \tau) \mid \nu\alpha.\, E &
\end{array}
$$

Figure 4. Values, results, evaluation contexts.

2.4. Semantics

The language $\mathsf{F}^\curlyvee$ is equipped with a small-step call-by-value reduction semantics. We begin with important remarks about substitutability, then define and explain values, and finally describe the reduction steps.

Substitution and purity. Some terms *cannot* be safely substituted, since substitution may violate the linear treatment of openings. It turns out that *pure* terms, *i.e.* terms that are typable in a pure environment, behave well with respect to substitution:

Lemma 1 (Substitution lemma)**.** *If* $\Gamma \vdash M : \tau$ *and* $\Gamma', x : \tau, \Gamma'' \vdash M' : \tau'$*, where* Γ *is pure and* $\Gamma \curlyvee \Gamma'$ *is well defined, then* $(\Gamma \curlyvee \Gamma'), \Gamma'' \vdash M'[x \leftarrow M] : \tau'$.

Therefore, values are substitutable if we restrict them to pure terms . Conversely, is every irreducible term also a pure term?

Results and values. Results are well-behaved irreducible terms. Results include values. In System F (as in many other languages) results actually coincide with values. However, this need not be the case. In $\mathsf{F}^\curlyvee$, results also include terms such as $\Sigma^\beta(\alpha = \tau)\, \lambda(x{:}\alpha)\, x$, which are well-behaved and cannot be further reduced, but are not values, as they are not pure and thus not substitutable.

More precisely, values are defined in Fig. 4. They are either pre-values or coerced pre-values, where pre-values are variables, functions, generalizations, existential values or records of values. In particular, nested coercions are not values—they must be further reduced. Notice that no evaluation takes place under λ's or Λ's. Finally, results are values preceded by a (possibly empty) sequence of Σ's.

The purity premises in some of the typing rules ensure that values are pure, hence by Lemma 1 substitutable.

Lemma 2 (Purity of values)**.** *If* $\Gamma \vdash v : \tau$*, then* Γ *pure.*

Extrusions. Values are substitutable, but some results are not values, namely a sequence of Σ's prefixing a value. How can we handle these results, when they ought to be substituted, without breaking linearity? Our solution is to extrude

$$
\begin{array}{l}
\quad \mathsf{let}\, x = \Sigma^\beta(\alpha = \mathit{int})\,(1{:}\alpha)\, \mathsf{in}\, \{\ell_1 = x\,;\, \ell_2 = (\lambda(y{:}\beta)\, y)\ x\} \\
\overset{1}{\rightsquigarrow} \Sigma^\beta(\alpha = \mathit{int}) \\
\quad \mathsf{let}\ x = (1 : \alpha)\ \mathsf{in}\ \{\ell_1 = x\ ;\ \ell_2 = (\lambda(y{:}\alpha)\, y)\ \ x\} \\
\overset{1}{\rightsquigarrow} \Sigma^\beta(\alpha = \mathit{int})\, \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = (\lambda(y{:}\alpha)\, y)\ \ (1 : \alpha)\} \\
\overset{1}{\rightsquigarrow} \Sigma^\beta(\alpha = \mathit{int})\, \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = (1 : \alpha)\}
\end{array}
$$

Figure 5. Example of extrusion.

$$
\begin{array}{rcll}
\{\ell = M\ ;\ r\} \uparrow \ell & = & M & \\
\{\ell : \tau = M\ ;\ r\} \uparrow \ell & = & (M : \tau) & \\
\{\ell' : \tau^? = M\ ;\ r\} \uparrow \ell & = & \{r\} \uparrow \ell & \text{if } \ell \neq \ell'
\end{array}
$$

Figure 6. Projection of a record.

the Σ's *just enough* to expose and perform the next reductions step.

For example, consider the reduction steps on Fig. 5. The initial expression is a let-binding of the form $\mathsf{let}\ x = w\ \mathsf{in}\ M$ where w is the result form $\Sigma^\beta(\alpha = \mathit{int})\,(1 : \alpha)$. Hence, the next expected reduction step is the substitution of w for x in M. However, since x occurs twice in M, this would duplicate the opening appearing in w breaking the linear use of β. The solution is to first *extrude* the Σ binding outside of the let-binding, so that the expression bound to x becomes the substitutable value form $(1 : \alpha)$. However, by enlarging the scope of Σ, we have put M in its scope, and as a consequence all occurrences of the external view β in M should be replaced with the internal view α. Then, we may perform let-reduction safely and further reduce the β-redex that has been created.

More generally, the reduction semantics will be set so that Σ can always be extruded out of redex forms.

Reduction. The semantics of $\mathsf{F}^\curlyvee$ is given by a call-by-value reduction strategy, described by a small-step reduction relation. We fix a left-to-right evaluation order so that the semantics is deterministic, although we could have left the order unspecified. By contrast having a call-by-value strategy and a weak-reduction is essential.

Evaluation contexts are described in Fig. 4. A one-step reduction is the application of a reduction rule in some evaluation context. The reduction relation is the transitive closure of the one-step reduction relation. Due to lack of space, we only describe a representative subset of the reduction rules in Fig. 7. The whole definition is available in the Appendix (Fig. 15). Reduction steps are sorted in three groups.

The main group of rules describe the contraction of redexes. The β-reduction and the reduction of type applications are as usual. Record projection (Rule REDEX-PROJ) is as usual, except type annotations must be kept during projection, as described in the definition of the projection function in Fig. 6. The last form of this group is the contraction of abstract types described by rules NU-SIGMA1 and NU-SIGMA2, which is used to simplify terms of the

$$(\lambda(x{:}\tau)\,M)\ v \ \rightsquigarrow\ \mathsf{let}\ x = v\ \mathsf{in}\ M \qquad \text{(REDEX-APP)}$$
$$\mathsf{let}\ x = v\ \mathsf{in}\ M \ \rightsquigarrow\ M\,[x \leftarrow v] \qquad \text{(REDEX-LET)}$$
$$\{r_v\}.\ell \ \rightsquigarrow\ \{r_v\} \uparrow \ell \qquad \text{(REDEX-PROJ)}$$
$$\nu\beta.\,\Sigma^{\beta}(\alpha = \tau)\,v \ \rightsquigarrow\ \nu\beta.\,\Sigma^{\beta}(\alpha = \tau)\,v\,[\alpha \leftarrow \tau] \ \text{if}\ \alpha \in \mathit{ftv}(v) \qquad \text{(NU-SIGMA1)}$$
$$\nu\beta.\,\Sigma^{\beta}(\alpha = \tau)\,v \ \rightsquigarrow\ v \ \text{if}\ \alpha \notin \mathit{ftv}(v) \qquad \text{(NU-SIGMA2)}$$
$$(\Sigma^{\beta}(\alpha = \tau)\,w_1)\ w_2 \ \rightsquigarrow\ \Sigma^{\beta}(\alpha = \tau)\ (w_1\ (w_2\,[\beta \leftarrow \alpha])) \ \text{if}\ \alpha \notin \mathit{ftv}(w_2) \qquad \text{(SIGMA-APP-LEFT)}$$
$$(\lambda(x{:}\tau)\,M : \tau_1 \to \tau_2)\ v \ \rightsquigarrow\ ((\lambda(x{:}\tau)\,M)\ (v : \tau) : \tau_2) \qquad \text{(COERCE-APP)}$$
$$((u : \tau) : \tau') \ \rightsquigarrow\ (u : \tau') \qquad \text{(COERCE-COERCE)}$$

Figure 7. Reduction steps (selected rules).

form $\nu\beta.\,\Sigma^{\beta}(\alpha = \tau)\,M$ by substituting the witness τ for α inside M. This is similar to the usual rule for contracting unpack-ing of pack's where the same substitution occurs.

The next group of rules implement the extrusion of Σ's, in all cases where a Σ comes up against a redex, as illustrated by rule SIGMA-APP-LEFT. As explained above, the internal view α must be substituted for the external one β in terms that become in scope of α during the extrusion.

The last group of reduction rules keep track of coercions during reduction, as exemplified by Rule COERCE-APP. Notice that nested coercions are ignored, the outer one taking priority (Rule COERCE-COERCE).

3. Type soundness

Type soundness results from the combination of the subject reduction and progress properties.

The subject reduction proof is, as usual, mainly built on the substitution lemma (Lemma 1). The proof itself is not really informative, but it is particularly interesting that the proof is absolutely standard and almost straightforward, as this is not usually the case for other approaches to modules.

Proposition 1 (Subject reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \rightsquigarrow M'$*, then* $\Gamma \vdash M' : \tau$.

This property is proved by induction on the reduction. In order to simplify case analysis, we first show that typing derivations can be put in a canonical form, where successive applications of Rule SHIFT have been fused.

Proposition 2 (Progress)**.** *If* $\Gamma \vdash M : \tau$ *and* Γ *does not contain value variable bindings, then either* M *is a result, or it is reducible.*

Progress is proved by induction on the typing derivation. The side condition that Γ does not contain any value variable is as usual. However, we cannot require the more restrictive hypothesis that Γ be empty as evaluation takes place under existential quantifiers. Moreover, this allows to consider the reduction of *open* programs, *i.e.* programs with free type variables. This is the case for programs with abstract types, which come from unrestricted openings. This closely corresponds to ML programs composed of modules with abstract types.

4. Translations from and to System F

We propose a translation from System F to $\mathsf{F}^{\forall}$ and, conversely, from a subset of $\mathsf{F}^{\forall}$ to System F. Although the first one is compositional, the latter is not. This embodies the fact that $\mathsf{F}^{\forall}$ is superior to system F in terms of modularity.

From F to $\mathsf{F}^{\forall}$. The syntactic sugar for pack and unpack provides the translation from F to $\mathsf{F}^{\forall}$, which is obviously compositional and both type and semantic preserving.

From pure $\mathsf{F}^{\forall}$ to F. The inverse translation merits closer examination. Hereafter, we define a non-compositional transformation from *pure* terms to terms of System F that is both type and semantics preserving in the following sense: the type-erasure[2] of the translation of M is not equal but let-reduces to the type-erasure of M. This remark already supplies some piece of information about modularity of $\mathsf{F}^{\forall}$ compared to F: the translation involves code shifting. Assuming that the translation is "optimal", this means that $\mathsf{F}^{\forall}$ brings more freedom, hence modularity, to write programs, whereas F requires a stricter definition discipline—imposed by the unpack construct, indeed. The translation is the composition of two sub-transformations: the first one takes care of pack's, and the second one copes with unpack's.

The pack constructs are recovered from the typing derivation. The transformation is specified by a judgment of the form $\Gamma \vdash M : \tau' \triangleright M'$ where M' is the translation of M. The main cases are given on Fig. 8: on the one hand, coercions are erased, whether they are attached to record fields or not; on the other hand, existential constructs are translated into pack's, where the witness substitution, which is pushed into the context, is applied to the subterm being packed.

The recovering of unpack's requires more care, as it does not preserve the structure of terms. It is defined through a rewriting system. Selected rules are given by Fig. 9. It is based on the following property: if $\nu\alpha.\,M$ is

[2] Type erasure is defined as usual—See Fig. 17 in the Appendix.

$$\frac{\Gamma, \alpha = \tau \vdash M \,:\, \tau' \triangleright M'}{\Gamma \vdash \exists(\alpha = \tau)\, M \,:\, \exists\alpha.\,\tau' \triangleright \mathsf{pack}\,\langle \tau, M'\,[\alpha \leftarrow \tau]\rangle\ \mathsf{as}\ \exists\alpha.\,\tau'} \qquad \frac{\Gamma \vdash M \,:\, \tau' \triangleright M' \qquad \Gamma \vdash \tau' \approx \tau}{\Gamma \vdash (M : \tau) \,:\, \tau \triangleright M'}$$

$$\frac{\Gamma \vdash M \,:\, \tau' \triangleright M' \qquad \Gamma \vdash \tau' \approx \tau \qquad \Gamma' \vdash \{r\} \,:\, \{(\ell_i : \tau_i)^{i\in 1..n}\} \triangleright \{r'\}}{\Gamma \curlyvee \Gamma' \vdash \{\ell \,:\, \tau = M \,;\, r\} \,:\, \{\ell : \tau \,;\, (\ell_i : \tau_i)^{i\in 1..n}\} \triangleright \{\ell = M' \,;\, r'\}}$$

Figure 8. From $\mathsf{F}^{\curlyvee}$ to F, first pass: recovering pack's (selected rules).

$$\begin{array}{rcl}
Q^\alpha & ::= & \mathsf{open}^\alpha\ M \;\mid\; Q^\alpha\ M \;\mid\; M\ Q^\alpha \;\mid\; Q^\alpha\ [\tau] \;\mid\; \mathsf{pack}\,\langle \tau, Q^\alpha\rangle\ \mathsf{as}\ \exists\beta.\,\tau' \;\mid\; \mathsf{unpack}\ Q^\alpha\ \mathsf{as}\ \beta, x\ \mathsf{in}\ M \\
& \mid & \mathsf{unpack}\ M\ \mathsf{as}\ \beta, x\ \mathsf{in}\ Q^\alpha \quad \text{if } \beta \neq \alpha \;\mid\; \nu\beta.\,Q^\alpha \quad \text{if } \beta \neq \alpha \;\mid\; \mathsf{open}^\beta\ Q^\alpha \quad \text{if } \beta \neq \alpha \;\mid\; Q^\alpha.\ell \\
& \mid & \{r \,;\, \ell = Q^\alpha \,;\, r'\} \;\mid\; \mathsf{let}\ x = M\ \mathsf{in}\ Q^\alpha \;\mid\; \mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ M
\end{array}$$

$$\begin{array}{rcl}
\nu\alpha.\ (Q^\alpha\ M) & \twoheadrightarrow & (\nu\alpha.\,Q^\alpha)\ M \\
\nu\alpha.\ (\mathsf{let}\ x = M\ \mathsf{in}\ Q^\alpha) & \twoheadrightarrow & \mathsf{let}\ x = M\ \mathsf{in}\ \nu\alpha.\,Q^\alpha \\
\nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ M\ \mathsf{in}\ M' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ M\ \mathsf{in}\ M' \\
\nu\alpha.\ (\mathsf{open}^\alpha\ M) & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ x \\
\nu\alpha.\ (\mathsf{let}\ x = \mathsf{open}^\alpha\ M\ \mathsf{in}\ M') & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M'
\end{array}$$

Figure 9. From $\mathsf{F}^{\curlyvee}$ to F, second pass: recovering unpack's (selected rules).

welltyped, then there is a *unique* subterm $\mathsf{open}^\alpha\ M'$ inside M and it is not located under λ's of Λ's. Thus, M is a term of the form Q^α, as defined[3] in Fig. 9. The rewriting rules operate on terms of the form $\nu\alpha.\,Q^\alpha$ and bring a restriction and its opening closer by decreasing the size of Q^α. This is done either by predefining the subterm containing the opening thanks to a let when it is possible or by making the restriction go down otherwise. The last two rules are terminal: they replace openings with unpackings when the pattern of unpack is recognized. When no rewriting rule applies anymore, the term has no restriction left. If moreover M is typable in a pure typing context, then it cannot contain openings either. So, the resulting term is in System F. Furthermore, each rewriting rule is both type and semantics preserving (the type-erasure of the right-hand side let-reduces to the type-erasure of the left-hand side). Appendix contains an application of the translation on an example.

It is an open question whether the translation can be extended to impure terms, *i.e.* terms that are welltyped in arbitrary contexts. The issue is how to keep track of linearity of existential resources.

5. Paths and shapes

A sledgehammer argument for paths in modules is their ability to factorize code [7, 13], as exemplified by the well-known diamond import pattern. Although this justifies the need of a path system, this does not specifically vindicate the use of *paths at the term level*. Instead, we introduce a path system *at the level of types*, which is composed of two key ingredients: type shapes and type projections.

[3] We temporarily add let's, pack's and unpack's to the syntax to allow for easier reasoning.

$$\begin{array}{rcll}
\kappa & ::= & \star \;\mid\; \kappa \to \kappa \;\mid\; \{(\ell_i : \kappa_i)^{i\in 1..n}\} & \text{(Kinds)} \\
\theta & ::= & \alpha \in \sigma & \\
\sigma & ::= & \top \;\mid\; \tau \;\mid\; [\theta]\ \sigma & \text{(Shapes)} \\
\tau & ::= & \ldots \;\mid\; \tau.\ell \;\mid\; \tau.1 \;\mid\; \tau.2 & \text{(Projections)}
\end{array}$$

Figure 10. Shapes and type projections.

5.1. Definitions

Differences from $\mathsf{F}^{\curlyvee}$ are described on Fig. 10 and explained below. We define kinds to forbid illegal projections: they consist in the base kind, the kind of arrow types and the kind of record types. By lack of space, we refer to the Appendix for details (Fig. 18 and 19). The syntactic categories θ, σ and τ are mutually recursively defined. They are explained hereafter.

Shapes. Shapes represent sets of types and are used to restrict the range of binding type variables. Their interpretation is given on Fig. 11: the top shape $\top$ is the set of all (wellformed) types, the singleton shape τ is the set containing the unique type τ, and the compound shape $[\alpha \in \sigma]\ \sigma'$ is the set of types $\tau'\,[\alpha \leftarrow \tau]$, where $\tau' \in \sigma'$ and $\tau \in \sigma$.

Binding occurrences of type variables henceforth carry a shape. For example, the identity function on homogeneous pairs (*i.e.* pairs whose components are of the same type), has type $\forall(\alpha \in \top)\,\{\ell : \alpha\,;\, \ell' : \alpha\} \to \{\ell : \alpha\,;\, \ell' : \alpha\}$, but also $\forall(\alpha \in [\beta \in \top]\ \{\ell : \beta\,;\, \ell' : \beta\})\,\alpha \to \alpha$. Shapes will play the role of ML's module signatures.

Type projections. Shapes can already express sharing properties, internally, as shown above. However, without

$$\frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau \in \top} \qquad \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau \in \tau}$$

$$\frac{\Gamma \vdash \tau \in \sigma \qquad \Gamma, \forall\,(\alpha \in \sigma) \vdash \tau' \in \sigma'}{\Gamma \vdash \tau'\,[\alpha \leftarrow \tau] \in [\alpha \in \sigma]\ \sigma'}$$

Figure 11. Meaning of shapes.

external access to shapes their benefits would be very limited. We introduce projections to access shapes externally.

The typing rules ensure that a universally typed variable α of shape σ will only be instantiated by types that are in σ. Hence, if the shape only contains, for instance, arrow types, which is the case if it has an arrow kind, then $\alpha.1$ and $\alpha.2$ designate the domain and the codomain of α. Similarly, if the shape only contains records with at least field ℓ, which is the case if the kind of the shape is a record kind with a field ℓ, then $\alpha.\ell$ designates the ℓ projection of α.

For instance, a function taking two pairs as arguments and returning the pair of their first components can be given the following type:

$$\begin{array}{c}\forall(\alpha_1 \in [\beta_1 \in \top]\ [\beta'_1 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta'_1\}) \\ \forall(\alpha_2 \in [\beta_2 \in \top]\ [\beta'_2 \in \top]\ \{\ell : \beta_2\,;\, \ell' : \beta'_2\}) \\ \alpha_1 \to \alpha_2 \to \{\ell : \alpha_1.\ell\,;\, \ell' : \alpha_2.\ell\}\end{array}$$

The resulting type is defined in terms of projections of the types of the arguments. Projections may also be used to express sharing *between* the shapes of the arguments, much as paths may express sharing between signatures of the arguments of a functor in ML. Pursuing the example, we may require that the first two components have the same type, so that the result type is an homogeneous pair, as follows:

$$\begin{array}{c}\forall(\alpha_1 \in [\beta_1 \in \top]\ [\beta'_1 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta'_1\}) \\ \forall(\alpha_2 \in [\beta'_2 \in \top]\ \{\ell : \alpha_1.\ell\,;\, \ell' : \beta'_2\}) \\ \alpha_1 \to \alpha_2 \to \{\ell : \alpha_1.\ell\,;\, \ell' : \alpha_1.\ell\}\end{array}$$

Equivalent types and shapes. Singleton shapes and type projections obviously suggest a notion of type equivalence, which we introduce as a judgment of the form $\Gamma \vdash \tau \equiv \tau'$. The main rules are shown in Fig. 12: Rule EQUI-VAR identifies elements of a singleton shape; types are equivalent up to projections (rules EQUI-PROJ-*). Rule EQUI-FOLD unfolds compound shapes inside the typing environment so as to expose inner sharing. Notice that this is a form of extrusion. (The whole set of rules can be found in the Appendix, Fig. 22.) We also define an equivalence on shapes from their set-based semantics: two shapes are equivalent if they contain the same types, which can also be characterized syntactically (Fig. 23 in the Appendix).

Of course, equivalent types are then identified through an additional, straightforward typing rule (Appendix, Fig.24, Rule EQUIV).

Summary. Shapes are stunningly simple objects: a remarkable feature of shape equivalence is that shapes have a

$$\text{EQUI-VAR}\ \frac{(\alpha \in \tau) \in \Gamma}{\Gamma \vdash \alpha \equiv \tau} \qquad \text{EQUI-PROJ-LABEL}\ \frac{\Gamma \vdash \tau \equiv \{\ell_i : \tau_i^{i \in 1..n}\} \qquad 1 \le k \le n}{\Gamma \vdash \tau.\ell_k \equiv \tau_k}$$

$$\text{EQUI-PROJ-ARROW}\ \frac{\Gamma \vdash \tau \equiv \tau_1 \to \tau_2}{\Gamma \vdash \tau.i \equiv \tau_i} \qquad \text{EQUI-FOLD}\ \frac{\alpha \notin \mathit{ftv}(\Gamma', \tau, \tau') \quad \Gamma, \forall\,(\alpha \in \sigma), \forall\,(\alpha' \in \sigma'), \Gamma' \vdash \tau \equiv \tau'}{\Gamma, \forall\,(\alpha' \in [\alpha \in \sigma]\ \sigma'), \Gamma' \vdash \tau \equiv \tau'}$$

Figure 12. Type equivalence (selected rules).

canonical form, where only $\top$ is used on the left-hand side of compound shapes. Hence, shapes are nothing more than types with holes (*i.e.* $\top$'s) and sharing: they can be represented as dags. The syntax of shapes may be extended with sharing constraints, much as the *with* construct in module signatures, enabling the definitions of new shapes from older ones, concisely.

5.2. Applications

Shapes increase conciseness and modularity. Shapes are an orthogonal extension to open existential types, and can already increase conciseness in System F, as illustrated by the use of projections in the examples above.

Another explanation of the power of shapes, is given by (a sketch of) a translation into System F: quantification over a type of the trivial shape $\top$ is translated into the usual quantification; quantification over a type of a singleton shape is translated by substituting the type of the singleton, which illustrates the *sharing power of shapes*. Finally, quantification over a compound shape $\forall(\alpha \in [\beta \in \sigma]\ \sigma')\,\tau$ is translated into an additional quantification $\forall(\beta \in \sigma)\,\forall(\alpha \in \sigma')\,\tau$. Thus, compound shapes can be seen as a way to save up explicit quantifications.

Simultaneous openings. An interesting application of paths is the factorization of openings, as in $\Sigma^{\beta}(\alpha \in [\gamma_1 \in \top]\,[\gamma_2 \in \top]\,\{\ell_1 : \gamma_1; \ell_2 : \gamma_2\} = \tau)\,M$. This adds β to the typing context, hence provides two fresh names $\beta.\ell_1$ and $\beta.\ell_2$ as if we had done two separate openings. This usage corresponds to modules with multiple abstract type components.

Scopeless type definitions. Paths also provide $\mathsf{F}^{\forall}$ with scopeless type definitions, as a side effect, by opening an existential value whose witness is specified with a singleton shape as in $\Sigma^{\beta}(\alpha \in \tau = \tau)\,M$. Since β has shape τ, $\beta \equiv \tau$ holds. This fact appears in the typing context of M as the binder $\exists\,(\beta \in \tau)$ is present and carried over by the zipping of contexts outside of M as the binder $\forall\,(\beta \in \tau)$, so that the information $\beta \equiv \tau$ is reachable in the whole program. A simple example can be found in Appendix.

Wrap-up. In spite of its intrinsic simplicity, the path system we propose is sufficiently expressive to write modules

and functors as compactly as in ML, while avoiding the cumbersome use of type components.

6. Related work

Open existential types. Russo [17] justifies the meaninglessness of paths for module types, by interpreting modules and signatures into semantic objects with System F types. He makes use of existential quantifiers to track type generativity. Unfortunately, no semantics is given for the semantic objects, so that one cannot calculate with them. Perhaps, $\mathsf{F}^{\curlyvee}$ can be seen as a concrete calculus for his system.

In the context of run-time type inspection, Rossberg [16] introduces λ_{N}, a version of System F with a construct to define abstract types and a mechanism of directed coercions. His abstract types can be automatically extruded to allow sharper type analysis, and are thus close to our Σ binder. His coercions resemble ours, though ours are symmetric, because they never cross the abstraction barrier. Although both systems seem kindred in spirit, they are subtly different, because they have been designed for different purposes: in particular, λ_{N} is only partially related to traditional existential types, since parametricity is purposely violated.

Dreyer defines RTG to deal with type generativity in the context of recursive modules [3]. He introduces *type references* which can be written at most once. The creation of a type reference with "new α in M" introduces a type variable in the scope of M that should be treated as a resource that can be set at most once, with his type reference update "set $\alpha := \tau$ in M". Then, M and only M will see the concrete definition τ for α while other paths of the program will see α abstractly.

Technically, the treatment of these linear resources differ significantly from ours: his semantics employs a type store to model static but imperative type reference updates, whereas we just use extrusions of Σ binders[4]. He uses an effect type system to guarantee the uniqueness of writing, which exposes the evaluation order in the typing rules of RTG, moving away from a logical specification, whereas we use *zipping* of contexts—a symmetric operation—to enforce sound openings and maintain a close correspondence with logic. *Intuitively,* we think of existential values as generating a fresh type when opened, while he sees them as functions in "destination passing style".

Despite these strong technical differences, the two systems can be put in close correspondence, when we remove paths from $\mathsf{F}^{\curlyvee}$ and equirecursive and higher-order types from RTG. In both directions, there exists an encoding[5] based on the typing derivation that replaces the new and set constructs with our ν and Σ binders, or conversely. In the forward direction, we assume that $\mathsf{F}^{\curlyvee}$ terms have been rearranged to use the pack form only. In the backward direction, we assume that terms of RTG do not use a type variable before it has been written, as this would translate in the general case into recursively defined abstract types, which $\mathsf{F}^{\curlyvee}$ does not allow. Interestingly, RTG has been designed specifically to allow this situation, as its goal was to model recursive modules, whereas $\mathsf{F}^{\curlyvee}$ has disallowed this situation to avoid the complexity of equirecursive types in a first step, as they are not needed to model ML modules. In this restrictive case, $\mathsf{F}^{\curlyvee}$ can *a posteriori* be seen as an alternative presentation of RTG in direct style.

Paths. Our type projections are similar to Hofmann and Pierce [8]'s type destructors, which they proposed to increase the expressiveness of $\mathsf{F}_{\leq}$. However, the supertype bounds of $\mathsf{F}_{\leq}$ have been replaced with shapes in $\mathsf{F}^{\curlyvee}$, which are considerably simpler.

Concluding remarks

We have defined $\mathsf{F}^{\curlyvee}$, a variant of explicitly-typed System F with primitive *open existential types*, which generalize the usual notion of (closed) existential types by splitting their creation and elimination into more atomic constructs. The subject reduction and progress theorems holds for $\mathsf{F}^{\curlyvee}$ and have straightforward routine proofs.

We have shown how openings of existential values tightly correspond to type abstraction and generativity in modules. Translations from and to System F have illustrated the modularity gain brought by $\mathsf{F}^{\curlyvee}$ and have restored the close connection between abstract programs and logic.

We have proposed a new notion of paths, at the level of types, that brings back the conciseness of writing provided by ML modules with almost no technical overhead. Moreover, we have kept type abstraction and paths as two independent mechanisms, though they are usually presented together as an atomic package.

We believe that $\mathsf{F}^{\curlyvee}$ is promising as the core of a programming language with first-class modules. The *bare simplicity* of $\mathsf{F}^{\curlyvee}$ is its best asset.

We have purposely limited the expressiveness of $\mathsf{F}^{\curlyvee}$ to the minimum that permits programming with modules in the ML-like style, directly. The integration of general purpose features such as recursive types, value references or recursion, should be straightforward. Extending $\mathsf{F}^{\curlyvee}$ both with equirecursive and with higher-order kinds while preserving its direct and logical style is currently under work, and promising. The former is required to enable programming with—and hopefully provide a simpler foundations for—recursive modules and mixin modules. The latter is motivated by Russo's work on modeling applicative functors. We have avoided the complication of using linear types by keeping resources in the typing contexts. Interestingly,

[4] Perhaps, these two approaches could be related by seeing our extrusion as a local treatment of his type store, as has been proposed for value references [5].

[5] The details of both encodings can be found in Appendix §A.4.

our systems remains sufficiently expressive to model ML-like modules. Still, this restriction prevents the definition of impure functions and breaks the usual symmetry between contexts and expressions. Adding linear types should not raise technical problems, but this remains to be investigated.

Of course, some form of type inference would eventually be needed in a real programming language based on $\mathsf{F}^{\forall}$. An easy solution is to stratify the type system—just for the purpose of type inference. We could infer ML-like types for the base level and require explicit type information for the module level, as for ML. Another more ambitious direction is to use a form of partial type inference with first-class polymorphism.

Acknowledgments The authors owe much to Paul-André Melliès for fruitful discussions that lead to technical simplifications of our proposal and for suggesting the *zipper* metaphor. We also thank Arthur Charguéraud and François Pottier for helpful discussions and comments.

References

[1] L. Cardelli and X. Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990. Also available as research report 56, DEC Systems Research Center.

[2] J. Courant. An applicative module calculus. In *Theory and Practice of Software Development 97*, Lecture Notes in Computer Science, pages 622–636, Lille, France, April 1997. Springer-Verlag.

[3] D. Dreyer. Recursive type generativity. *Journal of Functional Programming*, pages 433–471, 2007.

[4] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *2003 ACM SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[5] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[6] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM.

[7] R. Harper and B. C. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. The MIT Press, 2005.

[8] M. Hofmann and B. C. Pierce. Type destructors. In D. Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[9] X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[10] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.10*. INRIA, May 2007.

[11] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[12] D. MacQueen. Modules for standard ML. In *Proceedings of the ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

[13] D. B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 277–286, New York, NY, USA, 1986. ACM.

[14] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[15] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of ECOOP*, 2003.

[16] A. Rossberg. Generativity and dynamic opacity for abstract types. In *5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, September 2003.

[17] C. V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60, January 2003.

[18] *Standard ML of New Jersey User's Guide*.

A. Appendix

A.1. Additional example

The splitting of pack allows for putting the information of hiding parts of a type deeply inside a term: in the following record, some leaves have been abstracted away.

$$\begin{array}{l}\exists(\alpha = \mathit{int})\\ \quad \mathsf{let}\ x = \{\ell_1 = (1 : \alpha)\ ;\ \ell_2 = 2\}\ \mathsf{in}\\ \quad \mathsf{let}\ y = \{\ell_1 = x\ ;\ \ell_2 = x\}\ \mathsf{in}\\ \quad \{\ell_1 = y\ ;\ \ell_2 = y\}\end{array}$$

The corresponding system F term requires to repeat the type of the whole term.

$$\begin{array}{l}\mathsf{let}\ z =\\ \quad \mathsf{let}\ x = \{\ell_1 = 1\ ;\ \ell_2 = 2\}\ \mathsf{in}\\ \quad \mathsf{let}\ y = \{\ell_1 = x\ ;\ \ell_2 = x\}\ \mathsf{in}\\ \quad \{\ell_1 = y\ ;\ \ell_2 = y\}\ \mathsf{in}\\ \mathsf{pack}\ \langle \mathit{int}, z\rangle\ \mathsf{as}\\ \quad \exists\alpha.\ \{\ell_1 : \{\ell_1 : \{\ell_1 : \alpha\ ;\ \ell_2 : \mathit{int}\}\ ;\ \ell_2 : \{\ell_1 : \alpha\ ;\ \ell_2 : \mathit{int}\}\}\ ;\\ \qquad \ell_2 : \{\ell_1 : \{\ell_1 : \alpha\ ;\ \ell_2 : \mathit{int}\}\ ;\ \ell_2 : \{\ell_1 : \alpha\ ;\ \ell_2 : \mathit{int}\}\}\}\end{array}$$

Moreover, whereas the information of hiding was located at a single place in the $\mathsf{F}^{\forall}$-term, it is duplicated in the F-term, as if each leaf had been abstracted independently.

A.2. Translations from and to system F

This section contains some formal results and an example about the transformations presented in §4.

A.2.1 From F to $\mathsf{F}^{\forall}$

The (compositional) transformation from F to $\mathsf{F}^{\forall}$ consists in unfolding the syntactic sugar for pack and unpack. Its is characterized by the following property.

Proposition 3. *There exists a* compositional *transformation* $[\![\cdot]\!]$ *from* F *to* $\mathsf{F}^{\forall}$ *such that:*

- $\lfloor [\![M]\!] \rfloor = \lfloor M \rfloor$
- *if* $\Gamma \vdash_{\mathsf{F}} M : \tau$ *then* $\Gamma \vdash_{\mathsf{F}^{\forall}} [\![M]\!] : \tau$.

$\lfloor \cdot \rfloor$ is a type-erasure operation that is defined on Fig. 17.

A.2.2 From $\mathsf{F}^{\forall}$ to F

The encoding of a subset of $\mathsf{F}^{\forall}$ into F enjoys the following property.

Proposition 4. *There exists a* global *transformation* $(\!|\cdot|\!)$ *from* $\mathsf{F}^{\forall}$ *to* F*, defined for all terms* M *verifying* $\Gamma \vdash_{\mathsf{F}^{\forall}} M : \tau$ *for some pure and type-definition-free context* Γ*, such that:*

- $\lfloor (\!|M|\!) \rfloor \xrightarrow{let} \lfloor M \rfloor$
- *if* $\Gamma \vdash_{\mathsf{F}^{\forall}} M : \tau$*, then* $\Gamma \vdash_{\mathsf{F}} (\!|M|\!) : \tau$.

As described in §4, the translation consists in two sub-transformations: the first one deals with the recovering of pack's, the second one copes with unpack's. The soundness of the first pass is ensured by the following lemma.

Lemma 3. *If* $\Gamma \vdash M : \tau \triangleright M'$*, then* $\lfloor M \rfloor = \lfloor M' \rfloor$*. Moreover,* $\Gamma\sigma \vdash M'\sigma : \tau\sigma$ *and* $\Gamma \vdash \tau \approx \tau\sigma$ *where* σ *denotes the substitution induced by the equations of* Γ.

As the typing environment is required to be type-definition-free, the transformation is type preserving. The property between the underlying untyped terms of the source and the target are proved without difficulty by induction on the derivation.

Soundness of the second pass (Fig. 16) is proved independently for each rule.

Example 1. A pure $\mathsf{F}^{\forall}$-term and its translation in F.

Before the transformation:

$\nu\alpha.$
let $x = \exists(\beta = \mathit{int})$
  $\{\mathit{encode} : \mathit{int} \to \beta =$
    $\lambda(n : \mathit{int})$
      if $n < 0$ then $-2 * n + 1$ else $2 * n$;
   $\mathit{decode} : \beta \to \mathit{int} =$
    $\lambda(n : \mathit{int})$
      if $n \bmod 2 = 0$ then $n/2$ else $(1-n)/2\}$
in let $f = \Lambda\beta.$
  $\lambda(p : \{\mathit{encode} : \mathit{int} \to \beta ;\ \mathit{decode} : \beta \to \mathit{int}\})$
    $\lambda(x : \mathit{int})\ p.\mathit{decode}\ (p.\mathit{encode}\ x)$
in $\{\ell_1 = x ;\ \ell_2 = f ;\ \ell_3 = f\ [\alpha]\ (\mathsf{open}^{\alpha}\ x)\ 2008\}$

After the transformation:

let $x = \exists(\beta = \mathit{int})$
  $\{\mathit{encode} : \mathit{int} \to \beta =$
    $\lambda(n : \mathit{int})$
      if $n < 0$ then $-2 * n + 1$ else $2 * n$;
   $\mathit{decode} : \beta \to \mathit{int} =$
    $\lambda(n : \mathit{int})$
      if $n \bmod 2 = 0$ then $n/2$ else $(1-n)/2\}$
in let $f = \Lambda\beta.$
  $\lambda(p : \{\mathit{encode} : \mathit{int} \to \beta ;\ \mathit{decode} : \beta \to \mathit{int}\})$
    $\lambda(x : \mathit{int})\ p.\mathit{decode}\ (p.\mathit{encode}\ x)$
in $\{\ell_1 = x ;\ \ell_2 = f ;$
  $\ell_3 = \mathsf{unpack}\ x\ \mathsf{as}\ \alpha, y\ \mathsf{in}\ f\ [\alpha]\ (y)\ 2008\}$

Example 2. Another application of the transformation.

Before the transformation:

$\nu\alpha.$
let $r = \exists(\beta = \mathit{int})$
  $\{\mathit{encode} : \mathit{int} \to \beta =$
    $\lambda(n : \mathit{int})$
      if $n < 0$ then $-2 * n + 1$ else $2 * n$;
   $\mathit{decode} : \beta \to \mathit{int} =$
    $\lambda(n : \mathit{int})$
      if $n \bmod 2 = 0$ then $n/2$ else $(1-n)/2\}$ in
let $x =$
  let $y =$
    let $z = \{\ell = \mathsf{open}^{\alpha}\ r ;\ \ell' = 1\}$ in $z.\ell$
  in $\lambda(x : \mathit{int})\ y.\mathit{decode}\ y.\mathit{encode}\ x$
in $x\ 1$

After the transformation:

let $r = \exists(\beta = \mathit{int})$
  $\{\mathit{encode} : \mathit{int} \to \beta =$
    $\lambda(n : \mathit{int})$
      if $n < 0$ then $-2 * n + 1$ else $2 * n$;
   $\mathit{decode} : \beta \to \mathit{int} =$
    $\lambda(n : \mathit{int})$
      if $n \bmod 2 = 0$ then $n/2$ else $(1-n)/2\}$ in
unpack r as α, t in
  let $u = \{\ell = t ;\ \ell' = 1\}$ in
  let $x =$
    let $y =$
      let $z = u$ in $z.\ell$
    in $\lambda(x : \mathit{int})\ y.\mathit{decode}\ y.\mathit{encode}\ x$
  in $x\ 1$

A.3. Paths and shapes

A.3.1 Examples with functors

Example 3. Three versions of the identity function specialized on pairs:

- In System F: $\Lambda\alpha.\,\Lambda\beta.\,\lambda(x : \{\ell : \alpha ;\ \ell' : \beta\})\,x$
  And its type: $\forall\alpha.\,\forall\beta.\,\{\ell : \alpha ;\ \ell' : \beta\} \to \{\ell : \alpha ;\ \ell' : \beta\}$.

- With shapes:
  $\Lambda(\alpha \in [\beta_1 \in \top]\ [\beta_2 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta_2\})$
  $\quad \lambda(x\!:\!\alpha)\, x$
  And its type:
  $\forall(\alpha \in [\beta_1 \in \top]\ [\beta_2 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta_2\})\, \alpha \to \alpha$

- In ML:
  functor
  $\quad (X : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $= X$
  An its type:
  functor
  $\quad (X : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $\to \mathsf{sig\ type}\ t_1 = X.t_1\ \mathsf{type}\ t_2 = X.t_2$
  $\qquad \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end}$

Example 4. Three versions of a function taking two pairs as arguments and returning the pair of their first components:

- In System F:
  $\Lambda\alpha_1.\,\Lambda\alpha_2.\,\Lambda\beta_1.\,\Lambda\beta_2.$
  $\lambda(x\!:\!\{\ell : \alpha_1\,;\, \ell' : \alpha_2\})\,\lambda(y\!:\!\{\ell : \beta_1\,;\, \ell' : \beta_2\})$
  $\quad \{\ell = x.\ell\,;\, \ell' = y.\ell\}$
  And its type:
  $\forall\alpha_1.\,\forall\alpha_2.\,\forall\beta_1.\,\forall\beta_2.$
  $\{\ell : \alpha_1\,;\, \ell' : \alpha_2\} \to \{\ell : \beta_1\,;\, \ell' : \beta_2\}$
  $\to \{\ell : \alpha_1\,;\, \ell' : \beta_1\}$

- With shapes:
  $\Lambda(\alpha \in [\alpha_1 \in \top]\ [\alpha_2 \in \top]\ \{\ell : \alpha_1\,;\, \ell' : \alpha_2\})$
  $\Lambda(\beta \in [\beta_1 \in \top]\ [\beta_2 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta_2\})$
  $\quad \lambda(x\!:\!\alpha)\,\lambda(y\!:\!\beta)\,\{\ell = x.\ell\,;\, \ell' = y.\ell\}$
  And its type:
  $\forall(\alpha \in [\alpha_1 \in \top]\ [\alpha_2 \in \top]\ \{\ell : \alpha_1\,;\, \ell' : \alpha_2\})$
  $\forall(\beta \in [\beta_1 \in \top]\ [\beta_2 \in \top]\ \{\ell : \beta_1\,;\, \ell' : \beta_2\})$
  $\quad \alpha \to \beta \to \{\ell : \alpha.\ell\,;\, \ell' : \beta.\ell\}$

- In ML:
  functor
  $\quad (X : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $\quad (Y : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $= \mathsf{struct\ val}\ x = X.x\ \mathsf{val}\ x' = Y.x\ \mathsf{end}$
  And its type:
  functor
  $\quad (X : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $\to$ functor
  $\quad (Y : \mathsf{sig\ type}\ t_1\ \mathsf{type}\ t_2\ \mathsf{val}\ x : t_1\ \mathsf{val}\ x' : t_2\ \mathsf{end})$
  $\to \mathsf{sig\ val}\ x : X.t_1\ \mathsf{val}\ x' : Y.t_2\ \mathsf{end}$

A.3.2 Examples with existentials

Example 5. Comparison of scopeless type definitions in ML and $\mathsf{F}^{\curlyvee}$ with paths.

- In ML:
  $\mathsf{module}\ M =$
  $\quad \mathsf{struct\ type}\ t = \mathit{int} * \mathit{bool}\ \mathsf{val}\ x = (1, \mathit{false})\ \mathsf{end}$
  $\mathsf{let}\ f\ (p : M.t) = (1 + \mathit{fst}\ p, \mathit{snd}\ p)\ \mathsf{in}$
  $f\ M.x$

- In $\mathsf{F}^{\curlyvee}$ with paths:
  $\mathsf{let}\ m =$
  $\Sigma^{\beta}(\alpha \in \{\ell = \mathit{int}\,;\, \ell' = \mathit{bool}\} = \{\ell = \mathit{int}\,;\, \ell' = \mathit{bool}\})$
  $\quad \{x = \{\ell = 1\,;\, \ell' = \mathit{false}\}\}\ \mathsf{in}$
  $\mathsf{let}\ f = \lambda(p\!:\!\beta)\,\{\ell = 1 + p.\ell\,;\, \ell' = p.\ell'\}\ \mathsf{in}$
  $\quad f\ m.x$

In the second program, let us call τ_0 the type $\{\ell : \mathit{int}\,;\, \ell' : \mathit{bool}\}$. Then let us call M and P the subterm that defines the variable m and the one following m's definition, respectively. The last node of its derivation is as follows:

$$\frac{\exists\,(\beta \in \tau_0) \vdash M\ :\ \{x : \tau_0\} \qquad \forall\,(\beta \in \tau_0)\,, m : \{x : \tau_0\} \vdash P\ :\ \{x : \tau_0\}}{\exists\,(\beta \in \tau_0) \vdash \mathsf{let}\ m = M\ \mathsf{in}\ P\ :\ \tau_0}$$

As M opens an existential value with the name β, its typing environment contains the existential binding $\exists\,(\beta \in \tau_0)$. Notice the zipping operation transforms it into an universal one $\exists\,(\beta \in \tau_0)$ in the second branch of the let. Thus in both branches we can prove $\beta \equiv \tau_0$. This is as in the above ML code: inside and outside module M, we know the definition of $M.t$ since it is transparent.

A.4. Comparison between $\mathsf{F}^{\curlyvee}$ and RTG

Our system $\mathsf{F}^{\curlyvee}$ and Dreyer's RTG are tightly related. Using Dreyer's *destination-passing style* (DPS) encoding of existentials, we describe—informally—an encoding of $\mathsf{F}^{\curlyvee}$ into RTG:

$$[\![\exists\alpha.\,\tau]\!]_{\mathrm{RTG}} = \forall^{\uparrow}\alpha.\,\mathit{unit} \xrightarrow{\alpha\downarrow} [\![\tau]\!]$$

$$[\![\mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as}\ \exists\alpha.\,\tau'\,]\!]_{\mathrm{RTG}} =$$
$$\quad \mathsf{let}\ x = [\![M]\!]_{\mathrm{RTG}}\ \mathsf{in}$$
$$\quad \Lambda^{\uparrow}\alpha.\,\lambda(x\!:\!\mathit{unit})\ \mathsf{set}\ \alpha := [\![\tau]\!]_{\mathrm{RTG}}\ \mathsf{in}\ x : [\![\tau']\!]_{\mathrm{RTG}}$$

$$[\![\nu\alpha.\,M]\!]_{\mathrm{RTG}} = \mathsf{new}\ \alpha\ \mathsf{in}\ [\![M]\!]_{\mathrm{RTG}}$$

$$[\![\mathsf{open}^{\alpha}\ M]\!]_{\mathrm{RTG}} = \mathsf{let}\ x = [\![M]\!]_{\mathrm{RTG}}\ \mathsf{in}\ x\ [\alpha]\ ()$$

This puts in correspondence both points of view. Creating an existential value consists in delaying a type variable assignment. Opening an existential value corresponds to releasing the writing of a type reference. Finally, the creation of an existential resource in the typing environment is similar to the creation of a writable type reference.

Although this encoding is compositional, it is restricted to source terms where all existential values are introduced with the pack form. Thus, the more liberal forms of existential introduction $\exists(\alpha = \tau)\,M$ should have been put in pack forms first, based on their typing derivation.

The inverse translation can be informally defined on a strict subset of RTG that does not use equirecursive, higher-

order types and recursion. More precisely, we forbid using a writable type reference before it has been written. We purposely introduced this restriction in $\mathsf{F}^{\curlyvee}$, for simplicity, to keep the correspondence with System F, and because it was sufficient to encode ML modules.

Assuming this restriction on RTG terms, we give—informally—the following translation:

$$\left[\!\left[\forall^{\uparrow}\alpha.\,\tau_1 \xrightarrow{\alpha\downarrow} \tau_2\right]\!\right]_{\mathsf{F}^{\curlyvee}} = [\![\tau_1]\!]_{\mathsf{F}^{\curlyvee}} \to \exists\alpha.\ [\![\tau_2]\!]_{\mathsf{F}^{\curlyvee}}$$

$$[\![\Lambda^{\uparrow}\alpha.\,\lambda(x{:}\tau_1)\,e]\!]_{\mathsf{F}^{\curlyvee}} =$$
$$\lambda(x{:}\tau_1)\,\nu\alpha.\,\mathsf{let}\ x = [\![e]\!]_{\mathsf{F}^{\curlyvee}}\ \mathsf{in}\ \exists(\beta=\alpha)\,(x : [\![\tau_2]\!]_{\mathsf{F}^{\curlyvee}})$$

where τ_2 is the type of e, extracted from the derivation

$$[\![v_1\ [\alpha]\ v_2]\!]_{\mathsf{F}^{\curlyvee}} = \mathsf{open}^{\alpha}\ ([\![v_1]\!]_{\mathsf{F}^{\curlyvee}}\ [\![v_2]\!]_{\mathsf{F}^{\curlyvee}})$$

$$[\![\mathsf{let}\ \alpha = \tau\ \mathsf{in}\ e]\!]_{\mathsf{F}^{\curlyvee}} = [\![e]\!]_{\mathsf{F}^{\curlyvee}}\,[\alpha \leftarrow [\![\tau]\!]_{\mathsf{F}^{\curlyvee}}]$$

$$[\![\mathsf{new}\,\alpha\,\mathsf{in}\,e]\!]_{\mathsf{F}^{\curlyvee}} = \nu\alpha.\ [\![e]\!]_{\mathsf{F}^{\curlyvee}}$$

$$[\![\mathsf{set}\,\alpha := \tau_1\,\mathsf{in}\,e : \tau_2]\!]_{\mathsf{F}^{\curlyvee}} =$$
$$\Sigma^{\alpha}(\alpha = [\![\tau_1]\!]_{\mathsf{F}^{\curlyvee}})\,([\![e]\!]_{\mathsf{F}^{\curlyvee}} : [\![\tau_2]\!]_{\mathsf{F}^{\curlyvee}})$$

Thanks to the restriction above, a "DPS function" has a type $\forall^{\uparrow}\alpha.\,\tau_1 \xrightarrow{\alpha\downarrow} \tau_2$ where α is not free in τ_1. Hence, it can be translated into a function of type $[\![\tau_1]\!]_{\mathsf{F}^{\curlyvee}} \to \exists\alpha.\ [\![\tau_2]\!]_{\mathsf{F}^{\curlyvee}}$. Notice that this case requires the typing derivation, as in the other direction. The application of a DPS function is translated accordingly. The translation of other constructs is rather obvious.

A.5. Rules and definitions

The rest of the appendix gathers the rules and definitions used in this paper.

$$\textsc{Wf-Empty}\ \frac{\alpha \in \mathit{dom}\,\Gamma \qquad \Gamma \vdash \mathrm{wf}}{\Gamma \vdash \alpha\,\mathrm{wf}} \qquad \textsc{Wf-Arrow}\ \frac{\Gamma \vdash \tau\,\mathrm{wf} \qquad \Gamma \vdash \tau'\,\mathrm{wf}}{\Gamma \vdash \tau \to \tau'\,\mathrm{wf}}$$

$$\textsc{Wf-Record}\ \frac{\forall i, j,\ i \neq j \Rightarrow \ell_i \neq \ell_j \qquad \forall i \in \{1..n\}\,, \quad \Gamma \vdash \tau_i\,\mathrm{wf}}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\}\,\mathrm{wf}} \qquad \textsc{Wf-Forall}\ \frac{\Gamma, \forall\alpha \vdash \tau\,\mathrm{wf}}{\Gamma \vdash \forall\alpha.\,\tau\,\mathrm{wf}} \qquad \textsc{Wf-Exists}\ \frac{\Gamma, \exists\alpha \vdash \tau\,\mathrm{wf}}{\Gamma \vdash \exists\alpha.\,\tau\,\mathrm{wf}}$$

$$\textsc{Ok-Empty}\ \frac{}{\varepsilon \vdash \mathrm{wf}} \qquad \textsc{Ok-Var}\ \frac{\Gamma \vdash \mathrm{wf} \qquad \Gamma \vdash \tau\,\mathrm{wf} \qquad x \notin \mathit{dom}\,\Gamma}{\Gamma, x : \tau \vdash \mathrm{wf}} \qquad \textsc{Ok-Forall}\ \frac{\Gamma \vdash \mathrm{wf} \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \forall\alpha \vdash \mathrm{wf}} \qquad \textsc{Ok-Exists}\ \frac{\Gamma \vdash \mathrm{wf} \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \exists\alpha \vdash \mathrm{wf}}$$

$$\textsc{Ok-Eq}\ \frac{\Gamma \vdash \mathrm{wf} \qquad \Gamma \vdash \tau\,\mathrm{wf} \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \alpha = \tau \vdash \mathrm{wf}}$$

Figure 13. Wellformed types and environments.

$$\textsc{Var}\ \frac{\Gamma \vdash \mathrm{wf} \quad \Gamma\ \mathit{pure}}{\Gamma \vdash x\ :\ \Gamma(x)} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash M\ :\ \tau_2 \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \lambda(x{:}\tau_1)\,M\ :\ \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma_1 \vdash M_1\ :\ \tau_2 \to \tau \quad \Gamma_2 \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2\ :\ \tau}$$

$$\textsc{Let}\ \frac{\Gamma_1 \vdash M_1\ :\ \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ :\ \tau_2} \qquad \textsc{Gen}\ \frac{\Gamma, \forall\alpha \vdash M\ :\ \tau \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \Lambda\alpha.\,M\ :\ \forall\alpha.\,\tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash M\ :\ \forall\alpha.\,\tau' \qquad \Gamma \vdash \tau\,\mathrm{wf}}{\Gamma \vdash M\ [\tau]\ :\ \tau'\,[\alpha \leftarrow \tau]}$$

$$\textsc{Record-Empty}\ \frac{\Gamma \vdash \mathrm{wf} \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \{\}\ :\ \{\}} \qquad \textsc{Record-Val1}\ \frac{\forall i \in \{1, \ldots, n\}\,,\ \ell \neq \ell'_i \qquad \Gamma \vdash M\ :\ \tau \qquad \Gamma' \vdash \{r\}\ :\ \{(\ell'_i : \tau'_i)^{i \in 1..n}\}}{\Gamma \curlyvee \Gamma' \vdash \{\ell = M\ ;\ r\}\ :\ \{\ell : \tau\ ;\ (\ell'_i : \tau'_i)^{i \in 1..n}\}}$$

$$\textsc{Record-Val2}\ \frac{\forall i \in \{1, \ldots, n\}\,,\ \ell \neq \ell'_i \qquad \Gamma \vdash M\ :\ \tau_0 \qquad \Gamma \vdash \tau_0 \approx \tau \qquad \Gamma' \vdash \{r\}\ :\ \{(\ell'_i : \tau'_i)^{i \in 1..n}\}}{\Gamma \curlyvee \Gamma' \vdash \{\ell\ :\ \tau = M\ ;\ r\}\ :\ \{\ell : \tau\ ;\ (\ell'_i : \tau'_i)^{i \in 1..n}\}} \qquad \textsc{Proj}\ \frac{\Gamma \vdash M\ :\ \{(\ell_i : \tau_i)^{i \in 1..n}\} \qquad 1 \leq k \leq n}{\Gamma \vdash M.\ell_k\ :\ \tau_k}$$

$$\textsc{Exists}\ \frac{\Gamma, \alpha = \tau' \vdash M\ :\ \tau}{\Gamma \vdash \exists(\alpha = \tau')\,M\ :\ \exists\alpha.\,\tau} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash M\ :\ \tau' \qquad \Gamma \vdash \tau' \approx \tau}{\Gamma \vdash (M : \tau)\ :\ \tau} \qquad \textsc{Open}\ \frac{\Gamma \vdash M\ :\ \exists\alpha.\,\tau \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \exists\alpha \vdash \mathsf{open}^{\alpha}\ M\ :\ \tau}$$

$$\textsc{Nu}\ \frac{\Gamma, \exists\alpha \vdash M\ :\ \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\,M\ :\ \tau} \qquad \textsc{Shift}\ \frac{\Gamma' \vdash M\ :\ \tau \qquad \Gamma \Vdash \Gamma'}{\Gamma \vdash M\ :\ \tau}$$

Figure 14. All typing rules of $\mathsf{F}^{\curlyvee}$.

$$\mathsf{let}\ x = v\ \mathsf{in}\ M \ \rightsquigarrow\ M\,[x \leftarrow v] \qquad \text{(REDEX-LET)}$$

$$(\lambda(x{:}\tau)\,M)\ v \ \rightsquigarrow\ \mathsf{let}\ x = v\ \mathsf{in}\ M \qquad \text{(REDEX-APP)}$$

$$(\Lambda\alpha.\,M)\ [\tau] \ \rightsquigarrow\ M\,[\alpha \leftarrow \tau] \qquad \text{(REDEX-INST)}$$

$$\{r_v\}.\ell \ \rightsquigarrow\ \{r_v\} \uparrow \ell \qquad \text{(REDEX-PROJ)}$$

$$(\Sigma^\beta(\alpha = \tau)\,w_1)\ w_2 \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ (w_1\ (w_2\,[\beta \leftarrow \alpha])) \text{ if } \alpha \notin \mathit{ftv}(w_2) \qquad \text{(SIGMA-APP-LEFT)}$$

$$v\ (\Sigma^\beta(\alpha = \tau)\,w) \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ ((v\,[\beta \leftarrow \alpha])\ w) \text{ if } \alpha \notin \mathit{ftv}(v) \qquad \text{(SIGMA-APP-RIGHT)}$$

$$(\Sigma^\beta(\alpha = \tau)\,w)\ [\tau'] \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ (w\ [\tau'\,[\beta \leftarrow \alpha]]) \text{ if } \alpha \notin \mathit{ftv}(\tau') \qquad \text{(SIGMA-INST)}$$

$$(\Sigma^\beta(\alpha = \tau)\,w).\ell \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ (w.\ell) \qquad \text{(SIGMA-PROJ)}$$

$$\{r_v\ ;\ \ell\ :\ \tau'^{?} = \Sigma^\beta(\alpha = \tau)\,w\ ;\ r\} \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\,\{r_v\,[\beta \leftarrow \alpha]\ ;\ \ell\ :\ \tau'^{?}\,[\beta \leftarrow \alpha] = w\ ;\ r\,[\beta \leftarrow \alpha]\} \text{ if } \alpha \notin \mathit{ftv}(r_v) \cup \mathit{ftv}(r) \qquad \text{(SIGMA-RECORD)}$$

$$\mathsf{open}^\gamma\ (\Sigma^\beta(\alpha = \tau)\,w) \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ (\mathsf{open}^\gamma\ w) \text{ if } \gamma \notin \{\alpha, \beta\} \qquad \text{(SIGMA-OPEN)}$$

$$\exists(\gamma = \tau')\,(\Sigma^\beta(\alpha = \tau)\,w) \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau\,[\gamma \leftarrow \tau'])\ (\exists(\gamma = \tau'\,[\beta \leftarrow \alpha])\,w) \text{ if } \gamma \notin \{\alpha, \beta\} \qquad \text{(SIGMA-EXISTS)}$$

$$\nu\gamma.\,\Sigma^\beta(\alpha = \tau)\,w \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\ (\nu\gamma.\,w) \text{ if } \gamma \notin \{\alpha, \beta\} \cup \mathit{ftv}(\tau) \qquad \text{(SIGMA-NU)}$$

$$(\Sigma^\beta(\alpha = \tau)\,w : \tau') \ \rightsquigarrow\ \Sigma^\beta(\alpha = \tau)\,(w : \tau'\,[\beta \leftarrow \alpha]) \text{ if } \alpha \notin \mathit{ftv}(\tau') \qquad \text{(SIGMA-COERCE)}$$

$$(\lambda(x{:}\tau)\,M : \tau_1 \to \tau_2)\ v \ \rightsquigarrow\ ((\lambda(x{:}\tau)\,M)\ (v : \tau) : \tau_2) \qquad \text{(COERCE-APP)}$$

$$(u : \forall\alpha.\,\tau')\ [\tau] \ \rightsquigarrow\ (u\ [\tau] : \tau'\,[\alpha \leftarrow \tau]) \qquad \text{(COERCE-INST)}$$

$$(u : \{(\ell_i : \tau_i)^{i \in 1..n}\}).\ell_k \ \rightsquigarrow\ (u.\ell_k : \tau_k) \text{ if } 1 \le k \le n \qquad \text{(COERCE-PROJ)}$$

$$\mathsf{open}^\alpha\ (u : \exists\alpha.\,\tau) \ \rightsquigarrow\ (\mathsf{open}^\alpha\ u : \tau) \qquad \text{(COERCE-OPEN)}$$

$$((u : \tau) : \tau') \ \rightsquigarrow\ (u : \tau') \qquad \text{(COERCE-COERCE)}$$

$$\nu\beta.\,\Sigma^\beta(\alpha = \tau)\,v \ \rightsquigarrow\ \nu\beta.\,\Sigma^\beta(\alpha = \tau)\,v\,[\alpha \leftarrow \tau] \text{ if } \alpha \in \mathit{ftv}(v) \qquad \text{(NU-SIGMA1)}$$

$$\nu\beta.\,\Sigma^\beta(\alpha = \tau)\,v \ \rightsquigarrow\ v \text{ if } \alpha \notin \mathit{ftv}(v) \qquad \text{(NU-SIGMA2)}$$

These rules are closed under evaluation context and transitivity.

Figure 15. Reduction rules.

$$
\begin{array}{rcl}
Q^\alpha & ::= & \mathsf{open}^\alpha\ M \ \mid\ Q^\alpha\ M \ \mid\ M\ Q^\alpha \ \mid\ Q^\alpha\ [\tau] \ \mid\ \mathsf{pack}\ \langle \tau, Q^\alpha \rangle\ \mathsf{as}\ \exists\beta.\,\tau' \ \mid\ \mathsf{unpack}\ Q^\alpha\ \mathsf{as}\ \beta, x\ \mathsf{in}\ M \\
& \mid & \mathsf{unpack}\ M\ \mathsf{as}\ \beta, x\ \mathsf{in}\ Q^\alpha \quad \text{if } \beta \neq \alpha \ \mid\ \nu\beta.\,Q^\alpha \quad \text{if } \beta \neq \alpha \ \mid\ \mathsf{open}^\beta\ Q^\alpha \quad \text{if } \beta \neq \alpha \ \mid\ Q^\alpha.\ell \\
& \mid & \{r\ ;\ \ell = Q^\alpha\ ;\ r'\} \ \mid\ \mathsf{let}\ x = M\ \mathsf{in}\ Q^\alpha \ \mid\ \mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ M
\end{array}
$$

$$
\begin{array}{rcll}
\nu\alpha.\,\mathsf{open}^\alpha\ M & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ x & \\
\nu\alpha.\,Q^\alpha\ M & \twoheadrightarrow & (\nu\alpha.\,Q^\alpha)\ M & \\
\nu\alpha.\,M\ Q^\alpha & \twoheadrightarrow & M\ (\nu\alpha.\,Q^\alpha) & \\
\nu\alpha.\,Q^\alpha\ [\tau] & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ x\ [\tau] & \\
\nu\alpha.\,\mathsf{pack}\ \langle \tau, Q^\alpha \rangle\ \mathsf{as}\ \exists\beta.\,\tau' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ \mathsf{pack}\ \langle \tau, x \rangle\ \mathsf{as}\ \exists\beta.\,\tau' & \\
\nu\alpha.\,\mathsf{unpack}\ Q^\alpha\ \mathsf{as}\ \beta, y\ \mathsf{in}\ M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ \mathsf{unpack}\ x\ \mathsf{as}\ \beta, y\ \mathsf{in}\ M & \\
\nu\alpha.\,\mathsf{unpack}\ M\ \mathsf{as}\ \beta, x\ \mathsf{in}\ Q^\alpha & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \beta, x\ \mathsf{in}\ \nu\alpha.\,Q^\alpha & \\
\nu\alpha.\,\nu\beta.\,Q^\alpha & \twoheadrightarrow & \nu\beta.\,\nu\alpha.\,Q^\alpha & \text{if } \beta \neq \alpha \\
\nu\alpha.\,\mathsf{open}^\beta\ Q^\alpha & \twoheadrightarrow & \mathsf{open}^\beta\ \nu\alpha.\,Q^\alpha & \text{if } \beta \neq \alpha \\
\nu\alpha.\ (Q^\alpha.\ell) & \twoheadrightarrow & (\nu\alpha.\,Q^\alpha).\ell & \\
\nu\alpha.\,\{r\ ;\ell = Q^\alpha\ ;\ r'\} & \twoheadrightarrow & \{r\ ;\ \ell = \nu\alpha.\,Q^\alpha\ ;\ r'\} & \\
\nu\alpha.\,\mathsf{let}\ x = M\ \mathsf{in}\ Q^\alpha & \twoheadrightarrow & \mathsf{let}\ x = M\ \mathsf{in}\ \nu\alpha.\,Q^\alpha & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}^\alpha\ M\ \mathsf{in}\ M' & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ M\ \mathsf{in}\ M' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ M\ \mathsf{in}\ M' & \text{if } y \notin \mathit{fv}(M, M') \\
\nu\alpha.\,\mathsf{let}\ x = M\ Q^\alpha\ \mathsf{in}\ M' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = M\ y\ \mathsf{in}\ M' & \text{if } y \notin \mathit{fv}(M, M') \\
\nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ [\tau]\ \mathsf{in}\ M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ [\tau]\ \mathsf{in}\ M & \text{if } y \notin \mathit{fv}(M) \\
\nu\alpha.\,\mathsf{let}\ x = & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in} & \text{if } y \notin \mathit{fv}(M) \\
\quad \mathsf{pack}\ \langle \tau, Q^\alpha \rangle\ \mathsf{as}\ \exists\beta.\,\tau'\ \mathsf{in}\ M & & \quad \mathsf{let}\ x = \mathsf{pack}\ \langle \tau, y \rangle\ \mathsf{as}\ \exists\beta.\,\tau'\ \mathsf{in}\ M & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{unpack}\ Q^\alpha\ \mathsf{as}\ \beta, y\ \mathsf{in}\ M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ z = Q^\alpha\ \mathsf{in} & \text{if } z \notin \mathit{fv}(M, M') \\
\quad \mathsf{in}\ M' & & \quad \mathsf{let}\ x = \mathsf{unpack}\ z\ \mathsf{as}\ \beta, y\ \mathsf{in}\ M & \\
& & \quad \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{unpack}\ M\ \mathsf{as}\ \beta, y\ \mathsf{in}\ Q^\alpha & \twoheadrightarrow & \mathsf{unpack}\ M\ \mathsf{as}\ \beta, y\ \mathsf{in} & \text{if } x \neq y \\
\quad \mathsf{in}\ M' & & \quad \nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = \nu\beta.\,Q^\alpha\ \mathsf{in}\ M & \twoheadrightarrow & \nu\beta.\,\nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ M & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}^\beta\ Q^\alpha\ \mathsf{in}\ M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{open}^\beta\ y\ \mathsf{in}\ M & \text{if } y \notin \mathit{fv}(M, M') \\
\nu\alpha.\,\mathsf{let}\ x = Q^\alpha.\ell\ \mathsf{in}\ M & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y.\ell\ \mathsf{in}\ M & \text{if } y \notin \mathit{fv}(M) \\
\nu\alpha.\,\mathsf{let}\ x = \{r\ ;\ \ell = Q^\alpha\ ;\ r'\}\ \mathsf{in} & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in} & \text{if } z_1, z_2 \notin \mathit{fv}(Q^\alpha, r, M) \\
\quad M & & \quad \mathsf{let}\ x = \{r\ ;\ \ell = y\ ;\ r'\}\ \mathsf{in}\ M & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{let}\ y = M\ \mathsf{in}\ Q^\alpha\ \mathsf{in}\ M' & \twoheadrightarrow & \mathsf{let}\ x = M\ \mathsf{in} & \text{if } z \notin \mathit{fv}(M') \\
& & \quad \nu\alpha.\,\mathsf{let}\ z = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{let}\ y = x\ \mathsf{in}\ z\ \mathsf{in}\ M' & \\
\nu\alpha.\,\mathsf{let}\ x = \mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ M\ \mathsf{in}\ M' & \twoheadrightarrow & \nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{let}\ y = x\ \mathsf{in}\ M\ \mathsf{in}\ M' &
\end{array}
$$

Figure 16. From $\mathsf{F}^\forall$ to F, second pass: recovering unpack's.

$$
\lfloor x \rfloor \triangleq x \qquad \lfloor \lambda(x{:}\tau)\,M \rfloor \triangleq \lambda(x)\,\lfloor M \rfloor \qquad \lfloor M\ M' \rfloor \triangleq \lfloor M \rfloor\ \lfloor M' \rfloor \qquad \lfloor \mathsf{let}\ x = M\ \mathsf{in}\ M' \rfloor \triangleq \mathsf{let}\ x = \lfloor M \rfloor\ \mathsf{in}\ \lfloor M' \rfloor
$$

$$
\lfloor \Lambda\alpha.\,M \rfloor \triangleq \lambda(x)\,\lfloor M \rfloor \text{ if } x \notin \mathit{fv}(M) \qquad \lfloor M\ [\tau] \rfloor \triangleq \lfloor M \rfloor\ \{\}
$$

$$
\lfloor \{(\ell_i\ :\ \tau_i^?\ \mathsf{as}\ x_i = M_i)^{i \in 1..n}\} \rfloor \triangleq \{(\ell_i\ \mathsf{as}\ x_i = \lfloor M_i \rfloor)^{i \in 1..n}\} \qquad \lfloor M.\ell \rfloor \triangleq \lfloor M \rfloor.\ell \qquad \lfloor \exists(\alpha = \tau)\,M \rfloor \triangleq \lfloor M \rfloor
$$

$$
\lfloor (M : \tau) \rfloor \triangleq \lfloor M \rfloor \qquad \lfloor \mathsf{open}^\alpha\ M \rfloor \triangleq \lfloor M \rfloor \qquad \lfloor \nu\alpha.\,M \rfloor \triangleq \lfloor M \rfloor
$$

Figure 17. Type-erasure.

$$\textsc{Wf-Var}\ \frac{\Gamma, \alpha \in \sigma, \Gamma' \vdash \text{wf} \qquad \Gamma \vdash \sigma \ :: \ \kappa}{\Gamma, \alpha \in \sigma, \Gamma' \vdash \alpha \ :: \ \kappa} \qquad \textsc{Wf-Arrow}\ \frac{\Gamma \vdash \tau_1 \ :: \ \kappa_1 \qquad \Gamma \vdash \tau_2 \ :: \ \kappa_2}{\Gamma \vdash \tau_1 \to \tau_2 \ :: \ \kappa_1 \to \kappa_2} \qquad \textsc{Wf-Empty}\ \frac{\Gamma \vdash \text{wf}}{\Gamma \vdash \{\,\} :: \star}$$

$$\textsc{Wf-Record}\ \frac{1 \leq n \qquad \forall i \in \{1..n\}, \quad \Gamma \vdash \tau_i \ :: \ \kappa_i \qquad \forall i, j, i \neq j \Rightarrow \ell_i \neq \ell_j}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \ :: \ \{(\ell_i : \kappa_i)^{i \in 1..n}\}} \qquad \textsc{Wf-Proj-Label}\ \frac{\Gamma \vdash \tau \ :: \ \{(\ell_i : \kappa_i)^{i \in 1..n}\} \qquad 1 \leq k \leq n}{\Gamma \vdash \tau.\ell_k \ :: \ \kappa_k} \qquad \textsc{Wf-Proj-Arrow}\ \frac{\Gamma \vdash \tau \ :: \ \kappa_1 \to \kappa_2}{\Gamma \vdash \tau.i \ :: \ \kappa_i}$$

$$\textsc{Wf-Forall}\ \frac{\Gamma \vdash \sigma \ :: \ \kappa \qquad \Gamma, \alpha \in \sigma \vdash \tau \ :: \ \kappa'}{\Gamma \vdash \forall (\alpha \in \sigma)\, \tau \ :: \ \star} \qquad \textsc{Wf-Exists}\ \frac{\Gamma \vdash \sigma \ :: \ \kappa \qquad \Gamma, \alpha \in \sigma \vdash \tau \ :: \ \kappa'}{\Gamma \vdash \exists\, (\alpha \in \sigma)\, \tau \ :: \ \star}$$

Figure 18. Wellformed types.

$$\textsc{Wf-Top}\ \frac{\Gamma \vdash \text{wf}}{\Gamma \vdash \top \ :: \ \star} \qquad \textsc{Wf-Single}\ \frac{\Gamma \vdash \tau \ :: \ \kappa}{\Gamma \vdash \tau \ :: \ \kappa} \qquad \textsc{Wf-Shape}\ \frac{\Gamma \vdash \sigma \ :: \ \kappa \qquad \Gamma, \alpha \in \sigma \vdash \sigma' \ :: \ \kappa'}{\Gamma \vdash [\alpha \in \sigma]\ \sigma' \ :: \ \kappa'}$$

Figure 19. Wellformed shapes.

$$\textsc{Wf-Empty}\ \frac{}{\varepsilon \vdash \text{wf}} \qquad \textsc{Wf-Quantifier}\ \frac{\Gamma \vdash \text{wf} \qquad \Gamma \vdash \sigma \ :: \ \kappa \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \alpha \in \sigma \vdash \text{wf}} \qquad \textsc{Wf-Eq}\ \frac{\Gamma \vdash \text{wf} \qquad \Gamma \vdash \sigma \ :: \ \kappa \qquad \Gamma \vdash \tau \ :: \ \kappa' \qquad \Gamma \vdash \tau \ \in \ \sigma \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \alpha \in \sigma = \tau \vdash \text{wf}} \qquad \textsc{Wf-Var}\ \frac{\Gamma \vdash \text{wf} \qquad \Gamma \vdash \tau \ :: \ \kappa \qquad x \notin \mathit{dom}\,\Gamma}{\Gamma, x : \tau \vdash \text{wf}}$$

Figure 20. Wellformed environments.

$$\in\textsc{-Top}\ \frac{\Gamma \vdash \tau \ :: \ \kappa}{\Gamma \vdash \tau \ \in \ \top} \qquad \in\textsc{-Single}\ \frac{\Gamma \vdash \tau \ :: \ \kappa}{\Gamma \vdash \tau \ \in \ \tau} \qquad \in\textsc{-Shape}\ \frac{\Gamma \vdash \tau \ \in \ \sigma \qquad \Gamma, \alpha \in \sigma \vdash \tau' \ \in \ \sigma'}{\Gamma \vdash \tau'\,[\alpha \leftarrow \tau] \ \in \ [\alpha \in \sigma]\ \sigma'}$$

$$\in\textsc{-Equiv}\ \frac{\Gamma \vdash \tau' \ \in \ \sigma \qquad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash \tau \ \in \ \sigma} \qquad \in\textsc{-Fold}\ \frac{\alpha_1 \notin \mathit{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathit{ftv}(\Gamma_2, \tau, \sigma) \qquad \Gamma_1, \alpha_2 \in \sigma_2, \alpha_1 \in \sigma_1, \Gamma_2 \vdash \tau \ \in \ \sigma}{\Gamma_1, \alpha_1 \in [\alpha_2 \in \sigma_2]\ \sigma_1, \Gamma_2 \vdash \tau \ \in \ \sigma}$$

Figure 21. Shapes membership.

$$\textsc{Equi-Var-Single}\ \frac{\Gamma \vdash \text{wf} \qquad \alpha \in \tau \in \Gamma}{\Gamma \vdash \alpha \equiv \tau} \qquad \textsc{Equi-Record}\ \frac{\forall i \in \{1..n\}, \quad \Gamma \vdash \tau_i \equiv \tau_i'}{\Gamma \vdash \{(\ell_i : \tau_i)\}^{i \in 1..n} \equiv \{(\ell_i : \tau_i')\}^{i \in 1..n}} \qquad \textsc{Equi-Proj-Label}\ \frac{\Gamma \vdash \tau \equiv \{(\ell_i : \tau_i)^{i \in 1..n}\}}{\Gamma \vdash \tau.k \equiv \tau_k} \qquad \textsc{Equi-Proj-Arrow}\ \frac{\Gamma \vdash \tau \equiv \tau_1 \to \tau_2}{\Gamma \vdash \tau.i \equiv \tau_i}$$

$$\textsc{Equi-Forall}\ \frac{\Gamma \vdash \sigma \equiv \sigma' \qquad \Gamma, \alpha :: \kappa \in \sigma \vdash \tau \equiv \tau'}{\Gamma \vdash \forall (\alpha \in \sigma)\, \tau \equiv \forall (\alpha \in \sigma')\, \tau'} \qquad \textsc{Equi-Exists}\ \frac{\Gamma \vdash \sigma \equiv \sigma' \qquad \Gamma, \alpha :: \kappa \in \sigma \vdash \tau \equiv \tau'}{\Gamma \vdash \exists\, (\alpha \in \sigma)\, \tau \equiv \exists\, (\alpha \in \sigma')\, \tau'}$$

$$\textsc{Equi-Fold}\ \frac{\alpha_1 \notin \mathit{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathit{ftv}(\Gamma_2, \tau, \tau') \qquad \Gamma_1, \alpha_2 \in \sigma_2, \alpha_1 \in \sigma_1, \Gamma_2 \vdash \tau \equiv \tau'}{\Gamma_1, \alpha_1 \in [\alpha_2 \in \sigma_2]\ \sigma_1, \Gamma_2 \vdash \tau \equiv \tau'} \qquad \textsc{Equi-Refl}\ \frac{\Gamma \vdash \tau \ :: \ \kappa}{\Gamma \vdash \tau \equiv \tau} \qquad \textsc{Equi-Sym}\ \frac{\Gamma \vdash \tau_2 \equiv \tau_1}{\Gamma \vdash \tau_1 \equiv \tau_2} \qquad \textsc{Equi-Trans}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma \vdash \tau_2 \equiv \tau_3}{\Gamma \vdash \tau_1 \equiv \tau_3}$$

Figure 22. Equivalent types.

$$\frac{\Gamma \vdash \sigma_1 \sqsubseteq \sigma_2 \qquad \Gamma \vdash \sigma_2 \sqsubseteq \sigma_1}{\Gamma \vdash \sigma_1 \equiv \sigma_2} \qquad\qquad \frac{\Gamma, \alpha \in \sigma \vdash \alpha \in \sigma'}{\Gamma \vdash \sigma \sqsubseteq \sigma'}$$

Figure 23. Equivalent shapes.

VAR
$$\frac{\Gamma \vdash \text{wf} \quad \Gamma \textit{ pure}}{\Gamma \vdash x : \Gamma(x)}$$

LAM
$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2 \qquad \Gamma \textit{ pure}}{\Gamma \vdash \lambda(x{:}\tau_1)\, M : \tau_1 \to \tau_2}$$

APP
$$\frac{\Gamma_1 \vdash M_1 : \tau_2 \to \tau \quad \Gamma_2 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2 : \tau}$$

LET
$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

GEN
$$\frac{\Gamma, \forall\,(\alpha \in \sigma) \vdash M : \tau \qquad \Gamma \textit{ pure}}{\Gamma \vdash \Lambda(\alpha \in \sigma)\, M : \forall(\alpha \in \sigma)\, \tau}$$

INST
$$\frac{\Gamma \vdash M : \forall(\alpha \in \sigma)\, \tau' \qquad \Gamma \vdash \tau :: \kappa \qquad \Gamma \vdash \tau \in \sigma}{\Gamma \vdash M\ [\tau] : \tau'\,[\alpha \leftarrow \tau]}$$

RECORD-EMPTY
$$\frac{\Gamma \vdash \text{wf} \qquad \Gamma \textit{ pure}}{\Gamma \vdash \{\} : \{\}}$$

RECORD-VAL1
$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\},\ \ell \neq \ell'_i \\ \Gamma \vdash M : \tau \qquad \Gamma' \vdash \{r\} : \{(\ell'_i : \tau'_i)^{i \in 1..n}\}\end{array}}{\Gamma \curlyvee \Gamma' \vdash \{\ell = M\ ;\ r\} : \{\ell : \tau\ ;\ (\ell'_i : \tau'_i)^{i \in 1..n}\}}$$

RECORD-VAL2
$$\frac{\begin{array}{c}\forall i \in \{1, \ldots, n\},\ \ell \neq \ell'_i \qquad \Gamma \vdash M : \tau_0 \\ \Gamma \vdash \tau_0 \approx \tau \qquad \Gamma' \vdash \{r\} : \{(\ell'_i : \tau'_i)^{i \in 1..n}\}\end{array}}{\Gamma \curlyvee \Gamma' \vdash \{\ell\ :\ \tau = M\ ;\ r\} : \{\ell : \tau\ ;\ (\ell'_i : \tau'_i)^{i \in 1..n}\}}$$

PROJ
$$\frac{\Gamma \vdash M : \{(\ell_i : \tau_i)^{i \in 1..n}\} \qquad 1 \leq k \leq n}{\Gamma \vdash M.\ell_k : \tau_k}$$

EXISTS
$$\frac{\Gamma, \alpha \in \sigma = \tau' \vdash M : \tau}{\Gamma \vdash \exists(\alpha \in \sigma = \tau')\, M : \exists\,(\alpha \in \sigma)\, \tau}$$

COERCE
$$\frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \approx \tau}{\Gamma \vdash (M : \tau) : \tau}$$

OPEN
$$\frac{\Gamma \vdash M : \exists\,(\alpha \in \sigma)\, \tau \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, \exists\,(\alpha \in \sigma) \vdash \mathsf{open}^{\alpha}\ M : \tau}$$

NU
$$\frac{\Gamma, \exists\,(\alpha \in \sigma) \vdash M : \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\,(\alpha \in \sigma)\, M : \tau}$$

SHIFT
$$\frac{\Gamma' \vdash M : \tau \qquad \Gamma \Vdash \Gamma'}{\Gamma \vdash M : \tau}$$

EQUIV
$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash M : \tau'}$$

Figure 24. $\mathsf{F}^{\curlyvee}$ with paths: all typing rules. Differences with $\mathsf{F}^{\curlyvee}$ appear in a gray shade.