Modeling Abstract Types in Modules with Open Existential Types

Benoît Montagu Didier Rémy
INRIA
{Benoit.Montagu, Didier.Remy}@inria.fr

Abstract

We propose $\mathsf{F}^{\curlyvee}$, a *calculus* of *open* existential types that is an extension of System F obtained by decomposing the introduction and elimination of existential types into more atomic constructs. Open existential types model *modular* type abstraction as done in module systems. The static semantics of $\mathsf{F}^{\curlyvee}$ adapts standard techniques to deal with linearity of typing contexts, its dynamic semantics is a small-step reduction semantics that performs extrusion of type abstraction as needed during reduction, and the two are related by subject reduction and progress lemmas. Applying the Curry-Howard isomorphism, $\mathsf{F}^{\curlyvee}$ can be also read back as a logic with the same expressive power as second-order logic but with more modular ways of assembling partial proofs. We also extend the core calculus to handle the double vision problem as well as type-level and term-level recursion. The resulting language turns out to be a new formalization of (a minor variant of) Dreyer's internal language for recursive and mixin modules.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features — Abstract data types, Modules; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic —Lambda calculus and related systems.

General Terms Design, Languages, Theory

Keywords Lambda-Calculus, Modules, Type systems, Abstract types, Generativity, Existential Types, Linear type systems, Modularity.

1. Introduction

Modularity has always been the key to robust, manageable, and maintainable large software. It is even more so as the size and complexity of software keeps increasing. 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 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 (10), 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, while the successful history of the ML module system shows a relative ease of use, at least for the most frequent cases, its metatheoretical study has probably been one of the most difficult parts of the ML language. Even with today's state-of-the-art technology, it is still usually considered involved. The discrepancy between the intuitive, perhaps misleading intelligibility of modules and their intricate formal description is inconveniently surprising.

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 as used in modules. Since then, an abstract type has been considered as a type whose definition has been forgotten. Consequently, an abstract type cannot be referred to by *how* it is defined; instead, it is referred to by *where* it is defined, *i.e.* as a projection *path* from a value variable bound to the module where it is defined. Two decades later, *paths* are still at the foundation of the ML-based module systems in use (11; 13; 19), and of more recent designs such as Scala (16).

Unfortunately, this formalization is still the source of a major difficulty with module systems: because types appear as components of module expressions, abstract types, which are designed by paths, syntactically depend on values, which pulls into the language all the complexity of dependent types. In fact, an important property of a module system called the *phase distinction* (7), which states that types should not *dynamically* depend on values and so ensures static typechecking and permits separate compilation, appears to be in contradiction with the use of dependent types.

Hence, a large amount of work has been dedicated to showing that dependencies used in module systems are in fact purely static, and that types used to formalize them are not *truly* dependent on values[1]. This restriction is now well understood and has been elegantly formalized using singleton kinds (4; 22) to capture the absence of dynamic dependencies. Among other benefits, this line of work treats type abstraction as subtyping, is highly expressive, and rather close to ML. Moreover, its metatheory has been formalized in Twelf (9), and thus mechanically verified. The robustness of the approach has also been demonstrated by adapting the framework to model type abstraction in a distributed setting (17).

However, despite these many positive results, there remain several drawbacks with this approach—or with the ML module system. At the source of all difficulties, the tension between the presence of type components in values and the phase distinction is still present: in fact much of the formalism sophistication is for ensuring that types do not actually depend on values. One tech-

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

[1] There seems to be no name agreement yet on how to call this very limited form of dependent types.

nical difficulty, known as the *avoidance problem* (8), forces much more type annotations in source expressions than what would otherwise be necessary. As a consequence, the internal calculus of modules, which has a clean and standard mathematical formalization is mostly used as an internal language for a surface language with a rather sophisticated elaboration mechanism, which therefore does not inherit the properties of the internal calculus. While quite general and expressive, this modeling of modules is perhaps further from the programmers intuitions than other aspects of the language. In addition, this approach does not seem to easily accommodate to recursive or mixin modules.

These arguments are as many invitations to pursue the investigation for finding alternative explanations of modules. Hopefully, it should be conceptually economical and should more closely reflect the intuitive simplicity of the underlying mechanisms: our goal is not, at least in a first step, to increase expressiveness.

Interestingly, in the purpose of explaining type abstraction and generativity for recursive modules and solving the *double vision problem*, Dreyer introduced an internal language, called RTG, as a target of the elaboration for a surface language of recursive modules (2; 3). Remarkably, the elaboration process is rather simple and compositional, while the internal language provides neither type components nor dependent types. Hence, although this does not seem to have been Dreyer's goal, it appears as a corollary that these two ingredients do not seem to be necessary to model modules in an accurate and easy manner. This is a great motivation to pursue investigation in this direction.

A closer look at RTG shows an expressive but intriguing set of primitives that allows to create undefined type references and later assign a definition to them: "new α in M" introduces a type reference named α in the scope of M that should be set at most once, with the type reference update "set $\alpha := \tau$ in $M : \tau'$". Then, M and only M will see the concrete definition τ for α while other parts of the program will see α abstractly. In this way, RTG can handle two views for a given type variable: an abstract one, without definition, and a concrete one, equipped with a definition.

RTG is obviously quite expressive, since it can be used to model recursive modules. However, its static semantics is surprising and somewhat *ad hoc*: its typechecking rules use *assignments* in a global store to keep track of type definitions, which makes the system non compositional and unnecessarily asymmetrical, and deviates from the traditional presentation of typing rules. The dynamic semantics of RTG is also given through an abstract machine that carries a global *type store* to record type assignments in an imperative manner, unnecessarily enforces a deterministic evaluation order, and confuses the uses of assignment for building recursive values and for modeling type abstraction. Thus, although primitives of RTG *seem* to be adequately chosen, the presentation of its static and dynamic semantics raises questions on the fundamental reasons of its suitability and therefore turns RTG away from the status and potential applications that it really deserves: a truly interesting calculus that could be used not only as an internal language, but hopefully also as the core of a surface language in which modules could directly be programmed.

Our goal in this work is to give an alternate presentation of a minor variant of RTG, that attempts to improve its foundations and metatheoretical properties, and that hopefully gives a justification for its set of primitives. Thus, like RTG, our system focuses on type abstraction only: the study of other features, such as translucent signatures or sharing constraints, is deferred to future work.

We first present core $\mathsf{F}^{\curlyvee}$, a *calculus* of *open* existential types (§2) without recursion, that is obtained by decomposing the usual introduction and elimination constructs for existential types into more atomic ones. We observe that $\mathsf{F}^{\curlyvee}$ already permits *modular* type abstraction, without any recursion mechanism.

We use known techniques to deal with linearity of typing contexts (§3) instead of RTG's static effect calculus, thus keeping a symmetric and compositional presentation (§3.3). We give $\mathsf{F}^{\curlyvee}$ a traditional small-step reduction semantics (§3.4) that expresses the inherent notion of sharing behind generativity through the *extrusion* of some binder. This permits to finely trace abstraction during reduction. The static and dynamic semantics of $\mathsf{F}^{\curlyvee}$ are related by subject reduction and progress lemmas (§3.5).

Thanks to the Curry-Howard isomorphism, $\mathsf{F}^{\curlyvee}$ can be read back as a logic with the same expressive power as second-order logic but with more modular ways of assembling partial proofs (§3.6), and in which the essence of ML modules can already be modeled.

We build a series of conservative extensions (§4) on top of this core to increase its expressiveness with, namely, more liberal non-recursive type definitions (§4.1), a solution to the double vision problem (§4.2), mutually recursive type equations (§4.3), and term-level recursion (§4.4). A crucial point is that these extensions are independent from each other.

We end with a discussion of related work (§5) and concluding remarks.

2. Open existential types

2.1 Abstract types as existential types

Mitchell and Plotkin (14) showed that abstract types could be understood as existential types. However, it has also been noticed that existential types do not *accurately* model type abstraction in modules, because they 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 $\exists\alpha.\,\tau'$.

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

Existential types are eliminated by the unpack construct: provided M has type $\exists\alpha.\,\tau$, the expression $\mathsf{unpack}\ M\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ M'$ 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.

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

From now on, we assume that System F is equipped with records and with the above primitive constructs, although they could also be provided as a well-known syntactic sugar (18).

2.2 Atomic constructs for existential types

In this section we split off the constructs for existential types. Indeed, both pack and unpack have modularity problems.

The crucial issue 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. The reason is that unpack is doing too many things at the same time: opening the existential type, binding the opened value to a variable, and restricting the scope of the fresh type variable.

The problem with pack is mostly *verbosity*: it requires to completely specify the resulting type, thus duplicating type information in the parts that have not been abstracted away. This can be annoying when hiding only a small part of a term, whereas this term has a very long type. 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 abstracted away under that abstract name.

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 pieces, recovering modularity while preserving expressiveness of existential types. To achieve this decomposition, we first need to enrich typing environments with new items.

2.2.1 Richer contexts for typing judgments

The contexts of typing judgments in System F are sequences of items, where 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 $\exists\alpha$ to keep track of the scope of (open) abstract types, and type definitions $\forall(\alpha = \tau)$ to concisely mediate between the abstract and concrete views of types. That is, typing environments are as follows:

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

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. It is sensible to consider them as Skolem's constants and to understand type definition bindings as explicit type substitutions. For the moment, we consider environments as sequences modulo reordering of independent items. Their structure will be enriched again in §4.1.

We define the domain of a binding as follows:

$$\mathit{dom}(x : \tau) \triangleq x$$
$$\mathit{dom}(\forall\alpha) \triangleq \mathit{dom}(\forall(\alpha = \tau)) \triangleq \mathit{dom}(\exists\alpha) \triangleq \alpha$$

The domain of an environment is, as usual, the union of the domains of the bindings it contains. In addition, we may use the following notations for specific domains:

$$\mathit{dom}^{=}\,\Gamma \triangleq \{\alpha \mid \forall(\alpha = \tau) \in \Gamma\}$$
$$\mathit{dom}^{\forall}\,\Gamma \triangleq \{\alpha \mid \forall\alpha \in \Gamma\} \quad \mathit{dom}^{\exists}\,\Gamma \triangleq \{\alpha \mid \exists\alpha \in \Gamma\}$$

2.2.2 Splitting unpack

We 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* $\mathsf{open}\ \langle\alpha\rangle\ M$ 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.

$$\textsc{Open}\ \frac{\Gamma \vdash M : \exists\alpha.\,\tau}{\Gamma, \exists\alpha \vdash \mathsf{open}\ \langle\alpha\rangle\ M : \tau}$$

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 $\mathsf{open}\ \langle\alpha\rangle\ M'$ expression occurring within M.

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

As with RTG, one may recover unpack as syntactic sugar:

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

This makes explicit the simultaneous operations performed by unpack, which turns out not to be atomic at all: 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 needs not (always) be used anymore. The abstract type α may now be introduced at the outermost level or given by the typing context and freely made available to the whole program.

2.2.3 Splitting pack

We replace pack with three orthogonal constructs: *existential introduction*, which creates an existential type, *open witness definition*, which introduces a type witness and gives it a name, and *coercion*, which determines which parts of types are to be hidden. We present this separation in two stages: first, we separate the (closed) definition of a witness from the information of which parts are abstracted away; then, we split the definition of a witness again into two pieces that introduce an existential quantifier and the witness, separately.

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

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

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

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

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

$$\mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as}\ \exists\alpha.\,\tau' \triangleq \exists(\alpha = \tau)\,(M : \tau') \quad \text{if } \alpha \notin \mathit{ftv}(M)$$

However, the description of what is being hidden can now be separated from the action of hiding, which avoids repeating some type information. Hence, it makes the creation of existential values, shorter, thus easier, and more maintainable. Indeed, it allows for putting the information of hiding parts of a type deeply inside a term, like in the following record, in which 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 of the record had been abstracted independently.

To complete the separation, we now split $\exists(\alpha = \tau)\, M$ further. The *existential introduction* $\exists\alpha.\, M$ introduces an existential type variable in the environment while typing M, and makes α existentially bound in the resulting type. This is the exact counterpart of the open construct.

$$\textsc{Exists}\quad \frac{\Gamma, \exists\alpha \vdash M\ :\ \tau}{\Gamma \vdash \exists\alpha.\, M\ :\ \exists\alpha.\, \tau}$$

The *open witness definition* $\Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M$ introduces the witness τ for the type variable α: similarly to what is done for $\exists(\alpha = \tau)\, M$, the equation $\forall(\alpha = \tau)$ is added to the context while typing M. In addition, an external name β is provided, in the same way as for the open construct. The internal name α and its equation are only reachable internally, but the witness is denoted externally by the abstract type variable β. The resulting type does not mention the internal name, since it has been substituted for the external one. In other words, the witness definition defines *a frontier between a concrete internal world and an abstract external one*. To keep the system sound, we ensure that a unique witness is hidden behind an external name, hence the use of an existential resource. The typing rule will be refined later to handle the double vision problem.

$$\textsc{Sigma}\quad \frac{\Gamma, \forall(\alpha = \tau) \vdash M\ :\ \tau'}{\Gamma, \exists\beta \vdash \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M\ :\ \tau'[\alpha \leftarrow \beta]}$$

Again, the split construct $\exists(\alpha = \tau)\, M$ can be recovered by the following syntactic sugar:

$$\exists(\alpha = \tau)\, M\ \triangleq\ \exists\beta.\, \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\, M \quad \text{if } \beta \notin \mathit{ftv}(\tau, M)$$

It is worth noting that the *open witness definition* corresponds to type abstraction as it is currently done in module languages: a type definition is kept hidden for the outer environment and a type name is generated so that we can refer to it without knowing its concrete definition. Usual existential types are recovered by closing the open witness definition, *i.e.* by hiding the external name for the witness.

As an example, the following piece of program, written in an ML-like syntax, defines an abstract module of integers:

$$
\begin{array}{l}
\mathsf{module}\ X : \mathsf{sig}\ \ \mathsf{type}\ t \quad \mathsf{val}\ z : t \quad \mathsf{val}\ s : t \to t \quad \mathsf{end} = \\
\quad \mathsf{struct}\ \mathsf{type}\ t = \mathit{int} \quad \mathsf{val}\ z = 0 \quad \mathsf{val}\ s = \lambda(x : \mathit{int})\, x + 1\ \mathsf{end}
\end{array}
$$

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 $\mathsf{F}^{\forall}$ is defined hereafter:

$$
\begin{array}{l}
\Sigma\ \langle\beta\rangle\ (\alpha = \mathit{int}) \\
\quad (\{z = 0\ ;\ s = \lambda(x : \mathit{int})\, x + 1\} : \{z : \alpha\ ;\ s : \alpha \to \alpha\})
\end{array}
$$

The two pieces of code look similar, except for the fact that the signature ascription has been replaced with an open witness definition. 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.

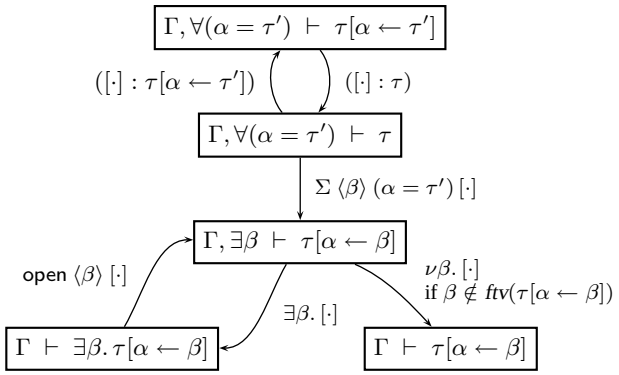

Figure 1. Open existential constructs

Notice that it is also possible to rewrite this program in two parts, by first creating an existential term and then opening it under the name β.

$$
\begin{array}{l}
\mathsf{let}\ x = \\
\quad \exists(\alpha = \mathit{int}) \\
\qquad (\{z = 0\ ;\ s = \lambda(x : \mathit{int})\, x + 1\} : \{z : \alpha\ ;\ s : \alpha \to \alpha\})\ \mathsf{in} \\
\mathsf{open}\ \langle\beta\rangle\ x
\end{array}
$$

It has essentially the same effect: in fact the latter will reduce to the former. It shows however that the mechanisms for type abstraction and opening of existentials are the same.

2.2.4 Generative functors

Following Russo (21), generative functors are functions that have a type of the form $\forall\alpha.\, (\tau_1 \to \exists\beta.\, \tau_2)$. In ML, generativity is *implicitly* released when the functor is applied. In $\mathsf{F}^{\forall}$, however, the result of the function must be *explicitly opened*, because generativity and evaluation are two separate notions. To get the same result with another fresh type, it suffices to open it again under another name.

2.2.5 A summary of the constructs for open existential types

The different constructs introduced for open existential types are gathered on the diagram of Fig.1, which describes their impact on both the typing environment and the resulting type. To increase readability, terms are not printed on the judgments.

The topmost judgment corresponds to a concrete program (of type $\tau[\alpha \leftarrow \tau']$) with an equation $\forall(\alpha = \tau')$ in its environment. With the use of coercions one can mediate to a type τ where the equation has been folded and then go back to the concrete version. Then, using a Σ, we can make the witness abstract by removing the definition from the typing environment and using the external name β instead. In this process, the variable β is marked as existential and the internal name is replaced with the external one. If the external name does not occur free in the resulting type, we can remove the existential item from the environment, without changing the type, to get the bottom right judgment. If this is not the case, we can close the type by transferring the existential quantifier to the type (bottom left judgment). We can then go back by re-opening the existential.

2.2.6 Linearity to control openings and open witness definitions

As openings and open witness definitions use abstract names given by the environment, one must be careful to avoid "abstraction

$$b \curlyvee b = b \text{ if } b \neq \exists\alpha \quad \exists\alpha \curlyvee \forall\alpha = \exists\alpha \quad \forall\alpha \curlyvee \exists\alpha = \exists\alpha$$

Figure 2. Zipping of bindings (preliminary definition).

capture", as in the following (ill-typed) example.

$$\begin{array}{l}\mathsf{let}\ f = \Sigma\,\langle\beta\rangle\ (\alpha = \mathit{int})\ (\lambda(z : \mathit{int})\ z + 1 : \alpha \to \alpha)\ \mathsf{in}\\ \mathsf{let}\ x = \Sigma\,\langle\beta\rangle\ (\alpha = \mathit{bool})\ (\mathit{true} : \alpha)\ \mathsf{in}\ f\ x\end{array}$$

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

To prevent abstraction capture, it suffices that *every name β be used in exactly one opening or open witness definition under name β*. This may be achieved by treating the existential items of the typing environment in a *linear* way. As usual in the literature, 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. We give in Fig.2 and in this paragraph a preliminary definition of zipping to convey the intuition. It will be completed later. 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 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 the witness. Therefore the other side *must* consider the choice of the witness as external, which is why it is given the item $\forall\alpha$.

Note that an equivalent presentation, using two contexts, one of them being linear, is also feasible. However, the current presentation makes extensions easier to define.

2.3 The appearance of recursive types

The above idea of zipping is unfortunately too generous: it makes recursive types appear naturally without any control. Indeed 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 given. Recursive types can appear through type abstraction, *i.e.* through openings or open witness definitions, in two ways.

We call *internal recursion* the first way, which is highlighted by the following example:

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

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 $\mathsf{open}\ \langle\beta\rangle\ \exists(\alpha = \beta \to \beta)\ M$, which leads us to the recursive equation $\beta = \beta \to \beta$.

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

$$\begin{array}{l}\{\ \ell_1 = \Sigma\,\langle\beta_1\rangle\ (\alpha_1 = \beta_2 \to \beta_2)\ M_1\ ;\\ \ \ \ell_2 = \Sigma\,\langle\beta_2\rangle\ (\alpha_2 = \beta_1 \to \beta_1)\ M_2\ \}\end{array}$$

The above code is a pair whose components have been abstracted away and the witnesses are mutually dependent. If we remove the type abstractions we get the recursive equation system $\beta_1 = \beta_2 \to \beta_2$ and $\beta_2 = \beta_1 \to \beta_1$.

Notice that recursive types never arise when using System F's unpack. Consider the following piece of code, where C_1 and C_2 denote contexts:

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

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. By splitting unpack, however, this restriction has been waived.

$$\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau \to \tau \mid \{(\ell_i : \tau_i)^{i\in 1..n}\} & (\textit{Types})\\
 & \mid & \forall\alpha.\,\tau \mid \exists\alpha.\,\tau & \\
M & ::= & x \mid \lambda(x : \tau)\,M \mid M\ M & (\textit{Terms})\\
 & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M \mid \Lambda\alpha.\,M \mid M\,\tau & \\
 & \mid & \{(\ell_i = M_i)^{i\in 1..n}\} \mid M.\ell & \\
 & \mid & \exists\alpha.\,M \mid \Sigma\,\langle\beta\rangle\ (\alpha = \tau)\,M \mid (M : \tau) & \\
 & \mid & \mathsf{open}\ \langle\alpha\rangle\ M \mid \nu\alpha.\,M & \\
v & ::= & u \mid (u : \tau) & (\textit{Values})\\
u & ::= & x \mid \lambda(x : \tau)\,M \mid \Lambda\alpha.\,M & (\textit{Pre-values})\\
 & \mid & \{(\ell_i = v_i)^{i\in 1..n}\} \mid \exists\beta.\,\Sigma\,\langle\beta\rangle\ (\alpha = \tau)\,v & \\
w & ::= & v \mid \Sigma\,\langle\beta\rangle\ (\alpha = \tau)\,w & (\textit{Results})
\end{array}$$

Figure 3. Syntax: types, terms, values, and results.

3. Core F$^{\curlyvee}$

We now present the core of our system, which prevents the appearance of recursive types in a simple manner. We present its semantics and show that its expressive power corresponds exactly to the one of System F. The translation used for this purpose brings interesting insight on the gain of modularity that F$^{\curlyvee}$ achieves.

3.1 A more restrictive zipping

The zipping we defined above is too liberal in the sense that the introduction of abstract types does not follow the scope of term variables, but this can be enforced again. Hence, we define a special zipping, written $\curlyvee'$, specialized for the let rule, that requires that, if $\Gamma_1 \curlyvee' \Gamma_2$ is defined and if $\exists\alpha$ appears in Γ_2, then $\forall\alpha$ must *not* appear in Γ_1, while, if $\exists\alpha$ appears in Γ_1, then $\forall\alpha$ should also appears in Γ_2, as before. Zipping for the other rules $\curlyvee$ is symmetric and requires that if $\exists\alpha$ appears on one side, then $\forall\alpha$ must not be present on the other side. This restriction easily permits to reproduce the usage of type variables in System F, while keeping the flexibility of our constructs.

3.2 Syntax

The language F$^{\curlyvee}$ is based on the explicitly typed version of System F with records and is extended with constructs of §2.2. Types and terms are described in Fig.3.

As open existentials do not introduce new forms of types, types of 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 a label and a type. Type wellformedness is defined as usual.

Terms of F$^{\curlyvee}$ are variables, functions (whose arguments are explicitly typed), applications, let-bindings, type generalizations and applications, introductions and projections of records, and the five constructs for open existentials described above: existential introductions, open witness definitions, coercions, openings, and restrictions. 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.

For conciseness, we also use the following syntactic sugar in technical developments for *closed witness definitions* :

$$\exists(\alpha = \tau)\,M \ \triangleq\ \exists\beta.\,\Sigma\,\langle\beta\rangle\ (\alpha = \tau)\,M \quad \text{if } \beta \notin \mathit{ftv}(\tau, M)$$

We write $\mathit{ftv}(\tau)$ (respectively $\mathit{ftv}(M)$) to denote the set of free type variables of a type τ (respectively a term M).

3.3 Typing rules

Typing rules for open existentials have already been presented in §2.2. The remaining typing rules (Fig.7) are as in System F with two small differences: first, as mentioned above, typing rules with several typing judgments as premises use zipping instead of equality to relate their typing environments. This is the case of Rules App, Let, and Record. Second, typing rules must also ensure that values can be substituted without breaking linearity, which is the case when the typing environment does not contain existential items.

Definition 1. When $\mathit{dom}^{\exists}\,\Gamma$ is empty, we say that Γ is pure and write Γ *pure*. □

This condition appears as an additional premise of typing rules of expressions that are also values (namely, Rules Var, Lam, Gen, and Empty). Purity will be used and explained in more details in §3.4.

Because Rule Open makes the environment decrease (if it is read bottom-up), the property of weakening is *not* provable in its whole generality: one can only weaken a judgment by a non-linear item that does not depend on linear items. This is sufficient for the proof of soundness. A primitive weakening rule will be added when considering extensions of core $\mathsf{F}^{\forall}$.

3.4 Reduction semantics

The language $\mathsf{F}^{\forall}$ 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 and open witness definitions. 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)**.** *Assume that* $\Gamma \vdash M : \tau$ *and* $\Gamma', x : \tau, \Gamma'' \vdash M' : \tau'$ *hold, where* Γ *is pure and* $\Gamma \between \Gamma'$ *is well defined. Then* $(\Gamma \between \Gamma'), \Gamma'' \vdash M'[x \leftarrow M] : \tau'$ *also holds.*

Therefore, values are substitutable if we restrict them to pure terms. But conversely, every irreducible term is *not necessarily* 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 $\mathsf{F}^{\forall}$, results also include terms such as $\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\lambda(x : \alpha)\,x$, 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.3. They are either pre-values or coerced pre-values, where pre-values are variables, functions, generalizations, records of values or existential values. Note that nested coercions are not values—they must be further reduced. Note also that no evaluation takes place under λs or Λs. Finally, results are values preceded by a (possibly empty) sequence of Σs.

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* $\Gamma \vdash v : \tau$ *holds, then* Γ *is pure.*

Extrusions Values are substitutable, but some results are not values, namely a sequence of Σs prefixing a value. How can we handle these results, when they ought to be substituted, without breaking

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

Figure 4. Example of extrusion.

linearity? Our solution is to extrude the Σs *just enough* to expose and perform the next reduction step.

For example, consider the reduction steps on Fig.4. The initial expression is a let-binding of the form $\mathsf{let}\ x = w\ \mathsf{in}\ M$ where w is the result form $\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,(1:\alpha)$. 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, thus 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, in which the external name β occurs. Therefore, we replace it with the internal one in the enlarged scope. Then, we may perform let-reduction safely and further reduce the redex that has been created.

More generally, the reduction semantics will be set so that Σs can always be extruded out of redex forms. Note that the separation of witness definitions from coercions (*i.e.* splitting pack) plays here an essential role: if the two constructs were bound together, coercions should be necessarily extruded too, which would be hard to achieve in a local manner. Here, only the witness definitions are extruded, while the coercions simply stay where they are.

Openings also introduce linear items into the environment and thus preclude substitution. Note however that they are neither part of values nor of results, because they can be eliminated: by reduction, an opening $\mathsf{open}\ \langle\beta\rangle\ M$ will eventually lead to an "open-exists" pattern $\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\,M'$. This combination just performs a transfer of an existential resource from the inner name α to the outer one β, as demonstrated by the following derivation:

$$
\dfrac{\dfrac{\Gamma, \exists\alpha \vdash M : \tau}{\Gamma \vdash \exists\alpha.\,M : \exists\alpha.\,\tau}\ \textsc{Exists}}{\Gamma, \exists\beta \vdash \mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\,M : \tau[\alpha \leftarrow \beta]}\ \textsc{Open}
$$

Therefore, the pattern $\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\,M$ can simply be eliminated into a renaming from the internal to the external name $M[\alpha \leftarrow \beta]$. This way, reduction makes the bottom-left cycle of Fig.1 vanish.

Reduction The semantics of $\mathsf{F}^{\forall}$ is given by a call-by-value reduction strategy, described by a small-step reduction relation, that does not rely on types (it is compatible with type erasure). 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.5. Note that, as opposed to Dreyer (2), evaluation also takes place under existential bindings. We define the *exposed type variables* of a context E, written $\mathit{etv}(E)$, that are either binding type variables or type variables that are carried by an opening or an open witness definition. 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. Reduction steps are sorted into four groups.

Rules of the main group describe the contraction of redexes. The let-reduction, the β-reduction, the reduction of type applications, and the record projection are as usual. The last rule of this group is the reduction of the "open-exists" pattern explained above. Notice that type substitution is a partial function on terms, because syntax is not stable under type substitution: for instance,

$$
\begin{array}{rcll}
\mathsf{let}\ x = v\ \mathsf{in}\ M & \rightsquigarrow & M[x \leftarrow v] & & \textsc{Redex-Let}\\
(\lambda(x:\tau)\,M)\ v & \rightsquigarrow & \mathsf{let}\ x = v\ \mathsf{in}\ M & & \textsc{Redex-App}\\
(\Lambda\alpha.\,M)\,\tau & \rightsquigarrow & M[\alpha \leftarrow \tau] & & \textsc{Redex-Inst}\\
\{(\ell_i = v_i)^{i\in 1..n}\}.\ell_k & \rightsquigarrow & v_k & \text{if } 1 \le k \le n & \textsc{Redex-Proj}\\
\mathsf{open}\ \langle\beta\rangle\ \exists\alpha.\,w & \rightsquigarrow & w[\alpha \leftarrow \beta] & & \textsc{Redex-Open}
\end{array}
$$

$$
\begin{array}{rcll}
E\,[\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w] & \rightsquigarrow & \Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,E\,[w]\,[\beta\leftarrow\alpha] & \text{if } \alpha \notin \mathit{ftv}(E) \text{ and } (\{\alpha,\beta\}\cup\mathit{ftv}(\tau))\cap\mathit{etv}(E)=\emptyset & \textsc{Extrude}\\
\Sigma\,\langle\beta_1\rangle\,(\alpha_1=\tau_1)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2=\tau_2)\,w & \rightsquigarrow & \Sigma\,\langle\beta_1\rangle\,(\alpha_1=\tau_1)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2=\tau_2[\alpha_1\leftarrow\tau_1])\,w & \text{if } \alpha_1 \in \mathit{ftv}(\tau_2) & \textsc{Sigma-Sigma}
\end{array}
$$

$$
\begin{array}{rcll}
((\lambda(x:\tau_0)\,M):\tau_1\to\tau_2)\ v & \rightsquigarrow & ((\lambda(x:\tau_0)\,M)\ (v:\tau_0):\tau_2) & & \textsc{Coerce-App}\\
(u:\forall\alpha.\,\tau')\,\tau & \rightsquigarrow & (u\,\tau:\tau'[\alpha\leftarrow\tau]) & & \textsc{Coerce-Inst}\\
(u:\{(\ell_i:\tau_i)^{i\in 1..n}\}).\ell_k & \rightsquigarrow & (u.\ell_k:\tau_k) & \text{if } 1\le k\le n & \textsc{Coerce-Proj}\\
\mathsf{open}\ \langle\alpha\rangle\ (u:\exists\alpha.\,\tau) & \rightsquigarrow & (\mathsf{open}\ \langle\alpha\rangle\ u:\tau) & & \textsc{Coerce-Open}\\
((u:\tau):\tau') & \rightsquigarrow & (u:\tau') & & \textsc{Coerce-Coerce}
\end{array}
$$

$$
\begin{array}{rcll}
\nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w & \rightsquigarrow & \nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w[\beta\leftarrow\alpha] & \text{if } \beta\in\mathit{ftv}(w) & \textsc{Erase-Nu-Sigma1}\\
\nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w & \rightsquigarrow & \nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w[\alpha\leftarrow\tau] & \text{if } \alpha\in\mathit{ftv}(w) \text{ and } \beta\notin\mathit{ftv}(w) & \textsc{Erase-Nu-Sigma2}\\
\nu\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,w & \rightsquigarrow & w & \text{if } \alpha,\beta\notin\mathit{ftv}(w) & \textsc{Erase-Nu-Sigma3}
\end{array}
$$

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid E\ M \mid v\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid E\,\tau\\
& \mid & \{(\ell_i = v_i)^{i\in 1..k}\,;\,\ell_{k+1}=E\,;\,(\ell_i=M_i)^{i\in k+2..n}\} \mid E.\ell\\
& \mid & \exists\alpha.\,E \mid \Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,E \mid (E:\tau) \mid \mathsf{open}\ \langle\alpha\rangle\ E \mid \nu\alpha.\,E
\end{array}
\qquad
\textsc{Context}\ \frac{M \rightsquigarrow M'}{E[M] \rightsquigarrow E[M']}
$$

$$
\mathit{etv}([\cdot]) = \emptyset \qquad \mathit{etv}(\Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,E) = \{\alpha,\beta\}\cup\mathit{etv}(E)
$$

$$
\left.\begin{array}{l}\mathit{etv}(\exists\alpha.\,E)\\ \mathit{etv}(\nu\alpha.\,E)\\ \mathit{etv}(\mathsf{open}\ \langle\alpha\rangle\ E)\end{array}\right\} = \{\alpha\}\cup\mathit{etv}(E)
\qquad
\left.\begin{array}{l}\mathit{etv}(E\ M)\quad \mathit{etv}(M\ E)\quad \mathit{etv}(\mathsf{let}\ x = E\ \mathsf{in}\ M)\\ \mathit{etv}(E\,\tau)\quad \mathit{etv}(E.\ell)\quad \mathit{etv}((E:\tau))\\ \mathit{etv}(\{(\ell_i=M_i)^{i\in 1..k}\,;\,\ell_{k+1}=E\,;\,(\ell_i=M_i)^{i\in k+2..n}\})\end{array}\right\} = \mathit{etv}(E)
$$

Figure 5. Reduction rules

$(\mathsf{open}\ \langle\beta\rangle\ M)[\beta\leftarrow\tau]$ is undefined. The type system ensures that type substitution is only performed when it is well-defined.

The second group of rules implements the extrusion of Σs through every other construct: Rule Extrude permits extrusion through evaluation contexts, provided this is valid with respect to scopes of (exposed) type variables. To make the exchange of two Σs possible, Rule Sigma-Sigma substitutes the definition of the outer one to delete dependencies.

The third group of reduction rules keeps track of coercions during reduction, as exemplified by Rule Coerce-App. Notice that nested coercions are merged, the outer one taking priority (Rule Coerce-Coerce), which makes the top-most cycle of Fig.1 vanish.

Finally, the fourth group of rules is responsible for the erasure of restricted open witness definitions. Rule Erase-Nu-Sigma1 replaces the external name with the internal one. The role of Rule Erase-Nu-Sigma2 is to replace the type variable of a witness with the witness itself: the same substitution occurs in System F while unpack-ing a pack-ed term. Finally, the restricted definition is erased by Rule Erase-Nu-Sigma3.

Remark that only Σs are extruded: every local introduction of resources by a ν stays local and is eventually eliminated. Similarly, coercions are not extruded either.

3.5 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) and the instantiation lemma, which comes in two forms:

Lemma 3 (Instantiation by equation)**.** *Assume that* $\Gamma \vdash \tau\ \mathrm{wf}$ *and* $\Gamma, \forall\alpha, \Gamma' \vdash M : \tau'$ *hold and that no free type variable of* τ *is existentially bound in* Γ*. Then* $\Gamma, \forall(\alpha=\tau), \Gamma' \vdash M : \tau'$ *holds.*

Lemma 4 (Instantiation by substitution)**.** *Assume that* $\Gamma, \forall(\alpha=\tau), \Gamma' \vdash M : \tau'$ *holds. Then* $M[\alpha\leftarrow\tau]$ *is well-defined and* $\Gamma, \Gamma'[\alpha\leftarrow\tau] \vdash M[\alpha\leftarrow\tau] : \tau'[\alpha\leftarrow\tau]$ *holds.*

The proof of subject reduction itself is not really informative, but it is particularly interesting that the proof is absolutely standard and almost straightforward. It proceeds by induction on the reduction relation.

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

Progress is proved by induction on the typing derivation.

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.*

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, since evaluation takes place under the binders ν and $\exists$. Moreover, this allows to consider the reduction of *open* programs, *i.e.* programs with free type variables. This is the case of programs with abstract types, which come from unrestricted openings or open witness definitions. This closely corresponds to ML programs composed of modules with abstract types.

3.6 Translation to System F

From F to F$^\forall$ As mentioned in §2.2, the encoding of pack and unpack is unsurprisingly straightforward. It preserves typing and abstraction as well as semantics: the encoding keeps the underlying untyped skeleton unchanged.

From F$^\forall$ to F Conversely, it is also possible to globally reorganize every closed term of F$^\forall$ so that it uses (the encodings of) pack and unpack. We sketch out this transformation that consists in five rewriting stages, which we review now:

$$
\begin{array}{rcl}
Q^\alpha & ::= & \mathsf{open}\ \langle\alpha\rangle\ M \ \mid\ \Sigma\,\langle\alpha\rangle\,(\beta=\tau)\,M \ \mid\ Q^\alpha\ M \ \mid\ M\ Q^\alpha \ \mid\ Q^\alpha\,\tau \ \mid\ \mathsf{pack}\ \langle\tau, Q^\alpha\rangle\ \mathsf{as}\ \exists\beta.\,\tau' \\
& \mid & \nu\beta.\,Q^\alpha \ \mid\ Q^\alpha.\ell \ \mid\ \mathsf{open}\ \langle\beta\rangle\ Q^\alpha \ \mid\ \Sigma\,\langle\beta\rangle\,(\gamma=\tau)\,Q^\alpha \qquad \text{where } \beta,\gamma\neq\alpha \\
& \mid & \{(\ell_i = M_i)^{i\in I}\ ;\ \ell = Q^\alpha\ ;\ (\ell_j = M_j)^{j\in J}\} \ \mid\ \mathsf{let}\ x = M\ \mathsf{in}\ Q^\alpha \ \mid\ \mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ M
\end{array}
$$

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

Figure 6. Translation to System F (excerpts): extrusion of opens and Σs, intrusion of νs.

1. From the typing derivation, insert coercions around Σs and $\exists$s in order to get $\Sigma\,\langle\beta\rangle\,(\alpha=\tau')\,(M:\tau)$ and $\exists\alpha.\,(M:\tau)$.
2. Replace existential quantifiers by uses of pack, according to the rule: $\exists\alpha.\,(M:\tau) \twoheadrightarrow \nu\alpha.\,\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{pack}\ \langle\alpha, x\rangle\ \mathsf{as}\ \exists\alpha.\,\tau$
3. Extrude opens and Σs using let-bindings (as described by a representative set of rules on left-hand side of Fig.6) and intrude νs so that they get closer to each other (right-hand side of Fig.6).
4. Recover System F constructs:

   $\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ M\ \mathsf{in}\ M' \twoheadrightarrow \mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M'$
   $\nu\alpha.\,\mathsf{let}\ x = \Sigma\,\langle\alpha\rangle\,(\alpha=\tau_0)\,(M:\tau)\ \mathsf{in}\ M'$
   $\twoheadrightarrow \mathsf{unpack}\,(\mathsf{pack}\ \langle\tau_0, M[\alpha\leftarrow\tau_0]\rangle\ \mathsf{as}\ \exists\alpha.\,\tau)\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M'$
5. Finally, remove all coercions.

All stages but (3) are compositional.

Proposition 3. *The translation is type-preserving, abstraction-preserving and semantics-preserving.*

The property holds for each stage of the translation and each rewriting rule. Abstraction is unchanged since the scopes of Σs are not altered by the transformation. Semantics is preserved in the following way: the untyped skeleton of the image of the translation let-reduces to the skeleton of the source. Thus, while the evaluation order is kept unchanged, reducing the image requires more β-reduction steps than reducing the source.

The latter point highlights the increase of modularity brought by $\mathsf{F}^{\curlyvee}$ over System F: it allows for organizing the code more freely.

The logical facet By erasing the terms from the typing rules, we can consider the logic underlying core $\mathsf{F}^{\curlyvee}$: not only the expressible formulas are exactly those of second-order arithmetic, but also we can deduce from the translations above that the *valid formulas are identical*. In particular, $\mathsf{F}^{\curlyvee}$'s logic is consistent. Moreover, since the reduction steps are increased by the translation and since the untyped skeletons of System F terms are terminating, the untyped skeleton of every closed program of $\mathsf{F}^{\curlyvee}$ is also terminating. In addition, the fact that the untyped skeleton of the image let-reduces to the untyped skeleton of the source essentially tells us that the two pieces of program compute *the same things* and *in the same way*: the translation to System F just performs a reorganization of the type derivation. Hence, the correspondence with System F is twofold: it holds on the static as well as on the dynamic viewpoint, which connects $\mathsf{F}^{\curlyvee}$ with System F in a very tight manner.

The gain of modularity brought by core $\mathsf{F}^{\curlyvee}$ in terms of programming can be read back in terms of proofs: it allows new assembling of partial proofs (*i.e.* with abstract types), where environments are *zipped* when combining proof-terms.

One can wonder what is the logical status of the typing rules we presented: Rule COERCE has the form of a subtyping rule with the semantics of the identity (coercions are erased by the translation); Rule EXISTS is the right introduction rule for the existential quantifier; Rule OPEN is the right elimination rule; Rule SIGMA is a left introduction rule; Rule NU is a left elimination rule.

4. Extensions of $\mathsf{F}^{\curlyvee}$

In this section we consider several extensions for $\mathsf{F}^{\curlyvee}$, for which soundness properties (lemmas 1 and 2) of $\mathsf{F}^{\curlyvee}$ extend.

4.1 More flexible non-recursive type equations

Core $\mathsf{F}^{\curlyvee}$ imposed a simple but strong restriction to enforce type equations to be acyclic. In this section we present a more general technique to control recursive types, by enriching the structure of typing environments in a natural way: we no longer consider them as sequences, *i.e. totally* ordered sets, but as *partially* ordered sets, where the order relation expresses dependencies between bindings and is required to be *acyclic*, which means that no binding can (transitively) depend on itself. This disallows the zipping of two environments when this condition could not be satisfied.

More specifically, a typing environment Γ is a dag represented as a pair $(\mathcal{E}, \prec)$ of a finite set of bindings $\mathcal{E}$ and an acyclic partial order $\prec$ on $\mathit{dom}\,\mathcal{E}$, *i.e.* there exists no binding b such that $\mathit{dom}\,b \prec \mathit{dom}\,b$. We sometimes write $b \prec b'$ instead of $\mathit{dom}\,b \prec \mathit{dom}\,b'$. If $b \prec b'$, we say that b depends on b'. We use the following notation for composing and decomposing typing environments so that typing rules look familiar:

Notation 1. We write $\Gamma_1, (b \prec \mathcal{D}), \Gamma_2$ when no binding in Γ_1 depends on b, and b does not depend on bindings of Γ_2, and $\mathcal{D}$ is the set of bindings b depends on. In particular, when Γ_2 is empty, b is minimal for the dependency relation.

Definition 2 (Zipping)**.** Let Γ_1 and Γ_2 be two typing environments of the form $(\mathcal{E}_1, \prec_1)$ and $(\mathcal{E}_2, \prec_2)$. Let $\prec$ be the transitive closure $(\prec_1 \cup \prec_2)^+$. If $\prec$ is acyclic, the zipping of Γ_1 and Γ_2, written $\Gamma_1 \curlyvee \Gamma_2$, is $(\mathcal{E}_1 \curlyvee \mathcal{E}_2, \prec)$, where $\mathcal{E}_1 \curlyvee \mathcal{E}_2$ is:

- $\{b_1 \curlyvee b_2 \mid b_1 \in \mathcal{E}_1 \wedge b_2 \in \mathcal{E}_2 \wedge \mathit{dom}\,b_1 = \mathit{dom}\,b_2\}$, if $\mathcal{E}_1$ and $\mathcal{E}_2$ have the same domain.
- $\mathcal{E}_1' \curlyvee \mathcal{E}_2'$ where $\mathcal{E}_1'$ is $\mathcal{E}_1 \cup \{(\forall\alpha) \mid (\exists\alpha) \in \mathcal{E}_2 \wedge \alpha \notin \mathit{dom}\,\mathcal{E}_1\}$ and symmetrically for $\mathcal{E}_2'$, when $\mathcal{E}_1'$ and $\mathcal{E}_2'$ have the same domain.
- undefined otherwise.

The zipping of Γ_1 and Γ_2 is undefined if $\prec$ is not acyclic or if $\mathcal{E}_1 \curlyvee \mathcal{E}_2$ is undefined. □

The second item in the definition of zipping extends the environments before considering their zipping. This performs an implicit weakening on each side that refines the detection of cycles, as will be exemplified below.

Rules SIGMA, OPEN and LET introduce new dependencies to keep track of cycles. We review them now.

$$
\textsc{Sigma}\quad \frac{\Gamma, (\forall(\alpha=\tau') \prec \mathcal{D}') \vdash M : \tau \qquad \mathcal{D}' \subseteq \mathcal{D}}{\Gamma, (\exists\beta \prec \mathcal{D}) \vdash \Sigma\,\langle\beta\rangle\,(\alpha=\tau')\,M : \tau[\beta\leftarrow\alpha]}
$$

Unsurprisingly, Rule SIGMA specifies that the external name has at least all dependencies of the internal name, among which lay the (dependencies of the) free type variables of the witness. This prevents the example of external recursion seen in §2.3, which we

$$\textsc{Var}\ \frac{\Gamma\ \mathit{pure} \qquad \Gamma \vdash \mathsf{ok}}{\Gamma \vdash x\ :\ \Gamma(x)} \qquad \textsc{Lam}\ \frac{\Gamma\ \mathit{pure} \qquad \Gamma, (x : \tau_1 \prec \mathcal{D}) \vdash M\ :\ \tau_2}{\Gamma \vdash \lambda(x : \tau_1)\, M\ :\ \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma_1 \vdash M_1\ :\ \tau_2 \to \tau \qquad \Gamma_2 \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2\ :\ \tau} \qquad \textsc{Let}\ \frac{\mathit{dom}^{\forall}\,\Gamma_1 \cap \mathit{dom}^{\exists}\,\Gamma_2 \subseteq \mathcal{D} \qquad \Gamma_1 \vdash M_1\ :\ \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ :\ \tau_2}$$

$$\textsc{Gen}\ \frac{\Gamma\ \mathit{pure}}{\Gamma, (\forall\alpha \prec \mathcal{D}) \vdash M\ :\ \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash \tau\ \mathsf{wf} \qquad \Gamma \vdash M\ :\ \forall\alpha.\,\tau'}{\Gamma \vdash M\,\tau\ :\ \tau'[\alpha \leftarrow \tau]}$$

$$\textsc{Empty}\ \frac{\Gamma\ \mathit{pure}}{\Gamma \vdash \{\}\ :\ \{\}} \qquad \textsc{Record}\ \frac{(\Gamma_i \vdash M_i\ :\ \tau_i)^{i\in 1..n} \qquad \mathsf{injective}\ (i \mapsto \ell_i)^{i\in 1..n}}{\Gamma_1 \curlyvee \cdots \curlyvee \Gamma_n \vdash \{(\ell_i = M_i)^{i\in 1..n}\}\ :\ \{(\ell_i : \tau_i)^{i\in 1..n}\}}$$

$$\textsc{Proj}\ \frac{1 \le k \le n \qquad \Gamma \vdash M\ :\ \{(\ell_i : \tau_i)^{i\in 1..n}\}}{\Gamma \vdash M.\ell_k\ :\ \tau_k} \qquad \textsc{Exists}\ \frac{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M\ :\ \tau}{\Gamma \vdash \exists\alpha.\,M\ :\ \exists\alpha.\,\tau}$$

$$\textsc{Coerce}\ \frac{\Gamma \vdash \tau' \equiv \tau \qquad \Gamma \vdash M\ :\ \tau'}{\Gamma \vdash (M : \tau)\ :\ \tau} \qquad \textsc{Sigma}\ \frac{\mathcal{D}' \setminus (\{\beta\} \cup \mathit{dom}\,\Gamma') \subseteq \mathcal{D},\ \text{if} \simeq \text{is} =}{\Gamma, (\exists\beta \prec \mathcal{D}), \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha \simeq \tau')\,M\ :\ \tau[\alpha \leftarrow \beta]}$$

$$\textsc{Open}\ \frac{\Gamma \vdash M\ :\ \exists\alpha.\,\tau}{\Gamma, (\exists\alpha \prec \mathit{dom}\,\Gamma \setminus \mathit{dom}^{\cong}\,\Gamma) \vdash \mathsf{open}\,\langle\alpha\rangle\ M\ :\ \tau} \qquad \textsc{Nu}\ \frac{\alpha \notin \mathit{ftv}(\tau) \qquad \Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M\ :\ \tau}{\Gamma \vdash \nu\alpha.\,M\ :\ \tau}$$

$$\textsc{Weaken}\ \frac{\Gamma' \Vdash \Gamma \qquad \Gamma \vdash M\ :\ \tau}{\Gamma' \vdash M\ :\ \tau} \qquad \textsc{Sim}\ \frac{\Gamma \vdash \tau \lhd \tau' \qquad \Gamma \vdash M\ :\ \tau'}{\Gamma \vdash M\ :\ \tau} \qquad \textsc{Fix}\ \frac{\Gamma, (x : \tau \prec \mathcal{D}) \vdash s\ :\ \tau}{\Gamma \vdash \mu(x : \tau)\,s\ :\ \tau}$$

Figure 7. Typing rules of the extended system

recall below, to be well-typed:

$$\{\ \ell_1 = \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \beta_2 \to \beta_2)\,M_1\ ;\\ \ell_2 = \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \beta_1 \to \beta_1)\,M_2\ \}$$

The dependency $\beta_1 \prec \beta_2$ is required to type the first component, since the witness depends on β_2. Symmetrically, $\beta_2 \prec \beta_1$ is also required to type the second component. Consequently, the zipping is forbidden because of the obvious cycle.

As opposed to the case of Rule SIGMA, the witness is unknown in the open construct. Hence, the condition placed on Rule OPEN is stronger: the abstract type variable (possibly) depends on every type variable present in the context, except on type definitions since these are only indirections: it is unnecessary to track dependencies on internal names since they are always included in the dependencies of the external names, as described by Rule SIGMA. Conversely, taking dependencies on internal names into account would be too coarse and impede subject reduction, since a consequence of extrusions is the expansion of the scope of internal names.

$$\textsc{Open}\ \frac{\Gamma \vdash M\ :\ \exists\alpha.\,\tau}{\Gamma, (\exists\alpha \prec \mathit{dom}\,\Gamma \setminus \mathit{dom}^{=}\,\Gamma) \vdash \mathsf{open}\,\langle\alpha\rangle\ M\ :\ \tau}$$

As a result, the above example would again be rejected if the Σs were replaced with "open-exists" patterns. By contrast, the following example is well-typed, since the witness of the first branch does not depend on β_2.

$$\{\ \ell_1 = \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \mathit{int})\,M_1\ ;\\ \ell_2 = \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \beta_1 \to \beta_1)\,M_2\ \}$$

Rewriting this piece of code with "open-exists" patterns is again well-typed, in spite of the stronger condition on Rule OPEN, thanks to the implicit weakening in zipping: we can type the first branch without using $\forall\beta_2$ in the environment (provided M_1 does not mention β_2). Therefore, the requirement $\beta_1 \prec \beta_2$ is not required in the first branch and no cycle is detected.

Finally, Rule LET (see Fig.7) highlights variables that are used, hence possibly hidden in an existential value, in the first branch of the let and used in an opening in the second branch. Therefore, the value variable that is bound in the let must depend on these variables. These are indeed responsible for the cycle in the example of internal recursion seen in §2.3 and reminded below:

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

The binding $\forall\beta$ is required in the typing environment of the bound expression, whereas the binding $\exists\beta$ appears in the typing environment for the body. Thus, the constraint $x \prec \beta$ is required in the typing environment of the body, which prevents typing $\mathsf{open}\,\langle\beta\rangle\,x$, as Rule OPEN requests that $\exists\beta$ must be minimal in the dependency relation.

4.2 Addressing the double vision problem

Defining an expression that manipulates an abstract type before its witness has been given is sometimes desirable, as it brings more freedom in the code structure. It may also become necessary when building recursive values. Currently, the following term is considered as ill-typed:

$$\exists\beta.\,\mathsf{let}\ f = \lambda(x : \beta)\,x\ \mathsf{in}\ \Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,(1 : \alpha)$$

This is because Rule SIGMA (see §2.2.3) does not let the external name β visible in its premise. It is easy to correct this by leaving a $\forall\beta$ in the premise instead of $\exists\beta$ (see below). However, the following piece of code would still be rejected:

$$\exists\beta.\,\mathsf{let}\ f = \lambda(x : \beta)\,x\ \mathsf{in}\ \Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,f\,(1 : \alpha)$$

After the existential resource β is introduced, it defines f as the identity on β and then uses f in the context of the open witness definition $\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})$. However, we do not know that α and β denote the *same* witness: the application $f\ (1 : \alpha)$ is ill-typed.

This is called the *double vision problem*: it characterizes the inability to maintain a link between the internal and external view of a given type. This problem is well-known in the study of recursive modules, but as we can see it already happens in the absence of recursion. To solve this problem, is suffices to carry the missing information in the context (for clarity, dependencies are omitted):

$$\textsc{Sigma}\ \frac{\Gamma, \forall\beta, \Gamma', \forall(\alpha \lhd \beta = \tau') \vdash M\ :\ \tau}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau')\,M\ :\ \tau[\alpha \leftarrow \beta]}$$

The typing environment is enriched with a new kind of equation $\forall(\alpha \lhd \beta = \tau')$, which says that the witness τ' is denoted by the internal name α, and, in addition, that the external name β can

be viewed internally as α. This is realized through the use of the *similarity* relation defined under a context Γ and written $\triangleleft$ that satisfies all the equalities between internal and external names that are present in the context Γ. It is used through Rule SIM (Fig. 7).

The reader may wonder why the authors decided to use both an external and an internal name, while they denote the same object, instead of using only one name as done in RTG where a single type reference is used along with two scopes, only one of which contains a type definition.

We give two reasons for handling two names and an equation relating them: first, it corresponds to practice in recursive modules, where a single type component is reached through two different paths, which leads to the double vision problem. Second, the use of two names makes programs more maintainable in the sense that it is more respectful to the notion of *interface*: whatever is the internal name, the external name will always be the same. Thus, one can apply an internal renaming without changing the external type.

4.3 Extending $\mathsf{F}^{\forall}$ with type-level recursion

Extended non-recursive type definitions lead to a finer type checking but do not require a change in the semantics. By contrast, permitting recursive type definitions has the reverse effect: typing is unchanged, but semantics must be adapted. We extend the type algebra with a fixpoint and specify with the use of the symbol $\approx$ instead of $=$ when a type equation is allowed to contribute to a cycle. Wellformedness ensures that recursive types are contractive.

$$\begin{array}{llll} \simeq & ::= & = \mid \approx & \qquad M \quad ::= \quad \ldots \mid \Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,M \\ \tau & ::= & \ldots \mid \mu\alpha.\,\tau & \qquad w \quad ::= \quad \ldots \mid \Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,w \end{array}$$

We also extend the type compatibility relation with the usual unfolding rule for recursive types (see Fig.12) and consider that type compatibility is co-inductively defined. We add the following rules to the reduction relation:

$$\begin{array}{l} \Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,\Sigma\,\langle\beta'\rangle\,(\alpha' \simeq \tau')\,w \\ \qquad\qquad \rightsquigarrow\ \Sigma\,\langle\beta'\rangle\,(\alpha' \simeq \tau'[\alpha \leftarrow \beta])\,\Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,w \end{array}$$

$$\begin{array}{l} \nabla\beta'.\,\Sigma\,\langle\beta'\rangle\,(\alpha' \approx \tau')\ (\Sigma\,\langle\beta_i\rangle\,(\alpha_i \simeq \tau_i))^{i\in I}\,v \\ \rightsquigarrow\ \nabla\beta'.\,\Sigma\,\langle\beta'\rangle\,(\alpha' = \mathit{close}(\alpha' \triangleleft \beta' \approx \tau', (\alpha_i \triangleleft \beta_i \simeq \tau_i)^{i\in I})) \\ \qquad (\Sigma\,\langle\beta_i\rangle\,(\alpha_i \simeq \tau_i))^{i\in I}\,v \quad \text{where } \nabla \text{ stands for } \nu \text{ or } \exists \end{array}$$

When two Σs have to be exchanged, it is no longer possible to substitute the first witness into the second one for wellformedness reasons. Instead, we replace the first internal name with the external one during swapping, as described by the first reduction rule. It should be applied when needed, to put a Σ closer to its corresponding ν or $\exists$, when there is one, so that the second rule can apply.

The second rule specifies that a closed or restricted, potentially recursive type definition can be resolved into a non-recursive one, that involves a recursive witness. To do this, the *close* operator, that is defined in Fig.8, gathers the other witnesses and ties the recursive knot. Thanks to co-induction, the provable equalities are unchanged. The reduction below exemplifies the closure operation:

$$\begin{array}{l} \nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 \approx \alpha_1 \times \beta_2)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \alpha_1 \times \alpha_2)\,v \\ \rightsquigarrow\ \nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau)\,\Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \alpha_1 \times \alpha_2)\,v \\ \rightsquigarrow\ \Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \tau \times \alpha_2)\,\nu\beta_1.\,\Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \tau)\,v \\ \rightsquigarrow\ \Sigma\,\langle\beta_2\rangle\,(\alpha_2 \approx \tau \times \alpha_2)\,v[\alpha_1 \leftarrow \tau] \\ \text{where } \tau = \mathit{close}(\alpha_1 \triangleleft \beta_1 \approx \alpha_1 \times \beta_2, \alpha_2 \triangleleft \beta_2 \approx \alpha_1 \times \alpha_2) \\ \qquad\quad = \mu\alpha_1.\,(\alpha_1 \times \mu\alpha_2.\,(\alpha_1 \times \alpha_2)) \end{array}$$

The term we consider contains two mutually recursive type definitions, and the external name β_1 of the first one is restricted. The *close* operator computes the closed witness τ, which becomes the new, recursive witness of β_1, defined by a non-recursive equation. Then, the innermost Σ can be extruded, and the restricted equation is eventually eliminated.

By definition, this semantics ensures that only equations that are marked as potentially recursive may actually create recursive types during reduction. Type soundness ensures that this is sufficient to reduce well-typed programs, *i.e.* that recursive types are never needed in other configurations. Hence, although abstract types can be used in a flexible manner, the risk of inadvertently using recursive types via type abstraction can be tracked by the type system and tightly tuned by the user.

It is also interesting that mutually recursive equations are explicitly resolved during reduction, and moreover in a standard way.

4.4 Extending $\mathsf{F}^{\forall}$ with term-level recursion

In this section, we extend $\mathsf{F}^{\forall}$ with recursive values $\mu(x{:}\tau)\,v$, which are necessary to define recursive modules.

Although it is possible to use the well-known backpatching semantics for fixpoints, we prefer a storeless, unrolling-based semantics, so as to avoid the need for references. Our unrolling semantics lies between the backpatching semantics, which computes recursive values at their creation and fails if they are ill-founded, and the lazy semantics, which unfolds recursive values only at their use. As the former we evaluate recursive definitions at their creation, by letting evaluation proceed under fixpoints, but *without* unrolling them. Instead, fixpoints are unrolled *on demand* when they need to be destructed, as with the lazy semantics. (Ill-founded recursion may thus loop at its use instead of its creation, as with the lazy semantics.) The two aspects of our semantics are captured by the form of evaluation contexts and the following reduction rule, respectively:

$$\begin{array}{lll} E & ::= & \ldots \mid \mu(x:\tau)\,E \\ R[\mu(x:\tau)\,v] & \rightsquigarrow & R[\mathsf{let}\ x = \mu(x:\tau)\,v\ \mathsf{in}\ v] \end{array}$$

where R is a redex-form, that is, an application $[\cdot]\ v$, an instantiation $[\cdot]\,\tau$, a projection $[\cdot].\ell$, or an opening $\mathsf{open}\ \langle\alpha\rangle\ [\cdot]$.

In order to enable unrolling, one must ensure that reducing under fixpoints and extruding Σs always give rise to a value, because impure results cannot be substituted. For this purpose, we restrict the body of fixpoints to be *extended results*, denoted by s, which are either results or themselves records, let-bindings, or projections of extended results.

$$\begin{array}{llll} M & ::= & \ldots \mid \mu(x:\tau)\,s & \\ u & ::= & \ldots \mid \mu(x:\tau)\,v \mid x.\ell_1 \cdots .\ell_n & \\ s & ::= & w \mid \mathsf{let}\ x = s\ \mathsf{in}\ s \mid \{(\ell_i = s_i)^i\} \mid s.\ell & \end{array}$$

Pre-values are extended with both fixpoints of values and (possibly empty) sequences of projections of variables.

Soundness properties are straightforwardly preserved by this extension.

In conclusion, adding term-level recursion to $\mathsf{F}^{\forall}$ is not an issue. Indeed, this is a direct consequence of the modularity of $\mathsf{F}^{\forall}$'s constructs. Moreover, our approach permits to keep a standard style of presentation: it uses evaluation contexts, and avoids using references to model recursion.

5. Related work

Russo (21) justifies the meaninglessness of dependent types for modules, by interpreting modules and signatures into semantic objects with System F types. He also uses existential quantifiers to track type generativity. It seems however that his existential types are *implicitly* opened and automatically extruded. Unfortunately, the dynamic semantics of semantic objects is not described.

In the context of run-time type inspection, Rossberg (20) 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

$$\mathit{close}(\alpha \lhd \beta = \tau) \triangleq \tau \qquad \mathit{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i)^{i\in I}, \alpha' \lhd \beta' = \tau') \triangleq \mathit{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i[\beta' \leftarrow \tau'])^{i\in I})$$
$$\mathit{close}(\alpha \lhd \beta \approx \tau) \triangleq \mu\alpha.\, \tau[\beta \leftarrow \alpha] \qquad \mathit{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i)^{i\in I}, \alpha' \lhd \beta' \approx \tau') \triangleq \mathit{close}((\alpha_i \lhd \beta_i \simeq_i \tau_i[\beta' \leftarrow \mu\alpha'.\, \tau'[\beta' \leftarrow \alpha']])^{i\in I})$$

Figure 8. Closing mutually recursive type equations

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 quite different purposes: in particular, λ_{N} is only partially related to traditional existential types, since parametricity is purposely violated.

In spite of strong similarities, some *deep* technical differences remain between RTG and $\mathsf{F}^{\curlyvee}$. The treatment of the linear resources differs significantly: RTG's semantics employs a type store to model static but imperative type reference updates, whereas we just use extrusions of Σ binders. These two approaches might be related by seeing our extrusion as a local treatment of his type store, as has already been proposed for value references (23). Dreyer uses assignment in a global store to guarantee the uniqueness of writing: this exposes the evaluation order in the typing rules of RTG and makes them asymmetrical, moving away from a logical specification, whereas we *zip* contexts 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 considers them as functions in "destination passing style" (DPS). Despite these strong technical differences, the two systems have similar constructs: the "new" primitive is similar to our ν binder; the "set $\alpha := \tau$ in M" is related to the $\Sigma\,\langle\alpha\rangle\,(\alpha = \tau)\,M$ construct. Note the use of a single type name here (as mentioned in §4.2). The two systems differ a little more in other constructs. In RTG, the creation of an *impure* function of type $\tau_1 \xrightarrow{\alpha\downarrow} \tau_2$, whose body defines a witness for a type variable α, is always prefixed by the DPS construct, namely the generalization by a writable type variable $\Lambda\alpha\uparrow K.M$. The former is useful to write typical examples of recursive modules and allows for their separate compilation. However, this construct taken alone would have to be treated linearly, which would require the introduction of linearity in types, and would raise type wellformedness issues with respect to type substitution. Hence, the two constructs are combined into a single form. It is said that a term with type $\exists\alpha\downarrow K.\tau$ can be understood as a DPS function of type $\forall\alpha\uparrow K.() \xrightarrow{\alpha\downarrow} \tau$. In other words, an existential value is a term where the assignment for the witness is frozen. This implies, however, that the body of a DPS function, hence the body of an existential term, is *not* evaluated. One could argue that it would suffice to predefine the body with a let-binding, so that it is evaluated, but this is not always feasible since the body can depend itself on the type variable α. By contrast, $\mathsf{F}^{\curlyvee}$ disallows the definition of impure functions, but the existential introduction $\exists\alpha.\,M$ corresponds to RTG's type variable generalization $\Lambda\alpha\uparrow K.M$ taken alone. However, evaluation *does* take place under existential quantifiers in $\mathsf{F}^{\curlyvee}$. To make it possible, our *local* management of existential resources and their elimination is of primordial importance. The approach followed in RTG treats type abstraction as a side effect and therefore correlates type abstraction with evaluation. To our point of view, the two must be separated, and $\mathsf{F}^{\curlyvee}$ demonstrates that this is achievable. Moreover, it leads to a finer semantics.

Flatt and Felleisen (5) introduced constraints within signatures to track dependencies, whereas we used constraints only in environments, thus enabling a natural generalization of their structure.

Concluding remarks

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

We showed how openings of existential values and open witness definitions tightly correspond to type abstraction and generativity in modules. More importantly, we highlighted that type abstraction and generativity *should and can* be separated from evaluation, and need *not* be explained as a side effect. Instead, the mechanism of *extrusion* plays a central role.

We exhibited a tight correspondence between core $\mathsf{F}^{\curlyvee}$ and System F: it has exactly the same expressive power, but allows to write programs more modularly. This gives strong theoretical foundations to $\mathsf{F}^{\curlyvee}$ and, by extension, to RTG.

The language $\mathsf{F}^{\curlyvee}$ handles liberal non-recursive type definitions, gives a solution to the double vision problem, and allows mutually recursive equations as well as recursive values.

We believe that $\mathsf{F}^{\curlyvee}$ is promising as the core of a programming language with first-class modules. The *bare simplicity* of the notions $\mathsf{F}^{\curlyvee}$ is based on is its best asset. It would be interesting to directly integrate our approach in existing works on mixin modules.

We limited $\mathsf{F}^{\curlyvee}$ to the definition of *pure* functions to keep the system simple enough: impure functions would indeed need to be treated linearly and would certainly introduce dependencies constraints into types. Yet, this extension is worth considering: it would make the system more canonical and would correlate functions with contexts as it is usually the case. For instance we could recover let-binding as a derived construct. In addition, it would permit to re-explore the duality between existentials and universals that is already visible in the typing rules.

Among future work remains the study of representation independence properties, as well as the integration of programming features such as higher-order types or value references. Higher-order types are motivated by Russo's work on applicative functors.

This work only realizes the first half of our project of defining a core calculus for a module language with simple and logical foundations. Indeed, $\mathsf{F}^{\curlyvee}$ still misses a significant, orthogonal ingredient to scale up: a *path system* must complete it, that would permit to write compact programs and overcome the diamond import problem. This second half, already briefly introduced in an earlier work (15), will be developed independently in another paper. Of course, some form of type inference will eventually be needed in a surface language based on $\mathsf{F}^{\curlyvee}$. 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 would like to thank Paul-André Melliès, François Pottier and Robert Harper for fruitful discussions, and anonymous referees for their helpful comments on earlier versions of this paper.

References

[1] Luca Cardelli and Xavier 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.

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

[3] Derek Dreyer. A type system for recursive modules. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 289–302, 2007.

[4] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 236–249, 2003.

[5] Matthew Flatt and Matthias Felleisen. Units: Cool modules for hot languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[6] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, New York, NY, USA, 1994. ACM.

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

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

[9] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. *SIGPLAN Not.*, 42(1):173–184, 2007.

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

[11] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.10*. INRIA, May 2007.

[12] David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

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

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

[15] Benoît Montagu and Didier Rémy. Towards a simpler account of modules and generativity: Abstract types have *Open* existential types. Available electronically, March 2008.

[16] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming*, pages 201–224, 2003.

[17] Gilles Peskine. *Abstract types in collaborative programs*. PhD thesis, Université Paris VII – Denis Diderot, June 2008.

[18] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[19] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual*, June 2000.

[20] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 241–252, Uppsala, Sweden, September 2003.

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

[22] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Proceedings of ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 214–227, Boston, January 2000.

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

$$\textsc{Entail-Refl}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \Vdash \Gamma} \qquad \textsc{Entail-Trans}\ \frac{\Gamma_1 \Vdash \Gamma_2 \qquad \Gamma_2 \Vdash \Gamma_3}{\Gamma_1 \Vdash \Gamma_3} \qquad \textsc{Entail-Binding}\ \frac{b \neq \exists\alpha \qquad \Gamma, (b \prec \mathcal{D}) \vdash \mathsf{ok}}{\Gamma, (b \prec \mathcal{D}) \Vdash \Gamma}$$

Figure 9. Entailment of environments

$$\textsc{Ok-Var}\ \frac{\mathit{ftv}(\tau) \subseteq \mathcal{D} \subseteq \mathit{dom}\,\Gamma \qquad \Gamma \vdash \tau\ \mathsf{wf} \qquad x \notin \mathit{dom}\,\Gamma}{\Gamma, (x : \tau \prec \mathcal{D}) \vdash \mathsf{ok}} \qquad \textsc{Ok-Exists}\ \frac{\mathcal{D} \subseteq \mathit{dom}\,\Gamma \qquad \Gamma \vdash \mathsf{ok} \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash \mathsf{ok}}$$

$$\textsc{Ok-Eq}\ \frac{\mathit{ftv}(\tau) \subseteq \mathcal{D} \subseteq \mathit{dom}\,\Gamma \qquad \Gamma \vdash \tau\ \mathsf{wf} \qquad \alpha \notin \mathit{dom}\,\Gamma \qquad \beta \notin \mathit{ftv}(\tau)}{\Gamma, (\forall(\alpha \lhd \beta = \tau) \prec \mathcal{D} \uplus \{\beta\}) \vdash \mathsf{ok}} \qquad \textsc{Ok-Forall}\ \frac{\mathcal{D} \subseteq \mathit{dom}\,\Gamma \qquad \Gamma \vdash \mathsf{ok} \qquad \alpha \notin \mathit{dom}\,\Gamma}{\Gamma, (\forall\alpha \prec \mathcal{D}) \vdash \mathsf{ok}}$$

$$\textsc{Ok-EqRec}\ \frac{\mathit{ftv}(\mu\alpha.\,\tau) \subseteq \mathcal{D} \subseteq \mathit{dom}\,\Gamma \qquad \Gamma \vdash \mu\alpha.\,\tau\ \mathsf{wf} \qquad \beta \notin \mathit{ftv}(\tau)}{\Gamma, (\forall(\alpha \lhd \beta \approx \tau) \prec \mathcal{D} \uplus \{\alpha, \beta\}) \vdash \mathsf{ok}} \qquad \textsc{Ok-Empty}\ \frac{}{\varepsilon \vdash \mathsf{ok}}$$

Figure 10. Wellformed environments

$$\textsc{Wf-Var}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \alpha \in \mathit{dom}\,\Gamma}{\Gamma \vdash \alpha\ \mathsf{wf}} \qquad \textsc{Wf-Arrow}\ \frac{\Gamma \vdash \tau_1\ \mathsf{wf} \qquad \Gamma \vdash \tau_2\ \mathsf{wf}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathsf{wf}}$$

$$\textsc{Wf-Record}\ \frac{\mathsf{injective}\ (i \mapsto \ell_i)^{i \in 1..n} \qquad (\Gamma \vdash \tau_i\ \mathsf{wf})^{i \in 1..n}}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\}\ \mathsf{wf}} \qquad \textsc{Wf-Empty}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \{\}\ \mathsf{wf}} \qquad \textsc{Wf-Forall}\ \frac{\Gamma, (\forall\alpha \prec \mathcal{D}) \vdash \tau\ \mathsf{wf}}{\Gamma \vdash \forall\alpha.\,\tau\ \mathsf{wf}}$$

$$\textsc{Wf-Exists}\ \frac{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash \tau\ \mathsf{wf}}{\Gamma \vdash \exists\alpha.\,\tau\ \mathsf{wf}} \qquad \textsc{Wf-Mu}\ \frac{\mu\alpha.\,\tau\ \text{contractive} \qquad \Gamma, (\forall\alpha \prec \mathcal{D}) \vdash \tau\ \mathsf{wf}}{\Gamma \vdash \mu\alpha.\,\tau\ \mathsf{wf}}$$

Figure 11. Wellformed types

$$\textsc{Eq-Refl}\ \frac{\Gamma \vdash \tau\ \mathsf{wf}}{\Gamma \vdash \tau \equiv \tau} \qquad \textsc{Eq-Eq-Left}\ \frac{\Gamma \vdash \tau[\alpha \leftarrow \tau''] \equiv \tau' \qquad \forall(\alpha \lhd \beta \simeq \tau'') \in \Gamma}{\Gamma \vdash \tau \equiv \tau'} \qquad \textsc{Eq-Eq-Right}\ \frac{\Gamma \vdash \tau \equiv \tau'[\alpha \leftarrow \tau''] \qquad \forall(\alpha \lhd \beta \simeq \tau'') \in \Gamma}{\Gamma \vdash \tau \equiv \tau'}$$

$$\textsc{Eq-Fix-Left}\ \frac{\Gamma \vdash \tau[\alpha \leftarrow \mu\alpha.\,\tau] \equiv \tau'}{\Gamma \vdash \mu\alpha.\,\tau \equiv \tau'} \qquad \textsc{Eq-Fix-Right}\ \frac{\Gamma \vdash \tau' \equiv \tau[\alpha \leftarrow \mu\alpha.\,\tau]}{\Gamma \vdash \tau' \equiv \mu\alpha.\,\tau}$$

(Rules for congruence are omitted.)

Figure 12. Compatible types (co-inductive definition)

$$\textsc{Sim-Refl}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \alpha \in \mathit{dom}\,\Gamma}{\Gamma \vdash \alpha \lhd \alpha} \qquad \textsc{Sim-Eq}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \forall(\alpha \lhd \beta \simeq \tau) \in \Gamma}{\Gamma \vdash \alpha \lhd \beta} \qquad \textsc{Sim-Empty}\ \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \{\} \lhd \{\}}$$

(Rules for transitivity and congruence are omitted.)

Figure 13. Similar types