Revised version, 14 April 2008

A Logical Account of Type Generativity:

Abstract types have *open* existential types

Benoît Montagu
INRIA
Benoit.Montagu@inria.fr

Didier Rémy
INRIA
Didier.Remy@inria.fr

Abstract

We present a variant of the explicitly-typed second-order polymorphic λ-calculus with primitive *open existential types*, *i.e.* a collection of more atomic constructs for introduction and elimination of existential types. We equip the language with a call-by-value small-step reduction semantics that enjoys the subject reduction property. We claim that open existential types model abstract types and module type generativity. Our proposal can be understood as a logically-motivated variant of Dreyer's RTG where type generativity is no more seen as a side effect. As recursive types arise naturally with open existential types, even without recursion at the term-level, we present a technique to disable them by enriching the structure of environments with dependencies. The double vision problem is addressed and solved with the use of additional equalities to reconcile the two views.

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 Languages, Theory

Keywords Lambda-Calculus, Modules, Type systems, Type abstraction, 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 the language ML first proposed by MacQueen (1984) in the mid 80's and independently improved and simplified in the mid 90's by Harper and Lillibridge (1994) and by Leroy (1996) is still the one in use in all dialects of ML, with relatively minor differences. Abstract types, higher-order functors, and sharing *a posteriori*, are key ingredients of its expressiveness and success.

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

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

In this paper, we present $\mathsf{F}^{\forall}$ (read F-zip), a variant of the second-order polymorphic λ-calculus with primitive *open existential types*, along with a call-by-value operational semantics, that allows to write programs more modularly, enjoys the *subject reduction property*, and whose underlying concepts are strongly and directly related to logic. More precisely, we decompose constructs for existentials into more atomic ones. We argue that this is the key to model abstract types and generativity. Interestingly, the new constructs in $\mathsf{F}^{\forall}$ for open existential types are very close to Dreyer (2007)'s RTG.

Though the notion of generativity is generally understood and modeled in an imperative way, we show this is not necessary and somewhat incorrect, since this relates type abstraction to evaluation. We argue instead that the fundamental notion of sharing behind generativity can be explained in terms of *extrusion* of some binding. This leads to a more direct and logical explanation of type generativity.

A consequence of the decomposition we operate on existential constructs of System-F is that recursive types appear naturally, even when no recursion is available at the term-level. To avoid their emergence, we introduce extra structure in typing environments for keeping track of dependencies that are requested to be acyclic.

We also introduce equations between abstract and concrete views to solve the double vision problem that occurs in our system.

As a result, we demonstrate that the cumbersome notion of *path* is useless in the sole purpose of type abstraction. We are conscious, however, that paths are useful to write *compact* programs. A path system at the level of types has been introduced by Montagu and Rémy (2008) to recover this compacity. However, its presentation is beyond the scope of this paper and is deferred to another one.

Our present goal is not to increase the expressiveness of the module language, but instead to simplify the underlying concepts and to bridge the gap between the complexity of the state-of-

the-art meta-theory of modules and the intuitive simplicity of the underlying mechanisms.

We think this is a necessary step towards the design of a language with first-class modules that is conceptually economical yet more expressive and flexible.

The rest of the paper is organized as follows. We review previous approaches to abstract types in the next section. We introduce open existential types in §3 and give a formal description in §4. We discuss related work in §5 before concluding remarks.

2. Previous approaches to type abstraction

Existing works on modules and type abstraction can be sorted in three categories. In earlier works, abstract types were usually identified with existential types. However, it has been realized in the mid 80's that existential types do not adequately model type abstraction. Since then, abstract types have been considered as types whose definition has been forgotten and that are accessible through a *path*, *i.e.* modeled by strong sums and dependent types. More recently, Dreyer (2007) used *linear type references* in an internal language to explain abstract and concrete views of type names.

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

We now review the three approaches to type abstraction. For pedagogical purposes, we do not follow the chronological order.

2.1 Paths-based systems

Paths are at the foundation of the majority of recent module systems (Leroy et al. 2007; Milner et al. 1997; Odersky et al. 2003). They arise when a type is made abstract. Since the definition of the type has been forgotten, one cannot refer to it by *how* it is defined, instead one designates it by *where* it is defined: an abstract type is referred to as a projection path from a value variable, which makes types depend on values. In fact, types only depend on value variables, and therefore only require a decidable fragment of dependent types. However, this fragment is not stable under term substitution. This is a serious problem for the definition of a small-step reduction semantics. Different solutions have been proposed in the literature. Yet, none of them is quite satisfactory. A quick review follows.

Courant (1997) designed a module system for the Coq proof assistant, together with a substitution semantics. To achieve this, he used the full power of dependent types along with the strong normalization property of Coq terms. This approach is not applicable to general purpose languages that allow non terminating programs.

Lillibridge (1997) designed a module calculus in which paths are extended so that they are stable under value substitution. He managed to define a substitution semantics and to prove the soundness of his system, but at the expense of a substantial technical complexity.

Leroy (1996) designed a module system and implemented it in the Objective Caml compiler (Leroy et al. 2007), but did not give a direct semantics. Instead he only gave a *translation semantics* of his system, using an untyped λ-calculus with records as the target, and proved the soundness of his system.

In the context of the definition of a formal specification for SML (Milner et al. 1997), Dreyer et al. (2003) defined a module type system along with a typed internal language. They defined an indirect semantics of their module system through a global non trivial elaboration phase towards their internal language and proved the soundness of the internal language.

2.2 Dreyer's RTG

In the purpose of explaining type abstraction and generativity for recursive modules and solving the *double vision problem*, Dreyer (2007) emphasizes the need for declaring a type variable before its definition can be given.

Thus, he introduces primitives to create a type reference and to assign a definition to them, respectively: "$\mathsf{new}\ \alpha\ \mathsf{in}\ M$" introduces a type reference in the scope of M that should be set at most once, with the type reference update "$\mathsf{set}\ \alpha := \tau\ \mathsf{in}\ M$". Then, M and only M will see the concrete definition τ for α while other paths of the program will see α abstractly. In this way, he can handle two visions for a given type variable: an abstract one, without definition, and a concrete one, equipped with a definition.

An effect system is used to track assignments of type references. Dreyer also gives a reduction semantics which carries a type store to record type assignments in an imperative manner.

His system RTG is meant to be an internal language for an elaboration procedure and, although primitives seem adequately chosen, its connection with logic is, unfortunately, not obvious.

2.3 Abstract types as existential types

Much earlier, Mitchell and Plotkin (1988) had shown that abstract types could be understood as existential types. However, they also noticed that existential types do not accurately model type abstraction in modules, especially the notion of generativity, and lack some modular properties.

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

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

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

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

From now on we will consider System-F is equipped with the above constructs, as they can be recovered as a well-known syntactic sugar (Reynolds 1983).

3. Open existential types

3.1 Atomic constructs for existential types

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

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

existential type, bind the opened value to a variable and restrict the scope of the fresh type variable.

The problem with pack is *verbosity*: it requires to completely specify the resulting type, thus duplicating type information in the parts that have not been abstracted away. This 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 constructs, recovering modularity while preserving expressiveness of existential types.

To achieve the decomposition of existential types into more atomic constructs, we first need to enrich typing environments with new items.

3.1.1 Richer contexts for typing judgments

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

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

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

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. For the moment, we consider environments as sequences. Their structure will be enriched again in §3.2.

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

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

The fact that, when it is read bottom-up, OPEN makes the environment decrease might seem unusual. Indeed, it imposes that the subterm should not mention the type variable with which it is opened. It follows a subtle control of scope that is already present in works on resourceful λ-calculi: Kesner and Lengrand (2007) introduce, for instance, an explicit weakening construct, that makes the environment decrease and hence finely controls the scope of a variable. Interestingly, our rule OPEN also looks dual to the usual rule of type generalization:

$$\textsc{Gen}\ \frac{\Gamma, \forall\alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\tau}$$

The quantifier moves downwards from the environment to the type, whereas it happens in the opposite way in rule OPEN.

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

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

We may recover unpack as syntactic sugar:

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

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

3.1.3 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 former construct we obtained into two pieces that introduce an existential quantifier and defines a witness, respectively.

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 makes α existentially bound 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:

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

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

away.

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

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

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

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

To complete the separation, we now split $\exists(\alpha = \tau)\, M$ again. 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.

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

The *open witness definition* $\Sigma\ [\beta]\ (\alpha = \tau)\, M$ 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.

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

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

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

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.

The following piece of program, written in an ML-like syntax, defines an abstract module of integers.

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

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:

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

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.

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.

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

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

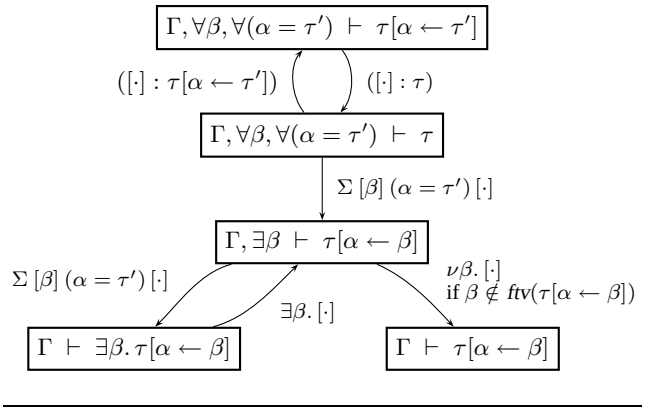

Figure 1. Open existential constructs

3.1.4 Generative functors

Following Russo, 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.

3.1.5 Open existential constructs: a summary

The different constructs introduced for open existential types are gathered on the diagram of Fig.1. It describes their effects on 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 remove the definition from the typing environment and use the external name β for the witness. 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

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

Figure 2. Zippings of bindings.

type by transferring the existential quantifier to the type (bottom left judgment). We can go back by re-opening the existential.

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

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

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* in the case of f and *bool* in the case of x). This yields to the unsound application $f\ x$, which evaluates to $1 + \mathit{true}$.

To prevent abstraction capture, it suffices that *every name 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. 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 give the intuition. It will be completed in §3.3. 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 internally the witness. Therefore the other side *must* consider the choice of the witness as external. That is why it is given the item $\forall\alpha$.

3.2 The appearance of recursive types

The above idea of zipping is unfortunately too generous: it makes recursive types appear naturally. 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 one. It is highlighted by the following example:

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

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

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

This leads us to the recursive equation $\beta = \beta \to \beta$.

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

$$\begin{array}{r}\{\ell_1 = \Sigma\,[\beta_1]\,(\alpha_1 = \beta_2 \to \beta_2)\,M_1\ ;\\ \ell_2 = \Sigma\,[\beta_2]\,(\alpha_2 = \beta_1 \to \beta_1)\,M_2\ \}\end{array}$$

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

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

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

If we consider this program as an unpack, then the contexts C_1 and C_2 are empty. Consequently, α cannot occur free in C_1 or C_2. By splitting unpack, however, this restriction has been waived.

3.3 Preventing the emergence of recursive types

Montagu and Rémy (2008) imposed the restriction above and so forbid the use of recursive types by restraining the zipping of contexts. Although, this solution is simple and of limited expressiveness so that it permitted a translation of terms into System-F terms, it also precludes interesting uses of abtract types.

In this paper we present a more general technique to control recursive types, by enriching the structure of typing environments in a somehow natural way. We no longer consider environments 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*, *i.e.* 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 $\mathcal{E}$, *i.e.* there exists no binding b such that $b \prec b$. If $b \prec b'$, we say 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.

Notation 2. We may use $(b, \mathcal{D})$ to denote the binary relation $\{(b, b') \mid b' \in \mathcal{D}\}$ obtained by lifting the set $\mathcal{D}$ with the binding b.

We write $\mathit{dom}\,b$, $\mathit{dom}\,\mathcal{E}$, and $\mathit{dom}\,\Gamma$ for the domains of b, $\mathcal{E}$, and Γ when seen as mappings.

Definition 1 (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 $(\prec_1 \cup \prec_2)^+$. 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 in the explanation of OPEN.

Rules SIGMA, OPEN and LET introduce new dependencies to keep track of cycles (see Fig.4). We review them now.

Unsurprisingly, SIGMA specifies the external name to have the same dependencies as the internal one, among which lay the (dependencies of the) free type variables of the witness. This prevents

the example of external recursion seen in §3.2, which we remind below, to be well-typed:

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

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 required to type the second component. Consequently, the zipping is forbidden because of the obvious cycle.

OPEN specifies that the abstract type variable (possibly) depends on every type variable present in the context, since we do not know the witness, as opposed to the case of SIGMA. The condition placed on OPEN is then stronger. That is why the above example would again be rejected if the Σ's 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.

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

Rewriting this piece of code with "open-exists" patterns is again well-typed, in spite of the stronger condition on 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_1). Therefore, the requirement $\beta_1 \prec \beta_2$ is not required in the first branch and no cycle is detected.

Finally, LET 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 §3.2 and reminded below:

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

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 $\mathsf{open}\ [\beta]\,x$, as Rule OPEN requests that no binding depend on $\exists\beta$.

3.4 Addressing the double vision problem

Independently from the control of recursion at the type level, we can now handle the double vision problem in a straightforward way. This problem characterizes the inability to maintain a link between the internal and external view of a given type. It happens, for instance, in the following term:

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

After the existential resource β is introduced, it defines f as the identity on β and then uses it under the open witness definition $\Sigma\,[\beta]\,(\alpha = \mathit{int})$. It is not typable, since we do not know that α and β denote the same witness, hence the application $f\ (1:\alpha)$ is ill-typed.

To solve this problem, is suffices to carry the missing information in the context:

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

The context is enriched with a new kind of equation $\forall(\alpha \lhd \beta = \tau')$. Again, it says that the witness τ' is denoted by the internal name α, and, in addition, that the external name β can be viewed under its internal version α. This is realized through the use of the *similarity* relation defined in a context Γ and written $\lhd$ that satisfies all the equalities between internal and external names that are present in the context Γ. It is used through the SIM rule:

$$\textsc{Sim}\quad \frac{\Gamma \vdash M\ :\ \tau' \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash M\ :\ \tau}$$

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. This is indeed Dreyer (2007)'s approach: a single type reference is used along with two scopes for it, one of them contains a definition, while the other does not.

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 yields to the double vision problem. Then, the use of two names increases maintainability of programs 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, it allows to apply an internal renaming without changing the type of a piece of program.

Types:

$$\begin{array}{rcl} \tau & ::= & \alpha \ \mid\ \tau \to \tau \ \mid\ \{(\ell_i : \tau_i)^{i\in 1..n}\} \\ & \mid & \forall\alpha.\,\tau \ \mid\ \exists\alpha.\,\tau \end{array}$$

Terms:

$$\begin{array}{rcl} M & ::= & x \ \mid\ \lambda(x:\tau)M \ \mid\ M\ M \\ & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M \ \mid\ \Lambda\alpha.\,M \ \mid\ M\,[\tau] \\ & \mid & \{(\ell_i = M_i)^{i\in 1..n}\} \ \mid\ M.\ell \ \mid\ \exists\alpha.\,M \\ & \mid & \Sigma\,[\beta]\,(\alpha = \tau)\,M \ \mid\ (M:\tau) \\ & \mid & \mathsf{open}\ [\alpha]\,M \ \mid\ \nu\alpha.\,M \ \mid \end{array}$$

Values and results:

$$\begin{array}{rcl} v & ::= & u \ \mid\ (u:\tau) \\ u & ::= & x \ \mid\ \lambda(x:\tau)M \ \mid\ \Lambda\alpha.\,M \\ & \mid & \{(\ell_i = v_i)^{i\in 1..n}\} \ \mid\ \exists\beta.\,\Sigma\,[\beta]\,(\alpha = \tau)\,v \\ w & ::= & v \ \mid\ \Sigma\,[\beta]\,(\alpha = \tau)\,w \end{array}$$

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

4. The language $\mathsf{F}^{\curlyvee}$

4.1 Syntax

The language $\mathsf{F}^{\curlyvee}$ is based on the explicitly typed version of System-F with records and is extended with constructs of §3.1. Types and terms are described in Fig.3.

As open existentials do not introduce new forms of types, types of $\mathsf{F}^{\curlyvee}$ are type variables, arrow types, record types, universal types, and existential types. The notation $(\ell_i : \tau_i)^{i\in 1..n}$ stands for a sequence of n pairs, each composed of a label and a type. Type wellformedness is defined as usual. Environment wellformedness also takes care of acyclicity of dependencies (see appendix A.1).

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

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

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

$$\textsc{Entail-Var}\ \frac{\Gamma, (x : \tau \prec \mathcal{D}) \vdash \mathsf{ok}}{\Gamma, (x : \tau \prec \mathcal{D}) \Vdash \Gamma} \qquad \textsc{Entail-Eq}\ \frac{\Gamma, (\forall (\alpha = \tau) \prec \mathcal{D}) \vdash \mathsf{ok}}{\Gamma, (\forall (\alpha = \tau) \prec \mathcal{D}) \Vdash \Gamma}$$

$$\textsc{Entail-DbEq}\ \frac{\Gamma, (\forall (\alpha \lhd \beta = \tau) \prec \mathcal{D}) \vdash \mathsf{ok}}{\Gamma, (\forall (\alpha \lhd \beta = \tau) \prec \mathcal{D}) \Vdash \Gamma}$$

Figure 5. Entailment of environments.

We write *ftv*(τ) (respectively *ftv*(M)) to denote the set of free type variables of a type τ (respectively a term M).

4.2 Typing rules

Typing rules for open existentials have already been presented in §3.1. The remaining typing rules are as in System-F with two small differences described and two new rules.

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.

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 2. When Γ does not contain existential items, 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 §4.3.

Because OPEN makes the environment decrease (if it is read bottom-up), the property of weakening is *not* provable in all its generality: one can only weaken a typing judgment with a non-linear item that does not depend on linear items (and does not create cycles). This restriction impedes simple but useful program transformations such as let-expansions. To get rid of this limitation, we add WEAKEN to the type system. This rule is based on the entailment relation on environments described in Fig.5: it says one can forget items of an environment, as long as they are not linear and they do not break dependencies.

Finally, SIM makes use of the similarity relation to present an abstract type variable under its internal concrete view.

4.3 Reduction semantics

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

4.3.1 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* $\Gamma', (x : \tau \prec \mathcal{D}), \Gamma'' \vdash M' : \tau'$ *where* Γ *is pure and* $\Gamma \curlyvee \Gamma'$ *is well defined. Then, we have* $(\Gamma \curlyvee \Gamma'), \Gamma''' \vdash M'[x \leftarrow M] : \tau'$ *where* Γ''' *is the pair* $(\mathcal{E}'', (\mathcal{D}'; (x : \tau, \mathcal{D})))$[1] *and* $(\mathcal{E}'', \mathcal{D}'')$ *is* Γ''.

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

Figure 7. Example of extrusion.

Therefore, values are substitutable if we restrict them to pure terms. Conversely, every irreducible term is *not necessarily* a pure term.

4.3.2 Results and values

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

More precisely, values are defined in Fig.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 λ's or Λ's. Finally, results are values preceded by a (possibly empty) sequence of Σ's.

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

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

4.3.3 Extrusions

Values are substitutable, but some results are not values, namely a sequence of Σ's prefixing a value. How can we handle these results, when they ought to be substituted, without breaking linearity? Our solution is to extrude the Σ's *just enough* to expose and perform the next reduction step.

For example, consider the reduction steps on Fig.7. The initial expression is a let-binding of the form $\mathsf{let}\ x = w\ \mathsf{in}\ M$ where w is the result form $\Sigma\,[\beta]\,(\alpha = \mathit{int})\,(1 : \alpha)$. Hence, the next expected reduction step is the substitution of w for x in M. However, since x occurs twice in M, this would duplicate the opening appearing in w breaking the linear use of β. The solution is to first *extrude* the Σ binding outside of the let-binding, so that the expression bound to x becomes the substitutable value form $(1 : \alpha)$. However, by enlarging the scope of Σ, we have put M in its scope, in which the external name β occurs. Thanks to SIM, β can be viewed as α. 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 Σ can always be extruded out of redex forms.

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 $\mathsf{open}[\beta]M$ will eventually lead to an "open-exists" pattern $\mathsf{open}\,[\beta]\,\exists\alpha.\,M'$. This combination just performs a transfer of an existential resource from the inner name α to the outer one β,

[1] $\mathcal{R}_1; \mathcal{R}_2 = \{(x, y) \mid (x \mathcal{R}_1 y \wedge y \notin \mathit{dom}\,\mathcal{R}_2) \vee (\exists z, x \mathcal{R}_1 z \wedge z \mathcal{R}_2 y)\}$

$$\textsc{Var}\ \frac{\Gamma\ \mathit{pure} \qquad \Gamma \vdash \mathsf{ok}}{\Gamma \vdash x\ :\ \Gamma(x)} \qquad \textsc{Lam}\ \frac{\Gamma, (x : \tau_1 \prec \mathcal{D}) \vdash M\ :\ \tau_2 \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \lambda(x : \tau_1)M\ :\ \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma_1 \vdash M_1\ :\ \tau_2 \to \tau \qquad \Gamma_2 \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2\ :\ \tau}$$

$$\textsc{Let}\ \frac{\{\alpha \mid (\exists\alpha) \in \Gamma_2 \text{ and } (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D} \qquad \Gamma_1 \vdash M_1\ :\ \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2\ :\ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ :\ \tau_2} \qquad \textsc{Gen}\ \frac{\Gamma, (\forall\alpha \prec \mathcal{D}) \vdash M\ :\ \tau \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \Lambda\alpha.\, M\ :\ \forall\alpha.\, \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash M\ :\ \forall\alpha.\, \tau' \qquad \Gamma \vdash \tau\ \mathsf{wf}}{\Gamma \vdash M\,[\tau]\ :\ \tau'[\alpha \leftarrow \tau]}$$

$$\textsc{Empty}\ \frac{\Gamma \vdash \mathsf{ok} \qquad \Gamma\ \mathit{pure}}{\Gamma \vdash \{\}\ :\ \{\}} \qquad \textsc{Record}\ \frac{\forall i \in 1..n,\ \Gamma_i \vdash M_i\ :\ \tau_i \qquad \forall i, j, i \neq j \Rightarrow \ell_i \neq \ell_j}{\Gamma_1 \curlyvee \cdots \curlyvee \Gamma_n \vdash \{(\ell_i = M_i)^{i\in 1..n}\}\ :\ \{(\ell_i : \tau_i)^{i\in 1..n}\}} \qquad \textsc{Proj}\ \frac{\Gamma \vdash M\ :\ \{(\ell_i : \tau_i)^{i\in 1..n}\} \qquad 1 \leq k \leq n}{\Gamma \vdash M.\ell_k\ :\ \tau_k}$$

$$\textsc{Exists}\ \frac{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M\ :\ \tau}{\Gamma \vdash \exists\alpha.\, M\ :\ \exists\alpha.\, \tau} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash M\ :\ \tau' \qquad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash (M : \tau)\ :\ \tau} \qquad \textsc{Sigma}\ \frac{\mathcal{D}' \setminus (\{\beta\} \cup \mathit{dom}\,\Gamma') \subseteq \mathcal{D} \qquad \Gamma, (\forall\beta \prec \mathcal{D}), \Gamma', (\forall(\alpha \lhd \beta = \tau') \prec \mathcal{D}') \vdash M\ :\ \tau}{\Gamma, (\exists\beta \prec \mathcal{D}), \Gamma' \vdash \Sigma\,[\beta]\,(\alpha = \tau')\,M\ :\ \tau[\alpha \leftarrow \beta]}$$

$$\textsc{Open}\ \frac{\Gamma \vdash M\ :\ \exists\alpha.\, \tau \qquad \mathcal{D} = \mathit{dom}\,\Gamma}{\Gamma, (\exists\alpha \prec \mathcal{D}) \vdash \mathsf{open}\,[\alpha]\,M\ :\ \tau} \qquad \textsc{Nu}\ \frac{\alpha \notin \mathit{ftv}(\tau) \qquad \Gamma, (\exists\alpha \prec \mathcal{D}) \vdash M\ :\ \tau}{\Gamma \vdash \nu\alpha.\, M\ :\ \tau} \qquad \textsc{Weaken}\ \frac{\Gamma \vdash M\ :\ \tau \qquad \Gamma' \Vdash \Gamma}{\Gamma' \vdash M\ :\ \tau} \qquad \textsc{Sim}\ \frac{\Gamma \vdash M\ :\ \tau' \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash M\ :\ \tau}$$

Figure 4. Typing rules

$$\begin{array}{rl}
\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 \leq k \leq n & \textsc{Redex-Proj} \\
\mathsf{open}\,[\beta]\,\exists\alpha.\, M\ \rightsquigarrow\ M[\alpha \leftarrow \beta] & \textsc{Redex-Open} \\
\\
E\,[\Sigma\,[\beta]\,(\alpha = \tau)\,w]\ \rightsquigarrow\ \Sigma\,[\beta]\,(\alpha = \tau)\,E\,[w] \text{ if } \alpha \notin \mathit{ftv}(E) \text{ and } (\{\alpha, \beta\} \cup \mathit{ftv}(\tau)) \cap \mathit{etv}(E) = \emptyset & \textsc{Extrude} \\
\Sigma\,[\beta_1]\,(\alpha_1 = \tau_1)\,\Sigma\,[\beta_2]\,(\alpha_2 = \tau_2)\,w\ \rightsquigarrow\ \Sigma\,[\beta_1]\,(\alpha_1 = \tau_1)\,\Sigma\,[\beta_2]\,(\alpha_2 = \tau_2[\alpha_1 \leftarrow \tau_1])\,w \text{ if } \alpha_1 \in \mathit{ftv}(\tau_2) & \textsc{Sigma-Sigma} \\
\\
((\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 \leq k \leq n & \textsc{Coerce-Proj} \\
\mathsf{open}\,[\alpha]\,(u : \exists\alpha.\, \tau)\ \rightsquigarrow\ (\mathsf{open}\,[\alpha]\,u : \tau) & \textsc{Coerce-Open} \\
((u : \tau) : \tau')\ \rightsquigarrow\ (u : \tau') & \textsc{Coerce-Coerce} \\
\\
\nu\beta.\, \Sigma\,[\beta]\,(\alpha = \tau)\,w\ \rightsquigarrow\ \nu\beta.\, \Sigma\,[\beta]\,(\alpha = \tau)\,w[\beta \leftarrow \alpha] \text{ if } \beta \in \mathit{ftv}(w) & \textsc{Erase-Nu-Sigma1} \\
\nu\beta.\, \Sigma\,[\beta]\,(\alpha = \tau)\,w\ \rightsquigarrow\ \nu\beta.\, \Sigma\,[\beta]\,(\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\,[\beta]\,(\alpha = \tau)\,w\ \rightsquigarrow\ w \text{ if } \alpha, \beta \notin \mathit{ftv}(w) & \textsc{Erase-Nu-Sigma3}
\end{array}$$

$$\begin{array}{rcl}
E & ::= & [] \ \mid\ E\ M \ \mid\ M\ E \ \mid\ \mathsf{let}\ x = E\ \mathsf{in}\ M \ \mid\ E\,[\tau] \ \mid\ \{(\ell_i = M_i)^{i\in 1..k}\ ;\ \ell_{k+1} = E\ ;\ (\ell_i = M_i)^{i\in k+2..n}\} \\
 & \mid & E.\ell \ \mid\ \exists\alpha.\, E \ \mid\ \Sigma\,[\beta]\,(\alpha = \tau)\,E \ \mid\ (E : \tau) \ \mid\ \mathsf{open}\,[\alpha]\,E \ \mid\ \nu\alpha.\, E
\end{array}$$

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

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

Figure 6. Reduction rules

as demonstrated by the following derivation:

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

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

4.3.4 Reduction

The semantics of $\mathsf{F}^{\forall}$ is given by a call-by-value reduction strategy, described by a small-step reduction relation. The order of evaluation is left unspecified everywhere but on let-bindings. By contrast having a call-by-value strategy and a weak-reduction is essential.

Evaluation contexts are described in Fig.6. Note that, as opposed to Dreyer (2007), 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 in four groups.

The main group of rules 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: for instance, $(\mathsf{open}\ [\beta]\ M)[\beta \leftarrow \tau]$ is not defined. The type system ensures type substitution is always performed when it is well-defined.

The second group of rules implement the extrusion of Σ's through every other construct: EXTRUDE permits extrusion through evaluation contexts, provided this is valid with respect to scopes of type variables. To make it possible for two Σ's to commute, rule SIGMA-SIGMA substitutes the definition of the outer Σ to eliminate dependencies.

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

Finally, the fourth group is responsible for the erasures of restricted open witness definitions. ERASE-NU-SIGMA1 replaces the external name with the internal one. ERASE-NU-SIGMA2 replaces the type variable of a witness with the witness: note that the same substitution occurs in System-F while unpack-ing a pack-ed term. Finally, ERASE-NU-SIGMA3 really erases the restricted definition.

4.4 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* $\Gamma, (\forall\alpha \prec \mathcal{D}), \Gamma' \vdash M : \tau'$ *and* $\mathit{ftv}(\tau) \subseteq \mathcal{D}$*. Then the judgment* $\Gamma, (\forall(\alpha = \tau) \prec \mathcal{D}), \Gamma' \vdash M : \tau'$ *holds.*

Lemma 4 (Instantiation by substitution)**.** *Assume that* $\Gamma, (\forall(\alpha = \tau) \prec \mathcal{D}), \Gamma' \vdash M : \tau'$*. Then,* $M[\alpha \leftarrow \tau]$ *is well-defined and* $\Gamma, \Gamma'' \vdash M[\alpha \leftarrow \tau] : \tau'[\alpha \leftarrow \tau]$ *where* Γ'' *is the pair* $(\mathcal{E}'[\alpha \leftarrow \tau], (\mathcal{D}'; (\forall(\alpha = \tau), \mathcal{D})))$ *and* $(\mathcal{E}', \mathcal{D}')$ *is* Γ'*.*

The proof of subject reduction itself is not really informative, but it is particularly interesting that the proof is absolutely standard and almost straightforward, as this is not usually the case for other approaches to modules with generativity.

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

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

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

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

5. Related work

Russo (2003) justifies the meaninglessness of paths for module types, 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 (2003) introduces λ_{N}, a version of System-F with a construct to define abstract types and a mechanism of directed coercions. His abstract types can be automatically extruded to allow sharper type analysis, and are thus close to our Σ binder. His coercions resemble ours, though ours are symmetric, because they never cross the abstraction barrier. Although both systems seem kindred in spirit, they are subtly different, because they have been designed for quite different purposes: in particular, λ_{N} is only partially related to traditional existential types, since parametricity is purposely violated.

Dreyer (2007) defines RTG to deal with type generativity in the context of recursive modules. He introduces *type references* which can be written at most once. The creation of a type reference with "$\mathsf{new}\ \alpha\ \mathsf{in}\ M$" introduces a type variable in the scope of M that should be treated as a resource that can be set at most once, with his type reference update "$\mathsf{set}\ \alpha := \tau\ \mathsf{in}\ M$". Then, M and only M will see the concrete definition τ for α while other paths of the program will see α abstractly.

Technically, the treatment of these linear resources differ significantly from ours: his semantics employs a type store to model static but imperative type reference updates, whereas we just use extrusions of Σ binders. To a limited extent, these two approaches might be related by seeing our extrusion as a local treatment of his type store, as has been proposed for value references (Felleisen 1987). Dreyer uses an effect type system to guarantee the uniqueness of writing, which exposes the evaluation order in the typing rules of RTG, moving away from a logical specification, whereas we use *zipping* of contexts—a symmetric operation—to enforce sound openings and maintain a close correspondence with logic. *Intuitively,* we think of existential values as generating a fresh type when opened, while he sees them as functions in "destination passing style" (DPS).

Despite these strong technical differences, the two systems have similar constructs: the "new" primitive is similar to our ν binder; the "$\mathsf{set}\ \alpha := \tau\ \mathsf{in}\ M$" is related to the $\Sigma\,[\alpha]\,(\alpha = \tau)\,M$ construct. Note the use of a single type name here (as mentioned in §3.4).

The two systems differs 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 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 allow 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 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 $\mathsf{F}^{\curlyvee}$.

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.

RTG handles a richer type algebra than $\mathsf{F}^{\curlyvee}$: the support of higher order as well as recursive types in $\mathsf{F}^{\curlyvee}$ is planned.

In a previous work, Montagu and Rémy (2008) define a weaker version of $\mathsf{F}^{\curlyvee}$. They point out the inherent presence of type recursion. They manage to disable recursive types with a restriction of the zipping operator thus avoiding explicit dependencies. They take advantage of this restriction to define a type and semantics-preserving translation to System-F. As the transformation is non local and performs let-expansions, they show that $\mathsf{F}^{\curlyvee}$ allows for programming more modularly than System-F. The current version is an extension of the former presentation where constructs are further decomposed, the double vision problem is handled, and recursion is tracked more precisely via explicit dependencies.

In a second stage, they equip their language with a path system at the level of types. They claim this is a necessary component to write *compact* programs, that is completely orthogonal to type abstraction, and would also benefit to System-F. Their path system can be added unchanged to this extended version of $\mathsf{F}^{\curlyvee}$, providing the complementary but necessary ingredient to program with modules.

Conclusive remarks

We defined $\mathsf{F}^{\curlyvee}$, 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. Moreover, we gave a solution to the double vision problem that integrates smoothly with this system. In spite of the inherent presence of recursive types, we managed to detect and forbid them by enriching typing environments to track dependencies. 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 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.

We purposely limited the expressiveness of $\mathsf{F}^{\curlyvee}$ to the minimum that permits programming with modules in the ML-like style, directly. Concentrating on the core significantly helped understanding the foundations of type generativity. The integration of general purpose features such as recursive types, higher-order types, value references or recursion, still requires some work.

The addition of recursive type equations and their control, especially, is currently under work and necessitate some design adjustments: the problem is to generalize iso-recursive types when recursion is split across module boundaries (external type recursion). Recursive type equations are required to enable programming with—and hopefully provide a simpler foundations for—recursive modules and mixin modules.

Recursion (that is possibly ill-founded) can be added through the use of a classical fixpoint combinator: if restricted to pure terms, the soundness is straightforward. It is also possible to take the fixpoint of impure terms, but one should ensure extrusion can proceed across recursion. This is achievable by syntactic restrictions.

Higher-order types are motivated by Russo's work on modeling applicative functors.

We also 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. 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. We also believe that linear types can be integrated to the system by a light modification of the zipping operator and of the notion of purity.

This work realizes the first half of our project of defining a module language with simple and logical foundations. Indeed, $\mathsf{F}^{\curlyvee}$ misses a significant element to be considered as a scalable module language: 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 (Montagu and Rémy 2008), will be developed independently in another paper.

Of course, some form of type inference will eventually be needed in a real programming 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 and François Pottier for fruitful discussions, as well as anonymous referees for useful remarks on an earlier version of this paper.

References

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. Also available as research report 56, DEC Systems Research Center.

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

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

Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *2003 ACM SIGPLAN Symposium on Principles of Programming Languages*, 2003.

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

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: http://doi.acm.org/10.1145/174675.176927.

Delia Kesner and Stéphane Lengrand. Resource operators for lambda-calculus. *Information and Computation*, 205(4):419–473, 2007.

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

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

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

David MacQueen. Modules for standard ML. In *Proceedings of the ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802036.

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

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988. ISSN 0164-0925.

Benoît Montagu and Didier Rémy. Towards a simpler account of modules and generativity: Abstract types have *Open* existential types. Available electronically from `http://gallium.inria.fr/~remy/modules/`, March 2008.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of ECOOP*, 2003.

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

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

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

A. Judgments

A.1 Wellformed environments

$$\textsc{Ok-Empty}\ \frac{}{\varepsilon \vdash \text{ok}} \qquad \textsc{Ok-Forall}\ \frac{\Gamma \vdash \text{ok} \qquad \mathcal{D} \subseteq \mathit{dom}\,\Gamma}{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \text{ok}}$$

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

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

$$\textsc{Ok-DbEq}\ \frac{\Gamma, (\forall (\alpha = \tau) \prec \mathcal{D}) \vdash \text{ok}}{\Gamma, (\forall (\alpha \lhd \beta = \tau) \prec \mathcal{D} \uplus \{\beta\}) \vdash \text{ok}}$$

A.2 Wellformed types

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

$$\textsc{Wf-Record}\ \frac{\forall i \in 1..n,\ \Gamma \vdash \tau_i\,\text{wf} \qquad \forall i, j, i \neq j \Rightarrow \ell_i \neq \ell_j}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\}\,\text{wf}} \qquad \textsc{Wf-Empty}\ \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \{\}\,\text{wf}}$$

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

A.3 Compatible types

$$\textsc{Eq-Refl}\ \frac{\Gamma \vdash \text{ok} \qquad \alpha \in \mathit{dom}\,\Gamma}{\Gamma \vdash \alpha \equiv \alpha}$$

$$\textsc{Eq-Eq}\ \frac{\Gamma \vdash \text{ok} \qquad \forall (\alpha = \tau) \in \Gamma \vee \forall (\alpha \lhd \beta = \tau) \in \Gamma}{\Gamma \vdash \alpha \equiv \tau}$$

$$\textsc{Eq-Arrow}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' \qquad \Gamma \vdash \tau_2 \equiv \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \textsc{Eq-Empty}\ \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \{\} \equiv \{\}}$$

$$\textsc{Eq-Record}\ \frac{\forall i \in 1..n,\ \Gamma \vdash \tau_i \equiv \tau_i' \qquad \forall i, j, i \neq j \Rightarrow \ell_i \neq \ell_j}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \equiv \{(\ell_i : \tau_i')^{i \in 1..n}\}}$$

$$\textsc{Eq-Forall}\ \frac{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \tau \equiv \tau'}{\Gamma \vdash \forall \alpha.\, \tau \equiv \forall \alpha.\, \tau'} \qquad \textsc{Eq-Exists}\ \frac{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \tau \equiv \tau'}{\Gamma \vdash \exists \alpha.\, \tau \equiv \exists \alpha.\, \tau'}$$

$$\textsc{Eq-Sym}\ \frac{\Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash \tau \equiv \tau'} \qquad \textsc{Eq-Trans}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 \qquad \Gamma \vdash \tau_2 \equiv \tau_3}{\Gamma \vdash \tau_1 \equiv \tau_3}$$

A.4 Similar types

$$\textsc{Sim-Refl}\ \frac{\Gamma \vdash \text{ok} \qquad \alpha \in \mathit{dom}\,\Gamma}{\Gamma \vdash \alpha \lhd \alpha} \qquad \textsc{Sim-DbEq}\ \frac{\Gamma \vdash \text{ok} \qquad \forall (\alpha \lhd \beta = \tau) \in \Gamma}{\Gamma \vdash \alpha \lhd \beta}$$

$$\textsc{Sim-Arrow}\ \frac{\Gamma \vdash \tau_1 \lhd \tau_1' \qquad \Gamma \vdash \tau_2 \lhd \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \lhd \tau_1' \to \tau_2'} \qquad \textsc{Sim-Empty}\ \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \{\} \lhd \{\}}$$

$$\textsc{Sim-Record}\ \frac{\forall i \in 1..n,\ \Gamma \vdash \tau_i \lhd \tau_i' \qquad \forall i, j, i \neq j \Rightarrow \ell_i \neq \ell_j}{\Gamma \vdash \{(\ell_i : \tau_i)^{i \in 1..n}\} \lhd \{(\ell_i : \tau_i')^{i \in 1..n}\}}$$

$$\textsc{Sim-Forall}\ \frac{\Gamma, (\forall \alpha \prec \mathcal{D}) \vdash \tau \lhd \tau'}{\Gamma \vdash \forall \alpha.\, \tau \lhd \forall \alpha.\, \tau'} \qquad \textsc{Sim-Exists}\ \frac{\Gamma, (\exists \alpha \prec \mathcal{D}) \vdash \tau \lhd \tau'}{\Gamma \vdash \exists \alpha.\, \tau \lhd \exists \alpha.\, \tau'}$$

$$\textsc{Sim-Trans}\ \frac{\Gamma \vdash \tau_1 \lhd \tau_2 \qquad \Gamma \vdash \tau_2 \lhd \tau_3}{\Gamma \vdash \tau_1 \lhd \tau_3}$$