Internship report : Extending ML^F with Higher-Order Types

Gabriel Scherer

2010

This thesis presents the context and nature of the work during a six-month internship at Gallium, a research team of the Institut National de Recherche en Informatique et en Automatique (INRIA) which is specialized in the design, formalization and implementation of programming languages and systems.

My advisor was Didier Rémy, a senior member of the team, which specializes, among other things, in type systems and type inference. The internship subject was MLF, a type system developped by Didier and two of his former PhD students, Didier Le Botlan and Boris Yakobowksi.

MLF is a rich research topic which, despite interest and research by other people and teams outside Gallium, has not yet been integrated in the "mainstream" common knowledge of our discipline. A significant part of the intership time was dedicated to the study of the subject bibliography, and of the type system itself: despite very attractive properties, it is a complex system that has not yet been presented in a way making possible to thouroughly understand it in a calm afternoon of scholarly readings.

The first chapter of this document will be dedicated to an informal presentation of the MLF system. I'm not trying to be formal and exhaustive (nor could I, given size limits), only to convey the necessary intuition to understand the specific problematics of my internship subject. The reader interested in a more complete description of the different part of MLF will be redirected to the growing existing litterature.

The other parts of my internship dedicated to my work — Gallium team is rich in distractions; I won't describe here the quite enjoyable *pauses café(s)*, the very interesting research seminar, etc. — was dedicated to research and implementation.

The research part was dedicated to an extension of MLF with explicit polymorphism and higher-order types, which we could name MLFω. This had already been the subject of an internship by Paolo Herm's last year [Her09], and we hoped to extend and improve the work done on several points that I will explain in time.

The second and third chapters will be dedicated respectively to the extension of MLF with explicit polymorphism (*à la* System F), and higher-order types (Fω).

The implementation part, supposed to correspond closely to the subject of my internship, turned out to be of quite larger scope. Some of the latter parts of Boris Yakobowski's PhD thesis had not been translated into code, and I was responsible for their integration in the current MLF typer prototype. I also tackled some other aspects of various natures — term and types printing and display, built-in types and values, interactive toplevel — that were present in the former prototypes of both Didier Le Botlan and Boris Yakobowski, but were not up-to-date anymore with the latest presentation of the MLF metatheory.

I will not insist here on the software development and the interesting but delicate software engineering considerations that I had to handle during my internship.

1 MLF and the context of my internship

This part is an informal introduction to the subject of my master thesis, MLF. It is only intended to convey a good intuition of MLF and various related formal type systems — eMLF, gMLF, xMLF — and its relation to the well-known systems ML and System F.

1.1 ML, System F and MLF

1.1.1 ML as a type system

We're interested in ML as a type system, so we will keep the term-level complexity of the language minimal:

```
e ::=                         ML expressions
    | x, y, z                 variables
    | λ(x) e                  λ-abstraction
    | e₁ e₂                   application
    | let x = e₁ in e₂        local definition
```

ML makes a distinction between two levels of types: monomorphic and polymorphic types — usually named *type schemes*:

```
τ ::=                      ML monotypes
    | α, β, γ              type variables
    | τ → τ                function types

σ ::=                      ML polytypes
    | τ                    monorphic type
    | ∀(α) σ               type quantification
```

For a formal description of the ML type system, typing judgments and derivations, see [Mil78]. Informally, the crucial part of the type system is *let-polymorphism*:

- λ-bound variables get assigned *monomorphic types*
- let-bound variables get assigned *polymorphic types*

In other words, `let`-definition are the only place in a ML program where polymorphism is introduced, by *generalizing* the monorphic type of the bound term. Of the two following expressions, only the second typechecks, because `(fun x -> x)` carries a monorphic type of the form $\alpha \to \alpha$, wich is distinct from the generalized type scheme $\forall(\alpha)\alpha \to \alpha$ and cannot be used with different values of α.

```
(fun id -> (id 1, id true)) (fun x -> x)
let id = (fun x -> x) in (id 1, id true)
```

In particular, the typing rule for application is monomorphic: when typing applications, both the left and right side are assigned monorphic types. When using a let-bound variable in such a position — such as `id` in `id 1` of the above example — we implicitly choose a monomorphic *instance* of its polymorphic type, by replacing each ∀-quantified type variable with a monomorphic type — here the constant type `int`.

More generally, there is an *instance relation* between polymorphic types: σ_1 is an instance of σ_2 if we can transform σ_1 into σ_2 by replacing some of its quantified variable by monomorphic types, possibly introducing new free type variables, and finally quantifying over those free variables. For exemple, $\forall(\beta)\ (int \to \beta) \to (int \to \beta)$ is an instance of $\forall(\alpha)\ \alpha \to \alpha$.

ML is a very succesful type system, used in practical languages, due to the possibility of *principal type inference*. There are inference algorithms for ML, based on first-order unification, which have the two desirable properties:

- They will decide a correct polymorphic type for every term typable in ML.
- The chosen type is *principal*: every other valid ML type for that term is an instance of this type.

In other words, it is possible to infer, for each typable term, a most general polymorphic type.

Note that the term language of ML does not contain any syntactic construct related to typability and polymorphism: type annotations, etc. ML features *implicit* polymorphism.

1.1.2 Limitations of ML restricted polymorphism

After using the ML type systems for decades, programmers have become quite accustomed to its limitations. For example, the following term is not typable in ML:

```
λ(f) (f 0, f true)
```

(For that example, we have enriched our language with built-in constants and types, and tuples.)

This is not typable because `f` cannot be a monorphic function, as it takes input of different types, integers and booleans. `f` could be the identity function, of type ∀(α) α→α, but that would require assigning a polymorphic type to a λ-parameter, which is not possible in ML. However, the following definition is correct:

```
let f = λ(x) x in (f 0, f true)
```

This artificial example is in fact representative a deep problem of software engineering in languages using the ML type system. If a programmer encounters a program expression e making heavy use of the variable `f`, he cannot necessarily abstract over it: `(λ(f) e) f`. If `f` is used polymorphically in `e`, this abstraction would break typability: we are not necessarily able to factor out parts of the program.

1.1.3 System F

System F is a different type system featuring *explicit* polymorphism: the System F term language contains explicit indications of which terms are polymorphic and when we're taking monomorphic instances of them.

```
e ::=                      system F expressions
    | x, y, z              variables
    | λ(x:σ) e             λ-abstraction
    | e₁ e₂                application
    | let x = e₁ in e₂     local definition
    | Λ(α) e               introduction
    | e[σ]                 elimination
```

The following example declares the polymorphic identity function and uses it on the constant 5, in ML and System F:

```
  let id = λ(x) x in id 5                          (* ML *)

  let id : ∀(α)α→α = Λ(α) λ(x:α) x in id[int] 5      (* System F *)
```

System F terms, being fully explicit about polymorphism, are much heavier to write and read. They can, however, express richer polymorphism than ML:

```
λ(f : ∀(α) α→α) (f[int] 5, f[bool] true)
```

This function takes a polymorphic parameter: its type is `(∀(α) α→α) → (int*bool)`. This was not possible in ML, were the types on both sides of an arrow had to be monotypes. System F has *higher-ranked types*:

```
σ ::=                      System F types
    | α, β, γ              type variables
    | σ → σ                function types
    | ∀(α) σ               type quantification
```

`let`-bindings are usually not included in presentations of System F. They're useful here for homogeneity with the other term languages presented — ML and MLF. It is interesting to note that, similarly to ML, `let` cannot be macro-expanded as in untyped λ-calculus. It would be possible for an *annotated* let-binding:

```
let x:σ = e₁ in e₂      ⟶       (λ(x:σ) e₂) e₁
```

With the non-annotated `let`, however, the type σ is not syntaxically apparent. Note that this doesn't introduce any impliciteness or non-trivial inference in System F type system: System F terms have unique types, and σ can thus be uniquely determined from `e₁`.

1.1.4 Inference and System F

Is it possible to keep the expressivity of System F, while not being so explicit about types and polymorphism?

If we drop type annotations completely from System F terms, including type introduction and elimiation, we exactly get back the ML terms as defined in subsection ML as a type system. But there are terms which have a System F type, while they are not typable in ML. Is there an algorithm to infer the System F type of a term? An easier question would be: given the System F type of a term, can we automatically recover type annotations, introductions and eliminations?

The answer to these questions is *no*. Those two questions — which are confusingly named *type reconstruction* and *type inference*, or *typability* and *type checking* — are undecidable [Wel94].

In other words, it is not possible to get the expressive power of System F without using some type annotations. System F terms, using explicit polymorphism, use *a lot* of type annotations that often feel redundant. Is it possible to do better?

This question is an active research topic. The programming languages based on the ML type system have evolved and are looking for more expressivity. They have included extensions of the ML type system, allowing some restricted forms of higher-order polymorphism, available to the user that correctly inserts type annotations at dedicated places. The idea is to bridge the gap between ML and System F, by finding the right compromise between inference and polymorphism annotations.

1.1.5 Principality in rich type systems

Rich type systems make type inference harder: more program are typable, which means more work to do for the inference algorithm. However, such a monotonic relation does not hold for principality: enriching the type system can make principality *easier*.

The simply typed λ-calculus — which is essentially our presentation of ML, restricted to monotypes only — has an easy type inference algorithm based on first-order unification, but does not benefit from principality. For example, the identity function (λ(x) x) accepts any monotype (τ→τ), none of them being more general than all others.

By enriching the type system with the polymorphic types of ML, we gain principality: the new type scheme ∀(α) α→α is able to express the commonality in all the derivable monotypes. Type quantification is exactly what we needed to express that, for a given term, some parts of the inferred types do not matter.

In other words, principality of type inference requires expressiveness at the *type level*. During the type inference process on a term, one often encounters different possibilities. If you cannot express the alternatives in their full generality in the result type, you cannot have principality.

1.1.6 From System F to MLF

MLF is not a possibility in a continuum from ML to System F. It is an extension of System F, that regains principality by enriching the expressiveness of the type level.

Consider the following `choose` function:

```
choose := (λ(x) λ(y) if true then x else y)
choose :  ∀(α) α→α→α
```

If we pass it the identity function as argument, what is the type of the result?

```
id := λ(x) x
id :  ∀(α) α
```

In ML, which does not allow polymorphism under arrows, the result has type:

```
σ₁ :=  ∀(β) (β→β) → (β→β)
```

But in system as expressive as System F, there is a second possibility:

```
σ₂ := (∀(β) β→β) → (∀(β) β→β)
```

Those two types are admissible, and none of them is more general from the other. We cannot have principality with System F type language.

It is quite clear than σ_2 is not an instance of σ_1: by going from an inner quantification (σ_2) to an outer quantification (σ_1), we have lost the information that the polymorphism is *local*.

What information have we lost in σ_2? It is sound to consider $(\forall(\beta)\ \beta\to\beta)\to(\text{int}\to\text{int})$ as an instance of σ_2. It is not sound, however, for $(\text{int}\to\text{int})\to(\text{int}\to\text{int})$. Indeed, terms with type σ_2 are functions that may use their argument polymorphically, we cannot pass them a $(\text{int}\to\text{int})$.

The information we have lost is the information that the argument and the result types are equal. When the type was written $\forall(\alpha)\alpha\to\alpha\to\alpha$, that *equality* was made explicit by the use of the same variable α for all components of the type. If we knew, somehow, that the instance and argument types of a term of type $(\forall(\beta)\ \beta\to\beta)\to(\forall(\beta)\ \beta\to\beta)$ are equal, then we could soundly instantiate it to $(\text{int}\to\text{int})\to(\text{int}\to\text{int})$.

MLF provides a new way to express equality information inside types. The most general type for `choose id`, expressing both local polymorphism and subtypes equality, is written:

```
∀(α ≥ ∀(β)β→β) α → α
```

1.1.7 A high-level description of MLF types

Such quantification using a ≥-bound is called *flexible quantification*. We can describe flexible MLF types with the following grammar:

```
σ ::=                    flexible MLF types
    | α, β, γ            type variables
    | ⊥                   bottom
    | α → β              function types (using only variables)
    | ∀(α ≥ σ₁) σ₂       flexible quantification
```

Any type is an instance of $\perp$, and instances of $\forall(\alpha\geq\sigma_1)\sigma_2$ may specialize both σ_1 and σ_2. The usual ML schemes $\forall(\alpha)\sigma$ can be written $\forall(\alpha\geq\perp)\sigma$.

Note that the arrow type (and, more generally, any of the traditional constructors of monomorphic types such as tuples) are expressed monomorphically, using only variable. $(\alpha\to\beta)\to\gamma$ is written $\forall(\delta\geq\alpha\to\beta)\delta\to\gamma$. This presentation does not change expressivity, but helps separating the first-order structure and the polymorphism structure of a term: the only parts relevant for polymorphism are in quantifications, we do not have to look deep into nested arrows to find instantiable subtypes.

This is only useful when the bound actually carry polymorphism. Syntactic sugar can be defined so that "inert" (not further instantiable) bounds such as $\alpha\to\beta$ can be inlined.

For example, we can express both System F types for `choose id` as instances of $\forall(\alpha\geq\forall(\beta)\ \beta\to\beta)\ \alpha\to\alpha$:

```
∀(β) ∀(α≥β→β) α → α                   loss of local polymorphism
∀(α'≥∀(β)β→β) ∀(α≥∀(β)β→β) α' → α    loss of sharing information
```

The second type is actually not a faithful transcription of the System F type $(\forall(\beta)\beta\to\beta)\to(\forall(\beta)\beta\to\beta)$. This System F type can represent terms where the polymorphism of the left-hand side is *required*. Using MLF flexible bound, we can only express types were polymorphism is *available*.

Therefore, we add a *rigid binding* to express required polymorphism:

```
σ ::=                MLF types
    | ...            [MLF flexible types](def-mlf-flexible)
    | ∀(α=σ₁) σ₂     rigid quantification
```

When instantiating a rigid quantification, we may *not* specialize the bound type σ_1. The System F type can then be expressed:

```
∀(α'=∀(β)β→β) ∀(α≥∀(β)β→β) α' → α    restricted polymorphism
```

This type is an instance of the former one: $\forall(\alpha\geq\sigma_1)\sigma_2$ can be weakened into $\forall(\alpha=\sigma_1)\sigma_2$. So we have actually used two steps to the System F type: first losing the equality information, then forgetting than the left hand side can still be refined.

1.1.8 MLF and type annotations

As we have seen earlier, to construct terms using higher-order polymorphism, it is necessary to use a certain amount of type annotations in terms. Where are the polymorphism annotations in MLF terms?

The answer is refreshingly simple: polymorphism is made explicit in MLF terms by using *constants*. Those constants introduce polymorphism in the form of rigid bounds. For each (polymorphic) type σ, we introduce a constant $\texttt{coerce}\left\{\sigma\right\}$ with the following type:

```
∀(α=σ) ∀(β≥σ) α → β
```

Those constants can be considered as coercion functions. When applied to a term, they force the inference algorithm to give a polymorphic type to this term — exactly σ, and not one of its monomorphic instances: this is the role of rigid quantification.

For example, the former type ($\forall(\alpha=\forall$*(β)β→β) ∀(α≥∀(β)β→β) α* $\to\alpha$) can be constructed with the following term:

$$\lambda(x)\ (\texttt{coerce}\left\{\forall(\beta)\ \beta\to\beta\right\}\ x)$$

In order to get more familiar-looking annotations, we define the following syntactic sugar:

$$\begin{array}{lcl} \texttt{let}\ x{:}\sigma = e_1\ \texttt{in}\ e_2 & \longrightarrow & \texttt{let}\ x = \texttt{coerce}\left\{\sigma\right\}\ e_1\ \texttt{in}\ e_2 \\ \lambda(x{:}\sigma)\ e & \longrightarrow & \lambda(x)\ (\texttt{let}\ x = \texttt{coerce}\left\{\sigma\right\}\ x\ \texttt{in}\ e) \end{array}$$

The former example can then be written simply

```
λ(x : ∀(β) β→β) x
```

1.1.9 The good properties of MLF

The coercion functions are actually the only part of the MLF type system that introduce rigid quantifications. All types generated during the inference of `let` bounds, functions and applications use flexible quantification.

We have the two following properties:

1. The MLF typable terms that do not use coercions are exactly the ML

typable terms. In particular, all ML terms are typable in MLF without requiring additional type annotations.

1. Terms admitting polymorphic types need not be annotated, only

function parameters that are *used polymorphically* need to.

Additionnally, typability of MLF terms is very stable, as it is preserved by a large range of transformations. For example, the following two examples of rewriting preserve the set of typable MLF terms:

```
e₁ e₂          ⟶       (λ(f) λ(x) f x) e₁ e₂
e₂[x←e₁]     ⟶      let x = e₁ in e₂      (* provided x appears free in e₂ *)
f              ⟶       λ(x) (f x)               (* when f is a function *)
```

Note in particular that, in the latter example, no annotation is required on `x` even if `f` requires a polymorphic type for its parameter: in that case, `x` is polymorphic but not *used polymorphically*, it is only passed to a function.

1.2 gMLF: Graphical MLF types

The formal definition of MLF syntactic types, instance relations, unification and inference algorithms are the main result of Didier Le Botlan's PhD thesis. They have proved very technical and difficult to get right. Furthermore, the resulting unification algorithm is not very efficient. Extending the type system with richer features also involved a large amount of work.

Intuitively, one of the reason why the metatheory of syntactical MLF types is so technical is the rich equivalence relation between types. For example, ∀(α≥σ)α is equivalent to σ, and the syntactical unification algorithm needs to be aware of it, burdening its description and correctness proof.

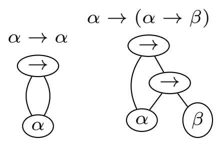
Boris Yakobowski's PhD thesis develops a new approach of MLF types. The idea is that, as the MLF quantifications are used to represent *sharing* information between subtypes, an appropriate representation for such types may not be a syntactic tree, but a *graph*. The graphical representation was not new, as it is informally used in Didier Le Botlan's thesis, but Boris gave it a formal status that led to a new, simplified and more maintainable definition of the MLF type system, named gMLF, which enables an efficient inference algorithm.

1.2.1 Graphic types

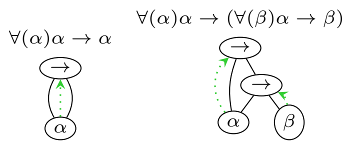
As we have noted earlier, MLF types provide a clean separation between the monomorphic structure and the polymorphic binders of a type.

Graphic types preserve this separation. A graphic type is the superposition of two layers, the *structure graph* and the *binder tree*.

The structure graph is a tree with shared nodes, which describes the monomorphic structure of the term.

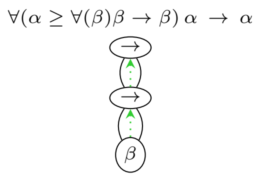


The binding tree is given by an arrow for each structure node, that goes *up* in the structure graph. They correspond to the variable scope. It is a tree because only one arrow can leave from a given node.

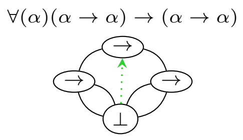


$\forall(\alpha)\alpha \to \alpha$
$\forall(\alpha)\alpha \to (\forall(\beta)\alpha \to \beta)$

In syntaxic types, we need variables to express sharing. The rich quantification of MLF was introduced to express sharing between arbitrary types, not leaf type variables. In a gMLF graph, this is very naturally expressed:

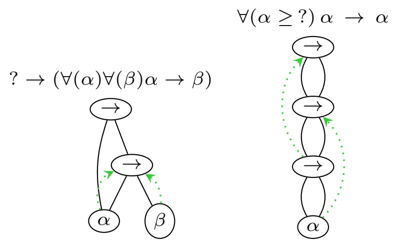


$\forall(\alpha \geq \forall(\beta)\beta \to \beta)\,\alpha \;\to\; \alpha$



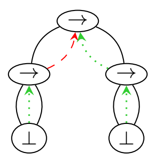
$\forall(\alpha)(\alpha \to \alpha) \to (\alpha \to \alpha)$

Note that there are restrictions on the binding arrow positions. I will not describe precisely the well-formedness condition, wich is too technical for this document, but instead give two representative examples of ill-formed graphs. In the first case, a binding arrow is "too low" : must be higher than any structure edge referring to the node content. In the second case, a "well-bracketing" condition is broken, as two binders "intersect".



$? \to (\forall(\alpha)\forall(\beta)\alpha \to \beta)$
$\forall(\alpha \geq ?)\,\alpha \;\to\; \alpha$

Finally, we use another kind of binding arrow to represent rigid flags.



As you may have noticed, variable names are not useful anymore in gMLF graphs : the binding arrow position is enough to indicate the scope of a quantifier. Therefore, we use only (⊥).

We may also notice that the structure graph can be decomposed in two separate components :

- a structure *tree*, without sharing
- a *sharing* relation between nodes

We can therefore present gMLF types on the base of the four following components:

1. the structure tree
2. the sharing relation
3. the binding tree

4. the flag binding association

When transformed into a formal definition of gMLF types, this presentation allows to handle each aspect separately during formal developments. It has been developped in more recent works, to appear.

1.2.2 Graphic types and the bound aliasing problem

It is interesting to compare syntactic and graphic types, and in particular the differences between syntactic quantifiers and binding arrows. It is interesting, because it is not a perfect fit : while we gained the ability to express sharing, we also lost some precision : there are particularities of syntactic types that cannot be expressed in the graph model.

For example, consider the type $\forall(\alpha)\ \mathtt{int}$: it has no direct graphical representation. The problem is that we cannot represent the $\forall(\alpha)$ quantification by a binding arrow, as we wouldn't know where to place it: as α is not used in the type, there is no structure node corresponding to α.

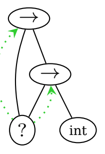
We also lost the ability to distinguish $\forall(\alpha)\forall(\beta)\sigma_1\to\sigma_2$ and $\forall(\beta)\forall(\alpha)\sigma_1\to\sigma_2$. In both cases, the arrows for α and β will point to the $(\to)$ node at the top of the graph. Binding arrows are unordered.

Those two differences can be seen as the strength of the graphic model. When doing type inference, we do not care about unusued type variables or quantifier ordering. In fact we do not *want* to care about it : the less the inference algorithm is burdened with such details, the better.

However, we must keep in mind that there may be mismatch between the syntactic and the graphic types. The user is familiar with the behavior of syntactic types, and may be sadly surprised if she was confronted to such differences.

There is a third important difference between syntactic and graphic types, which is probably the most problematic. How is the type $\forall(\alpha\geq\bot)\ \alpha\ \to\ \forall(\beta\geq\alpha)\ \sigma$ represented ? In general, for example with the type $\forall(\alpha)\ \alpha\ \to\ \forall(\beta\geq\sigma_1)\sigma_2$, there is no specific difficulty : α is represented by a $(\bot)$ node bound to the top, and β is an arrow from the top node of (σ_1) to the $(\to)$ node corresponding to $\alpha\ \to\ \ldots$.

$$\forall(\alpha)\alpha \to \forall(\beta? \geq \alpha)\beta \to int$$



But in the special case where σ_1 is itself a variable such as α, we have a problem : how could we add a binding arrow from α to the $(\to)$ node, when α is already bound at the top of the type ?

In other words, we have a problem because we have two different type variables that denote exactly the same type. That makes two different binding arrows that want to have the same source node, which is not possible.

This is called the *bound aliasing problem*. There are two simple ways to solve it, neither of them entirely satisfactory :

1. Raise a type error in case of bound aliasing: the user has to be careful.

2. Choose one of the arrows, and forget the other. For the result to be well-formed, we must choose the arrow bound the highest. In our example, it's α. If we drop the β quantification, we get the following type : $\forall(\alpha)\ \alpha\ \to\ \sigma[\beta\leftarrow\alpha]$. The user may be surprised.

This bound alias problem is particularly frustrating because of the discontinuity it creates inside types : not all types are equal. We may be perfectly fine using `unit → α` in a quantification, but changing it to α instead may deeply disturb the resulting type.

This bound alias problem has been constantly present during my internship. It can create special cases that can make reasoning, proofs, or even direct intuition harder. Of course, the type inference engine will not be concerned by the bound alias problem : it will infer correct types, without aliased bounds. It becomes problematic when we give the user to express some types herself, instead of letting the inference engine guess them: it is dangerous for explicit polymorphism. Unfortunately, this is precisely the subject of my internship.

1.2.3 Graphic instance relation

The *instance relation* is very important in a type system designed for type inference. Given any value with type σ, what other types, less general than σ, does it have ?

In the syntactic case, we have seen than the two main operations are :

- replacing a bound variable with some type structure: from $\forall(\alpha)\alpha\to\alpha$ to $\mathtt{int}\to\mathtt{int}$

- extruding quantifiers to the outer scope of the type: from $\mathtt{int}\to(\forall(\alpha)\alpha\to\alpha)$ to $\forall(\alpha)\ \mathtt{int}\to(\alpha\to\alpha)$

Note that, in the second case, not all extrusion are permitted. For example we may not extrude the quantifier of $(\forall(\alpha)\alpha\to\alpha)\to\mathtt{int}$: a value of that type may expect a polymorphic function as parameter.

Graphical types are quite similar. The general instance relation can be decomposed in small "operations" that represent, in a sense, "atomic" moves from a type to a slightly less general one. There also are restrictions on which operations are allowed in which part of the types. Those restrictions are specified by a *permission* system that we will describe.

For gMLF, we may describe four separate insantiation operations, one for each "component" of a gMLF type :

- On the structure tree, the *graft* operation, which replaces a $(\bot)$ node with any type.

- On the sharing relation, the *(un)merge* operation, which enrich the sharing relation to consider two given nodes as equal, or a given node with two different ancestors as two separate nodes.

- On the binding tree, the *raise* operation, that raise a binding arrow to the (binding) ancestor of a given node.

- On the binding flags, the *weaken* operation, which change a flexible flag into a rigid one.

We have already explained how, going from $\forall(\alpha\geq\bot)\ \alpha\to\alpha$, we may obtain either $\forall(\beta)\ (\beta\to\beta)\ \to\ (\beta\to\beta)$ or $(\forall(\beta)\beta\to\beta)\to(\forall(\beta)\beta\to\beta)$. It is instructive to review the process as a series of gMLF instantiation operations.

In either case, the first operation is the grafting of $\forall(\beta)\beta\to\beta$ on $\alpha\geq\bot$, which produces $\forall(\alpha\geq\forall(\beta)\beta\to\beta)\ \alpha\to\alpha$, the most general type inferred for `choose id`. Then :

```
∀(α≥∀(β)β→β) α→α
⟶ ∀(β) ∀(α≥β→β) α→α                  raise

∀(α≥∀(β)β→β) α→α
⟶ ∀(α₁≥∀(β)β→β) ∀(α₂≥∀(β)β→β) α₁→α₂    unmerge
⟶ ∀(α₁=∀(β)β→β) ∀(α₂≥∀(β)β→β) α₁→α₂    weaken
```

Note : In fact, we will only consider the `merge` operation, not unmerge. I will not discuss details, but the general idea is that, when we unify two types σ_1 and σ_2, we do not want to know if one is an instance of the other, but if they're both instances of a common, less general type σ_3 — to be determined by the unification process. It's okay if the merging goes in only one direction : we merge as much as possible on both sides, and check if they can meet for some σ_3. A more detailed explanation can be found for example in Boris Yakobowski's thesis [Yak08].

1.2.4 Binding contexts and permission

The intuitive meaning of MLF flexible quantification is to permit instantiation operations on a given subtyped (denoted by the quantification variable). On the contrary, rigid quantification preserve polymorphism by forbidding instantiation operations on the rigid variable : `∀(α=⊥)α→α` may not be intantiated into `int→int`.

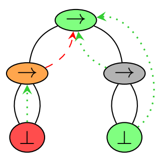
Therefore, if we want to modify a given part of a MLF type, we must check if this subtype was introduced by a flexible or rigid quantification. But this is not enough. For example, the β variable of `∀(α=∀(β≥⊥)β→β)α→int` is flexibly bound, but may not be changed into `int` nonetheless : it is restricted by an outer rigid binding `∀(α=..)`.

In the general case, to know the permission of a given type variable — and in MLF, every subpart of a type is a type variable — we must consider its *binding path*, or *binding context*, the succession of bindings from that variable to the top of the type.

There are three different situations :

- A node is *flexible*, or *green*, if its path to the root of the type has only flexible bindings. As a regular expression on the flags path: `≥*`.
- A node is *rigid*, or *orange*, if it is rigidly bound on a *green* node: `= ≥*`.
- A node is *locked*, or *red*, if it is flexibly bound to a *rigid* or *locked* node : `≥ (≥|=)* = ≥*`.

In a graphical representation, we may color each node according to its context, uniquely determined from the graph :



This graph uses a fourth context that we have already described : *inert nodes*, in gray, are the nodes that carry no polymorphism: they're not `(⊥)` nodes and not flexibly bound to an inner `(⊥)` node. As we will see, those nodes have liberal permission, independently of their context : with no power, comes no responsibility.

We then give the following permissions :

- Only the green `(⊥)` nodes can be grafted.
- All green, orange and inert nodes can merge or raise their bound.
- Inert nodes may always weaken their binding. Green node may only weaken if they're bound on a green node.
- In particular, red nodes may not do anything.

1.2.5 Unification for gMLF types

The unification algorithm for gMLF is quite simple. First, we unify the graph structures, using the usual first-order algorithm. Then, in a second pass, we compute the new binder tree and flags, by computing the least common ancestor of all nodes merged by the structure unification, and their weakest flag. During this second pass, we check permissions for each modified node. If a permission was not respected, the unification algorithm fails with a type error.

I will describe the first-order unification part in more detail in the third chapter of this document, as it had to be modified for higher-order types.

1.3 The gMLF inference process: solving graphic constraints

Similarly to virtually every modern type inference algorithm for ML [PR05], the gMLF inference is constraint-based. The idea is to read the program of the user, and translate it into a big graph using both gMLF type fragments, and additional "constraints" constructions. Then, we choose a resolution strategy, that walks the graph and solves the constraints. When all constraints are solved, the result is a gMLF type representing the most general type of the user input. If the resolution fails, the input was ill-typed.

1.3.1 Graphic constraints

Graphic constraints, also developed by Boris Yakobowski, extend gMLF types with three additional constructs :

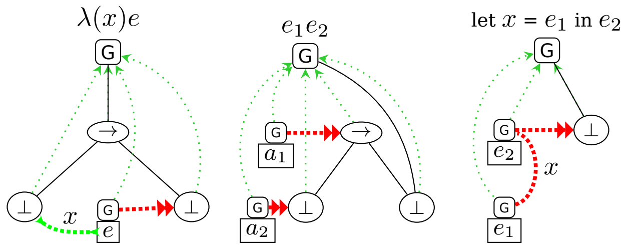
- *unification edges*, a constraint to unify two gMLF types
- *instance edges*, a constraint that force one gMLF types to be an instance of another
- *(G) nodes*, a new kind a structure node with specific properties, useful during constraint-solving.

I will not describe `(G)` nodes in depth. They're actually a non-trivial part of the gMLF theory. I was told by Boris that he and Didier spent months trying to remove the `(G)` nodes from the gMLF theory, until they realized they were absolutely necessary !

Vaguely, the `(G)` nodes form an inversed tree structure, on top of the constraint graph, that mirrors the structure — Abstract Syntax Tree — of the user input program. They're always flexibly bound, and a `(G)` node is always bound on a `(G)` node (or the root). They're useful to guarantee that every two nodes in the constraint graph, even if they were build from distant part of the input program, have a common flexible ancestor. In a sense, they enable non-local communication of type information.

1.3.2 Translating MLF terms into graphic constraints

We recursively translate a given MLF term by translating the head construct, and recursively translating subterms as described by the following translation rules. A square box with a `(G)` on top is the result of a recursive call of the translation algorithm on the given subterm. Instance edges are in red, unification edges in green.



There is an environment passing implicit to those figures. `e` in the λ-abstraction case and `e₂` in the `let` case are translated in an environment were the variable `x` is bound. It is important to remember if a given variable was bound by a `let` or a `λ`, because when we encounter a variable occurence the behavior is different:

- if the variable is λ-bound, we just reuse the given node (this is the meaning of the green unification constraint in the λ-case)
- if the variable is `let`-bound, we build a `(⊥)` node under a `(G)` node, and add an instantiation constraint from the definition site (as depicted on the figure) to the `(G)` node.

As a consequence, only `let`-bound variable behave polymorphically, that is, can be reused with different types at different places.

Note : I did not describe the translation for coercion functions. There is nothing very spectacular : they produce gMLF types with a main (→) node, left child rigidly bound and right child flexibly bound.

1.3.3 Expansion and instantiation

The graph resolution process relies on two resolution procedures, for unification and instance constraint edges. I have already described the unification algorithm.

The instantiation algorithm is based on an *expansion* procedure. Basically, when given the constraint that g_1 must be an instance of g_2, we make a fresh copy of g_2 and unify it with g_1. This will modify g_1 to be less general than g_2, or fail with a unification error.

Taking a fresh copy of a subgraph is the expansion procedure. There are two important points that are the core of the MLF inference algorithm:

1. Only the *interior* of g_2 is copied. The interior is the subset of the structural children of g_2 that are bound to g_2 or in its interior. In other words, they're the set of nodes in the scope of g_2. Other nodes, outside g_2, are "in the global environment" from g_2 point of view, and may not be modified by the expansion procedure. In order to preserve well-formedness of the graph, there is however a special case for *frontier nodes*, the first structural descendant to be outside the interior : they're not copied like the interior, but a fresh (⊥) node is created for each frontier node, and is unified to its corresponding frontier node after the expansion. This frontier handling will be discussed in more details in the third chapter.

2. During the expansion, a *binding reset* is performed. First, binding arrows pointing to the (G) node directly above g_2 are *lowered* to g_2. This is the only place in the gMLF routines where nodes get lowered. The idea is that, when we take an instance of g_2, we generalize the variables in the immediate environment: the binding reset creates local polymorphism. We then examine the binding of the node g_2 itself and, if it is a rigid binding, we *strengthen* it to a flexible binding. Again, this is the only place where this is done. The idea is that rigid bindings are there to preseve *inner* polymorphism, polymorphism in the subtypes (argument type of a function, etc.). Once it is at the root of the subgraph being expanded, it is no longer meaningful to block instantiation.

1.4 xMLF: a Church-Style Intermediate Language for MLF

MLF terms are implicitly polymorphic: types and polymorphism are not part of the term syntax. What would an explicit language for MLF look like?

Boris Yakobowski answered this question in the end of his thesis, by presenting xMLF, an explicitely typed language using MLF types. This language is quite similar to System F, except that the quantifier introductions are bounded $(\Lambda(\alpha \geq \sigma)\ e)$, and the type eliminations are replaced by finer-grained *type computations* that describe the delicate operations of an MLF instance relation.

xMLF has a dynamic semantic given by rewrite rules which was proved confluent, and preserve types — subject reduction and progress. Recent work by Giulo Manzonetto and Paolo Tranquili [MT10] has also proved xMLF strong normalization.

Boris Yakobowski finally presented an elaboration step that translated solved gMLF constraints into xMLF terms. With that final stone, it was possible to imagine a complete MLF implementation: code source is written in the MLF term syntax, the typer build a gMLF constraint from the code, solve it and infer types, and an xMLF term is elaborated from the solved constraint, which is finally evaluated by reduction.

For a detailed presentation of the xMLF language and the translation process, see [RY08].

1.4.1 Own implementation work

As the xMLF work came up late in Boris's thesis, he did not have the time to implement a prototype of translation from gMLF contraints to xMLF terms. The first task of my internship was to implement gMLF to xMLF translations on top of the most recent MLF prototype.

The translation is not direct: the constraint solving algorithm of gMLF do not preserve an history of instantiation steps that would be needed to construct the appropriate xMLF type computations. It is therefore necessary to make a second pass on solved constraint edges, replaying an instrumented resolution process to extract atomic instantiation steps. As "atomic instantiations steps" represent slightly different operations in the graph-driven gMLF instantiation process and the syntax-oriented xMLF type computations, the last part of the translation is not direct either.

In retrospect, that was a much larger amount of work than expected. gMLF implementations are quite delicate in that they handle graphs, with possible cyclicity. While modern ML languages (all the implementation work was done in OCaml) are well-equipped to deal with trees with rich recursive structure, graph processing is not as happy and still often requires effectful traversals and mutable structures. The rich invariants on graphic constraints and subtle pre/post-conditions of graph manipulations are delicate to maintain and debug.

For example, the gMLF unification algorithm is formulated in two parts: first, a first-order unification is run on the concerned nodes, then a second traversal checks that permission and binders were respected. The correctness conditions of the verification traversal are expressed in terms of both the old (pre-unification) graph state, and the new graph state. Maintaining those two states available while representing sharing with an imperative Union-find data structure is really tricky.

This work built upon an OCaml implementation of the xMLF language itself, which I developped as a first step to get familiar with the type system. It was the first time I used a De Bruijn based representation for binders, which comes with its share of subtleties; maybe particularly so in xMLF, which has two different kind of variables (type variables and term variables) and five binding constructs. However, xMLF is still a reasonably simple system based on usual syntactic rules, and its expliciteness make the typer checker straightforward, so I could concentrate my efforts on the gMLF system itself.

2 Mixing Implicit and Explicit polymorphism

MLF polymorphism is as powerful as System F polymorphism. However, MLF only provides implicit polymorphism: it is not *necessary* to specify where type introductions and elimination occurs, but it is not *possible* either. For various reasons, it is desirable to add explicit polymorphism, exactly as in System F, to MLF. It would make possible to embed a System F term, as is, into a MLF program. More importantly, it is necessary for the latter extension we have in sight, MLFω: as we will see, implicit polymorphism alone cannot handle higher-order types.

Designing an elaborate inference algorithm for an implicitly typed language is hard. In contrast, checking explicit type introductions and eliminations is rather easy. It so seems that adding explicit polymorphism to MLF, where the hard implicit work is already done, should be an easy task.

It turns out that it is actually a very subtle question, for two different reasons:

1. While it is easy to consider explicit polymorphism in isolation, the difficulty is in designing how it *interacts* with implicit polymorphism. For example, if we explicitly introduce a type variable to construct a polymorphic term, we expect, as in System F, to later explicitly instantiate it using a type elimination. Surprisingly, it is difficult to ask the typer

not to instantiate it implicitly, as he would for implicit polymorphism placed under its responsibility. « L'enfer est pavé de bonnes intentions. »

2. In graphic types, there is no notion of *type variable*. The polymorphism is not directed by variables, but by the binding arrows. Type introduction in System F is associated to a type variable binding, and reproducing its behavior (lexical scope, etc.) requires a representation closer to ordinary expression variable binders already present in MLF (`let` and λ), which are encoded using structure edges. There is therefore a tension between polymorphism, which would suggest a solution based on binding arrows, and lexical binding, which demands structure edges; those two concepts are very different in nature.

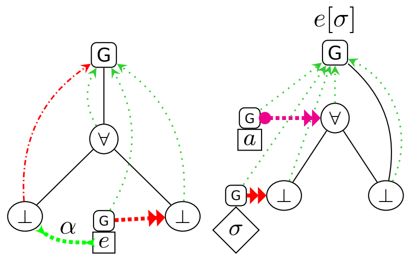
2.0.2 Paolo Herms's solution

My internship is actually the second internship on the same subject, MLFω: the first internship was done last year — 2009, spring and summer — by Paolo Herms. In his report [Her09], Paolo described his proposed solution to the present questions. He came up with a satisfying solution to the implicit-explicit mix, which I will now describe, and an early prototype implementation.

Paolo added one new type of structure nodes, *forall nodes*, which I will depict (∀). They are used to represent explicit quantification: the left child of a (∀) node represents a Λ-variable, and its right child the Λ-body.

He also added a new kind of binding, *explicit bindings*. This new binding arrow has very restrictive permissions: it may not be grafted, weakened or raised. Only merging is allowed. Additionally, it is possible to implicitly weaken a flexible binding into an explicit one.

The next figure represents translation rules from an MLF syntax, extended with explicit introduction and elimination, to the extended graph constraints. The similarity with the translations of λ-abstractions and function application is striking.



Note that, in the elimination case `e[σ]`, the instantiation edge between `e` and the (∀) node is not the usual instantiation constraint. It is a new kind of constraint edge, *explicit instance edges*, added to handle explicit bindings. The only difference with the usual instantiation edge is the following: during resolution, after expanding the (G)-node `e`, we change the (∀)-node that is at the top of the term by changing its left child (the Λ-variable) from *explicitly* bound to *flexibly* bound.

This rule assumes that the translation of `e` always begins with a head (∀)-node. If not, the constraint resolution fails with a typing error. This is natural: it means that only explicitly quantified terms may be used in type eliminations.

At the type level, we wish to distinguish implicit and explicit polymorphism. We therefore annotate ∀-types with their *explicitness*: ∀e(α) represents explicit polymorphism, and ∀i(α≥σ) is the usual implicit polymorphism.

From a high-level point of view, the life cycle of explicit polymorphism during the gMLF typing process is the following: for each explicitly polymorphic term Λ(α)e, a (∀) node is created, with an explicitly bound variable (left child). The explicit bound blocks implicit instantiation by the type inference engine, until the polymorphism is explicitly eliminated: elimination `e[σ]` is translated into an explicit instance edge, which *removes* the explicit bound, in a sense *unlocking* the Λ-variable polymorphism. From that point, the inference engine is free to instantiate the type of the bound (which was initially forced by the elimination translation to be σ), as with any other implicitly typed term.

Paolo also extended the prototype with those new structures.

2.0.3 Issues with Paolo solution: conversions between implicit and explicit polymorphism.

While Paolo's solution worked well, it also had drawbacks that Didier Rémy hoped to overcome. It was only one point in a seemingly large design space, and he asked me to reconsider other possibilities with the goal of improving it.

The main issue with the solution was the use of `(∀)` nodes, a new kind of structure nodes. Before that addition, polymorphism in MLF was handled solely by binding arrows, giving the whole system a satisfying orthogonal aspect. `(∀)` reintroduced polymorphism in the *structure* part of gMLF constraints.

This was not only a question of taste: Didier wanted implicit and explicit polymorphism to be similarly represented, so that they could easily be converted one into the other. We wished to be able to write a term in an explicit or implicit manner and, by some kind of annotation, convert between the two representations. For example, converting System F polymorphic identity into an implicitly typed equivalent would be written — introducing a new :> keyword, different from the usual coercion functions:

```
(Λ(α) λ(x:α) x :> ∀i(α≥⊥) α→α)
```

Paolo's solution allowed some form of conversion between the two polymorphisms... explicitly. For example, given the explicitly polymorphic type (∀e(α)α→α), it is possible to write two conversion functions:

```
λ(x : ∀e(α)α→α) some β in x[β]        : (∀e(α)α→α) → (∀i(α)α→α)
λ(x : ∀i(α)α→α) Λ(β) (x : β→β)        : (∀i(α)α→α) → (∀e(α)α→α)
```

The first annotation on `x` can be moved inside the term: λ(x) Λ(β) ((x : ∀i(α)α→α) : β→β). Note than in both cases, those annotations on `x`, indicating the *input type* of the conversion, are mandatory:

- In the (explicit → implicit) direction, `x` needs to be annotated as ∀e(α)... so that a (∀) node is generated for `x`. If it was a simple (⊥) node, the explicit instance resolution, which expects a (∀), would fail with a typing error.

- In the (implicit → explicit) direction, `x` is later given an explicitly polymorphic type. This means that `x` is *used polymorphically* and needs an annotation. This is easy to see when considering the graph inference. Usual polymorphic functions usually need to be applied with different input types; they are annotated so that their binding stay in their scope instead of going up in the environment; they can be later expanded at different place, with incompatible instantiations on their input type. The present polymorphic function needs the binding to stay in the local scope so that it can be unified[1] with bindings coming from the Λ(β)... part of the term. If they were in the global environment, we would also need to raise the (β) binding for unification, which is forbidden by the strict permissions of the explicit binding.

Those conversions, however, are manual and rather tedious, and will only work for explicit polymorphism present at the root of the term. Didier wished a stronger form of coercions that could change explicitness deep inside types — but in instantiable locations. With Paolo's explicit polymorphism representation, such a coercion would have to mutate the graph

[1] When unifying (x : ∀i(α)α→α) with the type β→β, we unify the implicit α with the explicit β. It is correct because the implicit binding on α is first weakened into an explicit binding: this is the use of the implicit→explicit weakening permission.

structure in depth, and we felt it was the wrong way to go: explicitness coercions should only care about the graph *bindings*, which by nature are much more dynamic during the inference process.

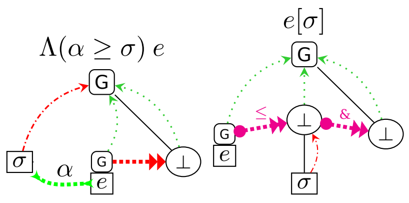
I was thus asked at the beginning of my internship — or rather, after a month of implementation and extensive debugging of the gMLF→xMLF translation — to explore the rest of the design space, in order to find a presentation of explicit polymorphism that would only change the bindings of graphs, not their structure. This mission came with a Pythian warning: Paolo, said Didier, had already considered such a possibility, but stopped exploring it for a reason that had been forgotten since.

2.0.4 Different attempts at a bindings-only representation of explicit polymorphism

My first attempt at the problem was to question the use of the new explicit binding arrow: both the explicit binding and the rigid binding are used to keep polymorphism local and block the inference process. Is it possible to factor out this behavior in only one type of binding? We quickly realized that this could not work: in the implicitly typed parts, we need the distinction between ≥ and =. If we ask quantifier introductions to rigidify ≥ into =, what happens at elimination sites? They are requested to unconditional strengthen = into ≥, which is not semantically grounded. Another problem with this idea is that explicit and rigid bindings do not exactly have the same permissions: rigid bindings can be raised but not merged, and explicit bindings can be merged but not raised.

From that first failed experiment, I kept the idea that *explicit bindings* were of a different nature than *rigid* and *flexible* bindings. The second serious attempt gave a richer structure to binding arrows: independently of being a rigid or flexible, an arrow could be *implicit* or *explicit*. An explicitly quantified term could therefore be rigid or flexible: we use the notation Λ(α≥σ)e and Λ(α=σ)e instead of the simple Λ(α)e.

By considering explicit arrows as an compromise between pure implicit arrows and structural edges, we could remove the need for (∀) node. See the following figure for the proposed translation of quantifiers introduction and elimination.



Proposed binding-only translations rules for `Λ(α)e` and `e[σ]`.

This proposal had the notable feature that elimination `e[σ]` was split in two parts:

- Bound instantiation, which we could note `e[≤σ]`, was inspired by xMLF type computations. The idea was that the gMLF typing engine would infer the instantiation relation between the current bound and the new imposed bound, and change the bound without removing the explicit polymorphism. For example, `(Λ(α≥⊥→⊥)e)[≤(⊥→int)]` would be equivalent to `Λ(α≥⊥→int)e`.

- Bound elimination, which we could note `e[&]` in reference to xMLF, would purely remove the explicit polymorphism by turning the explicit binding back into an implicit arrow. The flexibility information for the new flexible arrow is the same as the one carried by the explicit bound.

There was a first difficulty with this idea: as already mentioned, there is a mismatch between the handling of binding arrows in a graph, and the structure of binders in syntactic terms. With this representation, ∀e(α)∀e(β)... and ∀e(β)∀e(α) would be equivalent: two arrows pointing on the same node are unordered. The proposed translation for elimination is meaningless as well, unless we have a mean to specify which one, of the arrows bound on the eliminated node, is concerned.

This issue is not specific to the described proposal, but applies to any binding-based solution. There are basically two possibilities:

1. Bring the syntactic closer to the graphic, by breaking the usual conventions on quantifiers ordering. For example, an explicit polymorphism construct based on *named arguments* may fit much better its graphical counterpart.

2. Bring the graphic part closer to the syntactic, by imposing an ordering on explicit binders. As there are dependencies between explicit and implicit bindings (∀e(α≥⊥)∀i(β≥α→α)∀e(γ≥β)...), special care is required when defining the order and manipulating bindings.

It is important to notice that Paolo's (∀) structure node didn't have that mismatch with the syntactic binders: structure edges are naturally ordered by the father/child relation, and naive translation of a nested sequence of implicit and explicit quantifiers would work just as expected.

2.0.5 A note on coercions

For every binding-based solutions we considered, we had a rather clear idea of what explicitness coercions (e :> σ) (from implicit to explicit or inversely) would look like. It was actually a simple extension of the existing MLF coercion (e : σ), which generate an arrow node with two copies of the coerced types as children, with the left child rigidly bound.

We would reuse that simple schema, changing the explicitness status of the bindings inside each type copy, according to the desired coercion semantic. For example, asking for a coercion from ∀e(α)(α) to ∀i(α)(α) would generate a (→) node with two (⊥) children, the left one explicitly (and rigidly) bound, and the right one implicitly bound.

There was, however, a subtlety we initially missed. When encountering the term (e :> ∀e(α)∀e(β)..), we know that the coercion must produce two explicit bounds — on the right side of the coercion (→) — but we don't know the explicitness of the incoming type bound — on the left side. Should we generate a coercion from ∀i(α)∀e(β).., ∀e(α)∀i(β)..?

We had the idea of using a third explicitness status, *unknown explicitness*, that could be silently weakened into both explicit and implicit bounds. In an explicitness coercion, the left copy of the type would use bindings of unknown explicitness, to accommodate both explicitly and implicitly typed input.

We realized quickly that this idea wouldn't work in MLF: what should the type of such a coercion function be? If we could instantiate it to different input types (∀i(α)∀e(β)... and ∀e(α)∀i(β)... for example), what would the most general type be? In the MLF type language extended with explicit type, there is no way to express that a type can be either explicit or implicit. Given this expressiveness limit, such coercions would break principality.

We could have considered richer types, for example using *explicitness variables* ε that would range over $\{$i,e$\}$: the input type of the coercion would be something like:

```
∀(ε₁)∀(ε₂) ∀ε₁(α)∀ε₂(β)...
```

We felt, however, that this was going too far away from the main internship topic, higher-order types. Without extending MLF type system, it is necessary to be explicit about the *input type* of explicitness coercions, as well as the *output* type. The syntactic construct needed would therefore be (e : σ₁ :> σ₂), instead of the simple (e :> σ₂). The existing MLF coercions do not need two types because they do not change from one type to another, they only impose the presence of a given type.

With that heavier coercion construct, the justification for binding-only explicit polymorphism decreased: once the coercion is parametrized by both types, it is easy to build a generic coercion function that goes from one type to the other, even if they don't have the same graph structure. We could, for example, introduce or remove (∀) nodes between the input and output graph type, even in depth.

The only important aspect of this generalized coercion framework is the choice of the allowed coercions: given two types σ_1 and σ_2, does a coercion function going from σ_1 to σ_2 make sense? This is a reflexive relation between MLF types which, when specialized to the identity, provides exactly the existing MLF coercions. The specifics of this coercion admissibility relation are actually independent of the MLF inference machinery. This is a parameter to the MLF type system, and different MLF-based languages could make different choices.

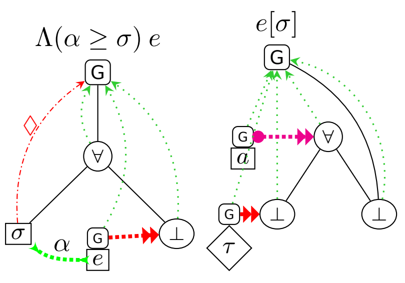
With this realization, we stopped to focus on coercions. This was a topic of larger scope than our MLF setting, and a different research subject — on which work is now ongoing inside Gallium.

2.0.6 Implementing the generalized form of Paolo's solution

We encountered important difficulties with the multiple bindings-only proposals for explicit polymorphism. This made for a quite frustrating part of the internship, where the proposed solutions fell down one after another for seemingly innocuous reasons. During more than a month, we experienced a long series of failures.

We were well aware that Paolo's solution was significantly simpler to implement, due to the binding ordering issues of non-structural solutions. When our idea of light explicitness coercions fell down, there was little motivation left for a bindings-only solution. We therefore decided to give up and reuse Paolo's structural solution.

We had however gained additional insight during the search process, that allowed me to generalize and incrementally improve on Paolo's solution: explicitness and flexibility of binders are orthogonal. We therefore kept the $\Lambda(\alpha\diamond\sigma)$.. syntax instead of Paolo's only $\Lambda(\alpha)$... The explicit instantiation constraint now change the binding from explicit to implicit, but without mutating the $\geq$,= status of the binder. This improvement is non-invasive, as only the introduction rule changed:



Final translations rules for `Λ(α◇σ)e` and `e[σ]`.

I implemented this generalized solution, inspired by Paolo's first implementation. This was a bit of more work as the prototype had been significantly expanded, in particular with xMLF elaboration.

The translation of explicit polymorphism to xMLF was not obvious to get right, but was finally very simple. The idea is that, after the type inference process, the ordering of quantifiers (both implicit and explicit) are only determined from the binder structure. The translation process already computes an ordering of binders, suitable for translation to syntactic quantifiers: no extra work was needed for explicit bindings. The structural (∀) node is only used to check whether an explicit bound really refers to a quantifier introduction — it could result from a simple binder weakening. In a sense, the xMLF translation layer act as if we had a binder-only solution.

A note on explicit bounds Changing the explicit bounds from a single $\Lambda(\alpha)$ to $\Lambda(\alpha\diamond\sigma)$ is more than an aesthetic choice for homogeneity in the type language.

Flexible explicit bounds have an interesting property: they allow partial type specification for the abstracted type. For example, I may abstract over a type, while keeping apparent that it is a function type that can be used as such.

```
Λ(α ≥ ∀(β)∀(γ)β→γ) λ(f:α) λ(x)λ(y) (f x, f y)
```

Such partial specification was not possible in System F, where abstracted variable must be used in a parametric, agnostic way. Of course, it is possible to write instead

```
Λ(β)Λ(γ) λ(f:β→γ) λ(x)λ(y) (f x, f y)
```

but this approach may require an arbitrary number of type abstractions were the more flexible bounded abstraction use only one.

Bounded explicit quantification gives a nice regularity property to MLF type variable declarations: everywhere a `some α in ...` expression is used, it may be replaced, if we know the type σ inferred for α, by the abstracted program `Λ(α≥σ) ...`. This wouldn't hold with the less flexible $\Lambda(\alpha)$... bound.

The case for explicit rigid bounds is, however, less clear: what is the use of abstracting over a type variable if the elimination type is known in advance and cannot be instantiated even, after abstraction elimination? Rigid explicit bounds act as local type definitions. More practical experience with MLF-based languages may discover interesting use for this construction, or confine them as merely anecdotal combination of orthogonal features.

There is also an issue with explicit rigid bound: they cannot be directly translated in xMLF. $\Lambda(\alpha\geq\sigma)$... bounds are naturally translated into the xMLF $\Lambda(\alpha\geq\sigma)$... construct. But xMLF does not have rigid bounds: as they do not help typability in an explicitly typed language, they are not necessary and can be inlined directly.

My proposed fix was to target a richer xMLF language, extended with rigid quantifications. This language could anyway be translated back into the simple $\geq$-only xMLF. An advantage of this extended language is that it gives better complexity properties to the gMLF→xMLF elaboration: presently, as rigid types must be inlined during translation, an xMLF term may be exponentially larger than its MLF counterpart. When targetting a language with rigid bindings we can estimate the translation cost more precisely.

On the practical side, this proposed extension of xMLF was not implemented during the internship, due to lack of time. Therefore, terms with rigid explicit bound cannot be translated into xMLF for now; this is one of the things that I will hopefully fix later.

3 Going to order ω

With the explicit polymorphism in place, we were ready to develop the main motivation of my internship, MLFω. Due to the considerable (and unexpected) difficulties encountered during the previous step, I didn't have much time left. The work presented happened during the last month of my internship, and after the internship ended. There are rough edges and remaining issues, and the implementation (in the developped prototype) is marked experimental and disabled by default.

In this chapter, I will give an informal overview of System Fω, then present the corresponding gMLF extensions, with a detailed description of the β-reduction process, and finally present the remaining issues and possible future developments.

3.1 System Fω

Simply typed λ-calculus is built on the λ abstraction construct, which is a way to express terms "parametrized" by other terms. System F polymorphism is a form of parametrization of terms by types: polymorphic terms take a type variable as parameter. System Fω adds a parametrization of types by types, allowing to abstract out parts of a type expression. [Gir72] [Pie02]

Here is the type language of System Fω:

```
σ ::=              System Fω types
    | α,β,γ        type variables
```

```
      | σ₁ → σ₂        function type
      | ∀(α:κ) σ       type quantification
      | λ(α:κ) σ       type abstraction
      | σ₁ σ₂          type application
```

The parametrization of types by types is represented by the λ(α:κ)σ abstraction construct, associated to a corresponding (σ_1 σ_2) type application. In essence, it is an embedding of the simply typed λ-calculus (STλC) into the type language of System F.

Exactly as in the STλC, variables are typed. What is the type of a type variable ? It's a *kind*, denoted by the metavariable κ. Here is the simple kind language of Fω:

```
 κ ::=                 System Fω kinds
      | ⋆              star kind
      | κ₁→κ₂          arrow kind
```

The ⋆ kind is a constant kind that classifies the types of the System F terms (unit, int, function type...). The arrow kind $\kappa_2 \to \kappa_2$ classifies the type operators that are parametrized by a type variable of kind κ_1, and return a type of kind κ_2. For example, λ(α:⋆) α→α is a type operator of kind ⋆→⋆ returning, when given a ⋆-type, the type of the monomorphic identity on that type. As kind ⋆ is often used, we conveniently write λ(α)σ instead of λ(α:⋆)σ.

It is important to understand the difference between type quantification ∀(α:κ)σ and type abstraction λ(α:κ)σ. Quantification is fundamentally a first-order construct: it is used to classify *terms* that depend on a type. On the contrary, λ(α)σ is a higher-order construct, specific to the type level: there is no term of type λ(α)σ. Those two extensions of the STλC are actually orthogonal: it is possible to consider a λ-calculus with higher-order types, but no term-level polymorphism: Fω, without the *F* part. This simpler calculus is called λω, or λω, in the type system litterature [Bar91].

Note that System F type quantification was extended to higher-order kinds: ∀(α:κ)σ. This allows a term to depend not only on ⋆-types, but on higher-order type operators. This is matched in the term syntax: the type introduction construct accepts higher-order kinds Λ(α:κ)e.

This is probably the most useful part of Fω for the working programmer. In the following example, we express the `twice` function, which composes a function with itself:

```
twice := Λ(F : ⋆→⋆) λ(f : ∀(α) α → F α) λ(x:α) f[F α] (f[α] x)
```

Given a (List:⋆→⋆) type with the usual `nil` and `cons`, we can write the following (∀(α) α → List (List α)) function:

```
twice List (Λ(α) λ(x:α) cons[α] x nil[α])
```

In ML or System F, the only possible type for `twice` is ∀(α)α→α, which is vastly less expressive.

System Fω is used as a core type system to express various features of modern programming languages, such as Haskell, Scala, and ML higher-order functors [RRD10].

3.1.1 Typing and β-reduction

In previous explicitely-typed systems such as System F, we only used equality between types up to substitution of type variables by their value in the environment. With λ-terms inside types, we want a larger relation. We want to accept, for example, `(λ(α)α→α) int` as a correct type for `λ(x) x+1`.

Formal presentations of Fω use an additional judgment of *equivalence* between types, the important part being:

```
(λ(α)σ₁)σ₂ ≡ σ₁[α←σ₂]
```

The algorithmic way to test equivalence between two types is to reduce all their redexes (reductible subterms of the form $(\lambda(\alpha)\sigma_1)\sigma_2$) until they reach a *normal form*, where no additionnal reduction is possible, and to test equality on the two normal forms.

3.2 From MLF to MLFω

Type inference in Fω is undecidable. Partial type inference have given (partially) satisfying results, but MLF design goal is rather to concentrate on complete and principal type inference for first-order types. In the long term, it may be interesting to try to go further and infer some higher-order types, but this is not in the scope of the present work.

We therefore decided to make all higher-order aspects of the language fully explicit. While explicit polymorphism at kind ⋆ — that is provided by the MLF Λ(α◇σ) construct — was mostly a convenience, explicit polymorphism is absolutely necessary for MLFω, and this was the main justification for adding explicit polymorphism to MLF: Fω expressivity is highly desirable.

MLFω, the extension of the MLF with Fω feature, is very similar to the extension from F to Fω.

```
σ ::=                      MLFω types
    | α, β, γ              type variables
    | ⊥                     bottom
    | α → β                function types (using only variables)
    | ∀(α◇σ₁) σ₂            bounded quantification (◇ ∈ {≥, =})
    | λ(α:κ) σ             type abstraction
    | σ₁ @ σ₂              type application
```

Note : We used an explicit infix operator `@` for type application, to make more apparent the presence of a structural node in the gMLF translation.

There is a small specificity in the term language: the two abstractions Λ(α◇σ) and Λ(α:κ) coexist separately.

```
e ::=                        MLFω expressions
    | x, y, z                variables
    | λ(x) e                 λ-abstraction
    | e₁ e₂                  application
    | let x = e₁ in e₂       local definition
    | Λ(α◇σ) e                first-order introduction
    | Λ(α:κ) e               higher-order introduction
    | e[σ]                   elimination
```

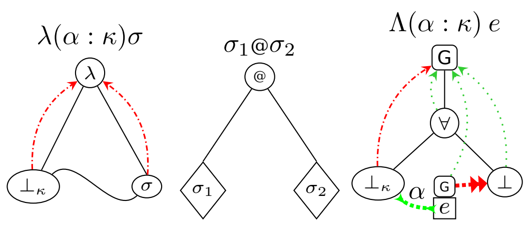
The reason why we don't use a combined construct Λ(α◇σ:κ) is that the first-order bound (α◇σ) rely on the MLF instance relation which is based on implicit polymorphism. We are very careful *not* to add any kind of implicit inference to the higher-order part of gMLFω.

The implicit bound Λ(α) may refer similarly to Λ(α≥⊥) and Λ(α:⋆) which, as we will see, have the same meaning in gMLFω.

3.2.1 From gMLF to gMLFω

For explicit polymorphism, we eventually added new structure nodes, (∀) nodes. To add higher-order types, we introduce two new nodes of arity 2, (@) application nodes and (λ) abstraction nodes.

The idea is that (σ_1 @ σ_2) will be translated to @((σ_1),(σ_2)), while λ(α:κ)σ is represented by λ(⊥,(σ)), where α is translated in (σ) by the left-child (⊥) node.



Note : we can now verify that Λ(α≥⊥) and Λ(α:⋆) are equivalent : they generate the same graph explicitly flexibly bound on Λ-var (⊥:⋆).

We do not want the inference process to try to infer the λ-variable or its body: both structural child of a (λ) node are explicitely bound to this node.

Finally, we also added a kind information on each structure node of the graph. This allows to check for well-kindedness. All the existing gMLF structural nodes have kind $\star$, except for $\perp$ nodes that may have any kind κ (so that they can represent higher-order type variables), and the two new structure nodes have kind:

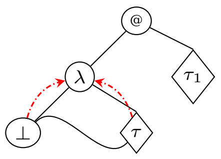
- $\kappa_1 \rightarrow \kappa_2$ for a (λ) node of left child with kind κ_1 and right kind κ_2
- κ_2 for a (@) node of left kind $\kappa_1 \rightarrow \kappa_2$ and right kind κ_1

Most kind checking is done during the translation from syntactic terms to graphic types. For example, when translating the $(\sigma_1 \rightarrow \sigma_2)$ type, it is easy to check that (σ_1) and (σ_2) graphic translations have kind $\star$.

There is one case, however, where kind checking must be delayed to the constraint solving process: type elimination `e[σ]`. We can determine the kind of σ, but the arity of the `e` node is unknown until its type is fully resolved. We will therefore add a kind checking step to the unification process.

3.3 Graphic β-reduction

The main component needed to extend the gMLF type system to gMLFω is a β-reduction operation. The β-reduction should operate on a β-redex, which is a (@) node whose left child is a (λ) node. It should return a new graph constraint corresponding to the reduced type. Approximatively :



- The λ-variable — left child of the (λ) node – should be replaced by the @-argument — right child of the (@) node.
- The (@) node itself should be replaced by the λ-body – the right child of the (λ) node.

How will we rigourously specify the graph β-reduction algorithm ? Paolo Herms had already worked on the subject. He had noticed an important distinction, that is one of the difficulties of the β-reduction process : we should only reduce if the (λ)-node is *local* to the (@)-node, and otherwise make a local copy of it before reducing.

The justification for this consideration is clear : if the (λ)-node is global in the environment, it may be shared with other nodes. If we mutate the (λ)-node during the reduction process, such changes may incorrectly affect other uses of the node.

Taking a local copy is possible due to the very useful — and lucky — property that (λ)-nodes are always *inert* : as both their children are explicitely bound, they carry no direct polymorphism (nothing is implicitly instantiable). It is therefore sound to *unmerge* them, by copying them, and then lowering the copy's bound by (flexibly) fixing it on the (@) node. Lowering the binder cannot create an ill-formed graph, as the (@) node is (λ) direct structural ancestor.

3.3.1 Implementation considerations : reusing expansion and unification ?

I tried to write a first prototype of the β-reduction process, using the low-level ”replace” operation provided by the graph interface (the modification happened at the level of the union-find tree used to merge nodes during unification). The result was very fragile and raised lots of bugs, as the graph butchery broke various graph invariants that must be preserved during the graph constraint resolution process.

For my second attempt at implementation, I decided to reuse existing code that was tested and efficient at preserving graph invariants. My idea was to use the existing ”expansion” algorithm to do the subgraph copying (unmerging the (λ)-node), and the ”unify” algorithm to do the replace/substitutions. The only unsafe operation needed was to atomically replace the (@) node by a $(\perp)$ node before unifying it with the λ-body.

It turned out that the result were ever worse than at the first try. There were three main defects, in increasing order of importance :

- The (β)-reduction process used the graph expansion routine, while the expansion routine used the unification routine (on the frontier nodes), which itself used the (β)-reduction algorithm. I had created a cylic dependency, which are always relatively painful on a software engineering level (recursive modules, welcome).
- The unification routine was proved correct on the usual $\star$-kinded nodes of gMLF, but I had no idea what might happen if a frontier unification constraint happened on a higher-kinded node, which may appear under a (λ) node.
- The idea of the gMLF binding permissions, which are carefully checked during expansion and unification, is that instantiation happen from the *root* of the graph constraint. An operation on a node is based on its binding path to the root. In contrast, he (β)-reduction needed to operate at a local level. For example, the (β)-reduction implementation completely failed if the (@) node was rigidly bound : all nodes under it were considered locked, and expansion/unification impossible.

Fundamentally, the intermediate steps (β)-reduction must not, contrarily to unification steps, be considered as instantiation operations. (β)-reduction needs to be a local process that changes the graph atomically, and is not concerned with the permissions of the outer binding context. Clearly, the unification and expansion algorithm were unsuited for such a task, and it would be dangerous to try to extend them in this direction.

I therefore had to develop a new description of the (β)-reduction algorithm, using only atomic, unsafe (no permission checking) operations. Great care must be taken and the different changes must be meticulously considered to answer two strongly correlated questions: Do they preserve the graph invariants ? Are they semantically sound ?

In the following subsections, I will precisely describe the four different steps of the (β)-reduction process.

3.3.2 Step 1. Taking a local copy of the (λ) node

If the (λ)-node of the redex is local to the (@)-node, nothing need to be done. If it is bound higher in the graph, it is necessary, as Paolo noted, to take a local copy of it, and (flexibly) bind that local copy to the (@) node.

To copy the (λ)-node, we copy the subgraph corresponding to its interior : all its structural children that are bound to it or in the interior. The only difference with expansion is that frontier node, the first structural descendants bound higher in the graph, are directly shared, instead of being copied as fresh $(\perp)$ nodes and unified after expansion — which is not possible as it would reuse unification.

Expansion moves subgraph ”horizontally” in the graph constraint. It uses frontier unification to raise frontier binders to a common ancestor of both the source and destination of the subgraph. In the present case, we move the graph ”vertically”: the (λ)-node was bound on a structural ancestor of the (@) node, and we lower its copy's bound to the (@) node. The frontier nodes were bound on structural ancestors of the (λ) node, and still are: the resulting graph is well-formed.

3.3.3 Step 2. Unfreezing both (λ) children nodes

We need to unfreeze both (λ) children nodes as a semantic justification to further modification of those nodes. Unfreezing is sound as the (λ) node is now local to the redex : this cannot affect outer nodes.

It is also important that the (λ) node, after unfreezing its children, is never made accessible to the global graph constraint again : the gMLFω type system relies on the invariant that (λ) nodes are always inert. The unfrozen (λ) node must not leak to the outer graph. We will have to verify this condition at the end of the β-reduction process.

The (λ) node was introduced by a graphic translation of a λ type. Both its children — or its unique children, in the case of λ(α)α — are flexible.

There is an edge case were the λ-body, due to an aliasing problem, is actually bound higher in the graph (λ(α)β). We cannot change its binder, but this is not an issue : the final step concerned with the λ-body will consider this possibility; see the ”Pathological case” subsection at the end of this section.

3.3.4 Step 3. Replace the λ-var with the @-argument

In order to soundly replace the λ-var, which is a (⊥) node, with the @-argument, we must first have both bindings bound on the same node, with the same binding flags.

The λ-var is flexibly bound to the (λ) node. The @-argument is bound higher in the tree – at least on the (@) node — with arbitrary flags. We can soundly raise the λ-var binding to the @-arg’s bound, and weaken its flags — both flexibility and expliciteness — to match the @-arg flags.

Once the two bindings are identical, we can merge the λ-var (⊥) node with the @-arg without breaking well-formedness of the graph.

3.3.5 Step 4. Replace the (@) node with the λ-body

This is the last, and final step, of the graph β-reduction process. As in the previous step, we must ensure bindings coincide (both in destination and in flags) before merging the two nodes.

In most cases — see next section for the unusual case — the λ-body is flexibly bound to the (λ) node. It happens exactly as in the previous step : we raise this flexible binding and weaken it, before substituting (@) with the λ-body. There is a special case where (@) is the root of the type graph, instead of being bound to an ancestor. In this case, we make the λ-body a root too before merging.

After the substitution, all structural information about the (@) node is lost, replaced in the graph by the λ-body (in which the variable substitution has been done). In particular, the (λ) node cannot be accessed anymore, preserving the invariant that all accessible (λ) nodes are inert.

3.3.6 The pathological case

There is however a pathological edge case in the last step : the λ-body is *not* necessarily flexibly bound to the (λ) node. It was true when the (λ) node was unfrozen, but something has changed since then : the λ-var has been substituted with the @-arg. There is a case where the λ-body and the λ-var share the same (⊥) node : the identity λ(α)α.

In the identity case, the λ-body binding is now the @-arg binding. It may, or may not, be compatible with the (@) node binding. Consider the following example :

```
∀e(β≥⊥) ∀i(γ = (λ(α)α) @ β) γ→γ
```

In this example, the (@) node is rigidly bound by the γ type variable, but the @-arg is flexibly implicitly bound.

This is actually an instance of the bound aliasing problem. We make a lenient choice : in case of conflict between two binders, do not fail, but instead give the priority to the binder bound higher. Potential disadvantages of this choice will be discussed in the ”bound aliasing problem” section at the end of this chapter.

3.4 Unification and reduction

We defined a graph β-reduction operation that reduces a graph redex. How is that β-reduction operation integrated in the existing gMLF inference algorithm? When should β-reduction happen?

The gMLF unification algorithm is split in two parts. First, unify_structure change the graph *structure*, doing first-order unification. Then, rebind makes a second pass on the modified subgraph, compares the differences between the old and the new structure, adjusts binders, and checks that the changes are compatible with the binding permissions.

In a ML-inspired pseudocode, here is the algorithm for the first pas, first-order unification of gMLF graphs :

```
unify_structure($g_1$, $g_2$) :=
  match the constructors of $g_1$ and $g_2$ with
  | ⊥, $c_2$ -> graft $g_2$ on $g_1$
  | $c_1$, ⊥ -> graft $g_1$ on $g_2$
  | $c_1$, $c_2$ when $c_1 \neq c_2$ -> fail
  | $c_1$, $c_2$ when $c_1 = c_2$ ->
    for each $n_1$ child of $g_1$ and $n_2$ the corresponding child of $g_2$:
      unify_structure($n_1$, $n_2$)
```

It is important that reduction happens before this unification. Indeed, the constructor comparison is not correct on higher-order constructs, in particular (@) nodes. Consider for example the following unification problem:

```
((λ(α) α→int) @ bool) ~ ((λ(α) bool→α) @ int)
```

The given algorithm would check the two @ constructors for equality, then try to recursively unify (λ(α)α→int) (λ(α)bool→α) and bool int. This is clearly wrong : we have to reduce before unification.

There are several reduction strategies. For example, we could choose to reduce as soon as a redex appear in our graph. We choosed instead a lazy[2] strategy, where redexes are only β-reduced when it is necessary to do so : just before unification.

3.4.1 Weak head normal form

In practice, it may not be the case that all (@)-nodes encountered during unification are redexes. The left child is not necessarily a (λ) node. As it has a higher-order kind, it cannot be a (→) node or a constant (int) node, but it could be (@) or (⊥). In the first case, what we need is to recursively try to reduce the inner (@) node, hoping reduction will yield a (λ)-node, forming a complete β-redex.

To define that ”recursive reduction” process, we define a *weak head reduction* algorithm, that reduces a given node until the head constructor cannot be reduced further. For notation convenience, we will express the algorithm in term of the MLF *syntactic* types, but the reader must keep in mind that it is actually a transformation on a gMLF graph.

```
whnf((λ(α)$σ_1$) @ $σ_2$)  := whnf($σ_1$[α←$σ_2$])
whnf(⊥ @ σ)          := (⊥ @ σ)
whnf($σ_1$ @ $σ_2$)        := whnf(whnf($σ_1$) @ $σ_2$)   when $σ_1 \notin \{⊥, (λ..)\}$
whnf(σ)              := σ                      when $σ \notin \{..@..\}$
```

3.4.2 Correction of the unification algorithm

The new algorithm for unification is exactly the previous one, where we just reduced each graph before comparing first-order structure. Note that it will recursively reduce children too.

[2] In reduction of λ-terms, there is an important distinction between ”lazy” and ”call by need” evaluation strategies. As we are in a graph, such distinction is not very meaningful : nodes are naturally shared and β-reduction does not duplicate its argument.

```
unify_structure($g_1$, $g_2$) :=
  check that $g_1$ and $g_2$ have the same kind
  $g_1$ ← whnf($g_1$)
  $g_2$ ← whnf($g_2$)
  match the constructors of $g_1$ and $g_2$ with
  | $\perp$, $c_2$ -> graft $g_2$ on $g_1$
  | $c_1$, $\perp$ -> graft $g_1$ on $g_2$
  | $c_1$, $c_2$ when $c_1 \neq c_2$ -> fail
  | $c_1$, $c_2$ when $c_1 = c_2$ ->
    for each $n_1$ child of $g_1$ and $n_2$ the corresponding child of $g_2$:
      unify_structure($n_1$, $n_2$)
```

There is an important subtlety : after whnf reduction, the recursive traversal is applied on all nodes, *even* higher-kinded nodes. As we have seen before, recursively unifying (@) children is obviously wrong in the general case. We will use unify_structure nonetheless. In the following sections, I will show that it is correct after whnf reduction.

Claim : The existing unification algorithm, as a combination of unify_structure and rebind, will *not* try to do any inference on higher-kinded nodes. It will only succeed on unification problems where the given nodes are either equal or — for (λ)-bindings — α-equivalent. The rebind pass will fail on all other — unsound — transformations.

There are three cases to consider, corresponding to the tree possible higher-order constructors : higher-order (⊥) nodes, (@) nodes and (λ) nodes.

3.4.3 The higher-order (⊥) case

In the (⊥) case, my claim is that rebind will reject all modifications on a higher-order (⊥) node : the only unification problems that won't fail are those where the two arguments are exactly the same node in the graph. To prove this, I will first prove that all higher-order (⊥) nodes in a gMLFω graph constraint are explicitely bound to a (∀) or (λ) node.

Where does a higher-kinded (⊥) node, of kind strictly bigger than $\star$, come from ? For a graph generated by the gMLFω translation process, the only possibility is: as a variable of some explicit abstraction – Λ(α) or ∀e(α) for (∀) nodes, λ(α) for (λ) nodes. In particular, the type constant ⊥ in a MLF type expression is always $\star$-kinded. And the higher-kinded variables of those abstractions are never unfrozen — made implicit:

- To unfreeze a (∀)-var, one need to eliminate it with a same-kinded type σ : e[σ]. But as the constant ⊥ is not higher-kinded, the only possible well-kinded types σ are :
  - higher-kinded variable explicitely bound higher in the environment : the result of elimination is still an explicitely bound (⊥)
  - higher-kinder constructs with other whnf than (⊥) — (@) or (λ). The result is no more a (⊥) node.
- Similarly, a (λ)-variable is replaced by a type application argument, which may only be an other explicitely bound variable or a non-(⊥) constructor.

Therefore, all higher-kinded (⊥) nodes are *explicitely* bound.

The permissions on explicit bounds tell us that such node cannot be grafted, but it may be merged with an isomorphic cobound node. Can two *different* explicit (⊥) nodes be cobound ? No: as explicit binding arrows cannot be raised, a higher-order (⊥) is always bound to its original (∀) or (λ) node. (∀) only have one explicit bound, so different cobound nodes are impossible. (λ) nodes have two explicit bound children, but the λ-body cannot be a different (⊥) node cobound to the λ-var:

- If the body is a type variable bound higher (λ(α) β), it is not bound to the current (λ) node.
- If we are in the identity case (λ(α)α), the λ-var and λ-body are the same (shared) node

In other words, the presence of a structural node ((∀) or (λ)) for each higher-order (⊥) introduction ensure uniqueness : if two (⊥) are explicitely bound on such a structural node, they must be equal. Therefore, allowing the merge of cobound (⊥) nodes is equivalent to the correct identity test : if the two arguments are equal, unification succeeds, otherwise it fails.

3.4.4 The (λ) case

Higher-kinded (λ) nodes cannot be grafted on a same-kinded (⊥) node : as we have shown, such (⊥) node are explicitely bound, therefore any grafting will be rejected by rebind.

As we have seen, when called recursively on the λ-var, which is an explicit (⊥) node, unification will only succeed if the two λ-vars are equal. This is correct. unify_structure, however, could recursively unify the λ-body of two (λ) nodes. This is only correct if the two (λ) nodes are α-equivalent.

My claim is that it is the only transformation that rebind will allow. Indeed, λ-body being explicitely bound to the (λ) node, all flexible (⊥) nodes in its body are red/locked. No instance operation on locked nodes is allowed by rebind. Therefore, the only possible unification operation is the merging of the two isomorphic inert (λ) nodes; but two (λ) nodes have isomorphic interior if and only if they are α-equivalent !

3.4.5 The (@) case

In the (@) case, we make use of the fact that the (@) node is in weak head normal form, which strongly restricts the left node: it is a (⊥), or a (@) node itself in whnf.

- if the left child is a (⊥), we have already shown that it may only unify with itself. In that case, any unification is correct on the right child : when F is a higher-kinded variable, it is always sound to resolve (F σ) (F σ) by σ σ
- if the left child is a (@)-whnf itself, a simple structural induction — which we can use because gMLFω graphs are acyclic — show us that the second-order parts of the two left child are not changed by unify_structure, only compared by equality and α-equivalence. Deriving σ_1 σ_2 from (φ_1 σ_1) (φ_2 σ_2) is sound when φ_1 and φ_2 are α-equivalent.

3.4.6 Conclusion

What we have shown in this possibly boring proof is that we do not have to specify a separate unification algorithm for higher-kinded nodes. It may seem rather surprising that the first-order unification algorithm, using the existing permission handling, is actually fully correct on the new higher-kinded terms.

A posteriori, this is actually not so surprising : the correctness proofs for the three cases were all based on the strict permissions of the explicit binding. This brings a new point of view on the explicit binding : it has exactly the permissions that force the unification algorithm to *not infer anything* in higher-order kinds. In other words, the explicit binding is exactly the good binding mode for higher-order types.

I believe this provides a good backwards validation of the explicit binding design. It also provides an additional explanation of why only the left child of (∀) is explicit, while both children of (λ) are : only the left child of a (∀) node may be of kind higher than $\star$, while the two children of (λ) generally are.

It is also a good sign for the extensibility of the higher-order features of gMLFω. What we first thought was that we would need to carefully study each new feature to design the right "unification" (that is actually not unification on higher-order types) algorithm for it. This experience with (∀) and (λ) nodes suggests that, with the right amount of explicit bindings, any new non-first-order feature should ”just work” with respect to the existing unification algorithm. Of course, correctness proofs are still needed.

3.5 Other considerations

3.5.1 η-reduction

We may also add a η-reduction rule :

```
whnf(λ(α) σ@α)       := σ                     when α is free in σ
```

This rules is semantically sound and widen the equivalence relation between higher-order nodes. It is commonly found in presentations of the Fω type system, and provides some additional flexibility.

I have concentrated my attention on the β-reduction, but I do believe η-reduction can easily be added : just change the `whnf` algorithm. However, extensions of gMLF have a repeated history of proving more delicate than expected.

3.5.2 The bound aliasing problem

The attitude towards bound aliasing presented in this report is rather lenient : in case of aliasing, just give the priority to the higher bound. This is a very recent (and quite daring) change to our perception of gMLF : our first impulse was to forbid it entirely. Indeed, it seems very strange that the explicit binding on β disappears in the following example :

```
Λ(α) Λ(β ≥ α) e
```

Is it really correct to simply forget the β≥α binding that the user explicitely specified ? We decided that it wasn’t, and raised a type error in that case.

However, with the type-level abstraction constructs, life got harder for bound alias detectors :

```
Λ(α) Λ(β ≥ (λ(γ) γ) α) e
```

In this case, the bound alias problem is only detected at β-reduction. With our lazy β-reduction strategy, it wouldn’t raise an error unless the bound was involved in an unification problem. And it is even worse :

```
delayed_alias := Λ(F:⋆→⋆) Λ(α) Λ(β ≥ F α) e
```

This looks like a perfectly valid MLFω expression. Give it to the toplevel and it will gladly accept it. Then, in the next phrase, try something with `delayed_alias[λ(γ)γ]`, and you get a *delayed bound alias* error.

We looked for all sorts of solutions to this issue, including simply forbidding the use of the type-level identity, or equally dangerous `(λ(α) β)` constant functions. The best solution so far is a kind system proposed by Didier Rémy, that would use two kinds `N` and `P` instead of ⋆: Non-protected (from bound alias) and Protected. The ”safe” types such as $\sigma_1 \to \sigma_2$ would be kinded `P`, and the ”dangerous” type variables would have kind `N`. In binding positions, we would only accept higher-kinded types with result kind `P`.

This solution can probably be hidden from the user by using some sort of *kind inference* : MLF syntactical expression would only use the ⋆ kind, and an analysis would infer if it really is N or P. However, that would only give a conservative approximation, with false negatives : some type-level functions would be rejected at elimination sites where they would actually be ”bound alias safe”.

The present lenient solution is much simpler, but has potentially undesirable side-effects. For example, the inferred type for `fun x -> x[int]` is, surprisingly, ∀i(α)α→α. The gMLF inference result is really ∀i(α) (∀i(β)α)→α, where the inner ∀i is the trace of an ”unlocked (∀) node”, but in gMLF unused bindings have no semantic meaning, so ∀i(β)α is equivalent to α.

The fact that the β binder comes from an unlocked explicit quantification can be recovered from the complete type : we can detect the presence of the (∀) node. This suggests that we may, in some cases, display those binders differently to the user, and in particular avoid that they get erased when they’re considered useless. We had no time to investigate further.

A promising approach to attenuate the sometime surprising effects of the lenient rule, suggested by Didier, would be to still emit warnings in bound aliasing situations. We would benefit from the two approaches at once : the — necessarily approximative for higher-order types — warnings would prevent user surprise at vanishing quantifiers, but the type system would use the simpler, predictable and complete lenient strategy.

We’re not sure the ”bound alias” story is definitely closed yet. We may discover new issues with the lenient approach, and search for a better solution. In the software prototype, Fω features have been marked ”experimental” and are disabled by default.

References

[Bar91] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972.

[Her09] Paolo Herms. Partial Type Inference with Higher-Order Types. Master thesis, University of Pisa and INRIA, 2009. Available from: http://pauillac.inria.fr/~remy/mlf/Herms@master2009.pdf.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[MT10] Giulio Manzonetto and Paolo Tranquilli. Harnessing mlf with the power of system f. In *Mathematical Foundations of Computer Science*, volume 6281 of *LNCS*, pages 525–536, August 2010. Available from: http://perso.ens-lyon.fr/paolo.tranquilli/content/docs/snmlf.pdf.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142, 2002. Available from: http://www.cis.upenn.edu/~bcpierce/tapl/.

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. Available from: http://cristal.inria.fr/attapl/.

[RRD10] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *TLDI '10: Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 89–102, New York, NY, USA, 2010. ACM. Available from: http://dx.doi.org/10.1145/1708016.1708028.

[RY08] Didier Rémy and Boris Yakobowski. A church-style intermediate language for MLF (extended version). Available at http://gallium.inria.fr/~remy/mlf/xmlf.pdf, September 2008. Available from: http://gallium.inria.fr/~remy/mlf/xmlf.pdf.

[Wel94] J. B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Ninth annual IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

[Yak08] Boris Yakobowski. *Graphical types and constraints: second-order polymorphism and inference*. PhD thesis, University of Paris 7, December 2008. Available from: http://www.yakobowski.org/phd-dissertation.html.