Structural interfaces for ML$_{\leq}$

Daniel Bonniot

April 21, 2000

Stage de DEA de l'Université Paris VII (Denis Diderot)

École Nationale Supérieure des Mines de Paris
Under the direction of François BOURDONCLE

Contents

Acknowledgments

I would like to thank François BOURDONCLE for the subject of this work, for his patience while explaining me the subtleties of $\mathrm{ML}_{\leq}$ and for the time we spent discussing the axiomatization;

Alexandre FREY, for all the explanations he gave me, for the nice piece of code I could re-use as a "black box" and for all the challenging discussions;

and also Patrice BERTIN, who is always ready to help me with reluctant software, Gilles MALFREYT for the impassioned discussions about programming languages in general, and about Bossa in particular, and Laurent MEUNIER, for the cakes and the coffee, without which nothing would have been possible.

1 Introduction

Type systems for object-oriented computer languages is yet an active research field. Those used in mainstream languages are too weak to model some properties needed by usual programs. On the other hand, many type systems proposed by the academic community have not been used in practice, because of either their in-decidability or their too high complexity that makes them very difficult to implement correctly and efficiently.

$\mathrm{ML}_{\leq}$ [BM97] is an attempt to extend the well-known Hindley/Milner ML type system [Mil78] with sub-typing and dynamic dispatch. As such an extension, it can give polymorphic types to methods. It has a strong and elegant axiomatization, with proofs of safety and decidability. Jazz [Fre99] is a strongly-typed lazy declarative single-assignment higher-order functional object-oriented language based on the $\mathrm{ML}_{\leq}$ type system. It demonstrates that $\mathrm{ML}_{\leq}$ succeeds in modeling all aspects of object-oriented languages, and can be implemented in an efficient manner.

However, practice with Jazz showed the need to enhance $\mathrm{ML}_{\leq}$, as we shall see in section 2.2. A solution based on so-called "interfaces" has been found to overcome those limitations. The subject of this training experience was:

- to modify the $\mathrm{ML}_{\leq}$ theory to incorporate interfaces;
- to prove that the main results remain valid;
- to find algorithmic solutions for the problems raised by interfaces;
- to implement a "proof of concept" language based on this extended version of $\mathrm{ML}_{\leq}$.

2 ML$_\leq$

2.1 Introduction to ML$_\leq$

ML$_\leq$ is a type system, designed to integrate the characteristics of functional and object-oriented programming. Its main idea is that it should be possible to re-use the well-known ML type-system to model the lesser-known object-oriented type system, with only a few additions when necessary.

Like ML, it has parametric polymorphism and higher order functions. It models object-orientation with a user-defined hierarchy of so-called *type constructors* instead of the set of unrelated atomic types of ML. Type constructors may have type parameters, in which case they model generic classes[1]. The number of type parameters of a type constructor is called its arity. The application of a type constructor to the correct number of *monotype*s is called a *monotype*[2]. Functional monotypes can be build by the application of the *arrow* type constructor.

For instance, here is a possible definition of lists:[3]

```
abstract class List<T> {}
class Nil<T> extends List<T> {}
class Cons<T> extends List<T> {}
```

This declares a type constructor `List` of arity 1 with two sub-classes `Nil` and `Cons`.

Polymorphic types – *polytypes* – are universally quantified monotypes. For instance, we can declare a method

```
List<T> cons < T > (T value, List<T> l);
```
[4]

which type, written in a more conventional way, is $\forall \mathtt{T}.(\mathtt{T}, \mathtt{List<T>}) \rightarrow \mathtt{List<T>}$[5].

We can see in this example that the ML data-type declarations are replaced by the declaration of one abstract class and one sub-class for each case. Besides the interest of getting rid of this concept, an advantage is that the type of non-nil lists exists. For instance, one can define

```
T tail < T > (Cons<T> l);
```

which is defined on its whole domain, unlike the ML

[1] We will thus sometimes use the word *class* to designate a type constructor.

[2] for monomorphic type. This definition of a monotype is inductive, the base cases being both type constructors of arity 0 and monotype variables, which will be introduced later.

[3] We will use a Java-like syntax, the new elements being described as they appear. Here, we introduce the notation for type parameters, written between angle brackets.

[4] This line is the declaration of a method `cons` with its type. It roughly corresponds to function prototypes in the C language.

[5] The quantification on `T` comes from the `<T>` after the name of the method.

```
'a list tail('a list)
```

that fails at runtime if `l` is nil.

In fact polytypes are universally quantified constrained monotypes. For example, we can defined a function that computes the sum of a list of numbers[6].

```
class Float {}
class Integer extends Float {}

T sum < T | T <: Float > (List<T>);
```

`sum` has type `List<T>` $\rightarrow$ `T`, for all `T` below Float. In particular, the sum of a list of integers has the (static) type `Integer`, a feature that would be impossible without constraints.

Method definitions

We have seen how to *declare* methods, that is to assert their existence and to specify their type. Methods must also be *defined* – attached to some code that defines their behavior when applied to arguments. To this end, each method is linked to several *alternatives*. Each alternative can restrict the domain of types that each argument can take. The selection of the alternative for the application of a method to arguments is done at runtime, based on the runtime type information of the arguments. This is a generalization – called multiple-dispatch – of the dispatch on the receiver of the methods that is present in many object-oriented languages[7].

For instance, here is a way of computing the length of a list:

```
Integer length < T > (List<T> l);

length(l@Nil)  = 0;
length(l@Cons) = 1+length(tail(l));
```

The first alternative(`l@Nil`) will be chosen if the argument is a nil list, the second one if it is a non-nil list. This resembles pattern-matching *à la* ML. An important difference is that pattern-matching works in a closed world: if one adds a case in a data type declaration, it will be necessary to modify existing code to add this new case to every "match" expression. On the other hand, the "alternatives" approach allows to define new method alternatives for this new case – possibly in a separate

[6] A simple numerical hierarchy is introduced here, but a more realistic presentation will be done in section 2.2

[7] Since methods dispatch on all their arguments, they are called multi-methods, in opposition to mono-methods, which dispatch only on their first argument, called the *receiver*. Since mono-methods have a privileged argument, they make it quite difficult to type binary methods.

file, which achieves better code locality. This may not seem important for a type like `List`, where the hierarchy is not bound to change. But in an abstract syntax tree with many different kinds of node, the extension of an existing hierarchy is quite common.

Fortunately, it is possible in both cases to have a static check that all possible cases for the arguments have been taken into account. In our case, this *coverage test* is done at link time, when the entire hierarchy is known. There is also a *non-ambiguity test*, that checks that for all possible call, a most-specific alternative exists[8].

It is interesting to notice the usage of method `tail`. Since it appears in an alternative where `l` is matched against `Cons`, the call `tail(l)` is well-typed. It has thus been possible to assert statically that the program will not try to take the tail of an empty list.

2.2 Need for interfaces

Let us see why $ML_{\leq}$ is both good and not sufficient for some useful and usual functions. A common example is the numerical hierarchy and the numerical functions. This hierarchy can be defined by:

```
class Float {}
abstract class Integer extends Float {}
class NativeInteger extends Integer {}
class BigInteger extends Integer {}
```

What is the type of the addition on numbers ? The basic idea is that the sum of two floats is a float, the sum of two integers is an integer, the sum of a float and an integer is a float [9]. More shortly and formally, the requirement is that "the type of the sum of two numbers is their least upper bound".

In the type system of many languages, it is not possible to give a function a type that matches this requirement. That is why arithmetic operators are treated apart [10] [11]. In $ML_{\leq}$ the most intuitive type we may write is:

[8]It may not be the case because of multiple inheritance, or for methods with at least two parameters.

[9]In Cecil[LC98], it has been proposed to represent this kind of type by such an enumeration. However, it not not satisfactory, since it does not work in an open world, where subclasses can be added after the type of the function has been given. Moreover, this approach leads to a very hish complexity of the type checking algorithm.

[10]But it should be remarked that this situation happens also with user defined types. So *ad hoc* typing is not sufficient.

[11]This problem is not even expressible in ML, which has no atomic sub-typing – that is, it is not possible to state that the `int` type is below the `float` type. That is why CAML, for instance, has unrelated functions for the addition of integers and for the addition of floats.

```
T plus < T | T <: Float > (T,T);
```

It is easy to see that this type meets the requirements stated above. But in a way, we see it's *too* precise: we don't want the sum of two `NativeInteger` to be a `NativeInteger`, but just an `Integer`, because this sum can be greater that any `NativeInteger`. In that case we want the result to be a `BigInteger`.

So we want to add to the above requirement that the type of the sum is always above `Integer`. That is to say that the new requirement becomes: "the type of the sum of two numbers is the upper bound of `Integer` and of their least upper bound". That is:

```
T plus < T | T >: Integer > (T,T);
```

However, this solution has the disadvantage that the type of `plus` can only be given once one knows the complete numerical hierarchy. For instance, if `Float` has an extension – say `IEEE_Float` – unrelated to `Integer`, the sum of two `IEEE_Float` will be a `Float` because `Float` is the least type `T` above `IEEE_Float` that satisfies the constraint `T>:Integer`. This may be what we want, but it may not. In any case, it is not satisfactory that the decision on the typing of `plus` on integers has this kind of "side-effect" on other unrelated types.

What we really wanted to express is that all classes below `Integer` are implementation classes, and thus have to be considered as equivalent to `Integer` as long as typing of numerical classes is concerned. That is what interfaces were designed for: to express some kind of equivalence relation on classes.

Interfaces

An interface is just a name, that represents a characteristic shared by a set of classes. In our example, we will declare an interface `Num`. It is intended to group numerical classes.

When a class `C` has the characteristic of an interface `I`, we will say that it *implements* this interface. For instance, `class Float implements Num`, since Float is a numerical class. We will also write `C:I`.

We also say that a class `C` *abstracts* an interface `I` if all the classes below it have to be considered equivalent to `C` *as long as the characteristic represented by* `I` *is concerned*. So in the numerical hierarchy, `class Integer abstracts Num`. The notation for abstraction is `C::I`.

Remark. It must noticed that the previous interpretation of abstraction supposes that a class that abstracts an interface also implements it. Although it seems that this should be the intended semantics when a user writes `class C abstracts I`, there is no theoretical reason for this. Furthermore, we will see in section 4.1 that it is sometimes useful to have a class that abstracts an interface without implementing it.

We can now give a satisfactory type to `plus`, that meet all the above requirements, including modularity:

```
T plus < T | T:Num > (T,T);
```
[12]

This new type succeeds in treating all forms of integers equivalently. For instance, suppose we add two `NativeInteger`s. Informally, since `NativeInteger` doesn't implement `Num` but is below `Integer` that abstracts it, the "best" solution for `T` is "`T=Integer`". The instance of the type of `plus` is thus `(T,T)`$\rightarrow$`T`, which means the only requirement for the return type is to be below `Integer`.

The reason why this solution is more modular than the previous one is that whenever we introduce a new class in the numerical hierarchy, we have the possibility to determine its behavior relatively to the `Num` interface.

This solution also gives an arguably more intuitive type to function `plus`: "`plus` has type `(T,T)`$\rightarrow$`T` for every numerical class `T`". In a sense, `NativeInteger` is *not* a numerical class, but an *implementation* of a numerical class.

The `Top` interface We introduce here an special interface called `Top`[13], since it will be used in the rest of this report. By definition, it is a super-interface of all other interface. Furthermore, every user-defined class implements it, and *final* classes abstract it.

Remark. `Top` is special in the sense that we don't want the user to have to assert the above properties, but is has no special treatement in the theory.

We will see in section 4.1 that `Top` enables us to model final classes.

3 Interfaces

In this section, we present the main ideas behind the extension of $\mathrm{ML}_{\leq}$ with interfaces. The formalization of this work can be found in appendix A.

To each group of type constructors with the same arity [14] we associate a set of *interfaces*, named i_C, j_C... This set is partially ordered by $\leq$. When $j_C \leq i_C$, we will say that interface j_C *extends* interface i_C.

For each interface, a predicate indicates which type constructors implement it, and which ones abstract it. We impose the following conditions:

- if a type constructor abstracts i_C, it is below every type constructor that implements i_C and with which it has a common lower bound; [15]

[12] The better-looking notation `T plus < Num T > (T,T);` has just the same meaning.

[13] In fact, we introduce a `Top<`n`>` interface for each arity n.

[14] Such a group is called a *type constructor* class in the $\mathrm{ML}_{\leq}$ terminology, but this overloads the name "class". The fact that each interface is associated to just one group is what makes them *structural* interfaces: there is no ambiguity about which kind of type constructors can implement or abstract a given interface.

[15] This rather un-intuitive condition is much simpler in the case of single inheritance. Then it becomes: "A type constructor that abstracts an interface has no type constructor strictly below it that implements this interface".

- if j_C extends i_C, then implementing j_C implies implementing i_C;
- there exists a type constructor that abstracts all interfaces.[16]

We now define the approximation relation. Its purpose is to indicate for each class a class it must be considered equivalent to, relatively to a given interface. Thus, in the example of section 2.2, we have $\texttt{NativeInteger} \xrightarrow{\texttt{Num}} \texttt{Integer}$[17]: as long as the numerical aspect is considered, class `NativeInteger` should be considered as equivalent to class `Integer`.

A crucial aspect is that this approximation relation must remain valid in all extensions of the current context[18]. This is necessary to guaranty that some form of modularity and separate type-checking can be achieved. So naturally, we state by definition that any extension is declared to be invalid if it does not satisfy this condition.

However, it would be a too easy way out if we didn't guaranty that it is possible to define useful valid extensions. The definition of the approximation relation has been carefully chosen for this purpose. In particular, no approximation can exist for a class except when a super-class abstracts a super-interface. The abstraction mechanism states that a class is the last one (the lowest) to implement some interface. So it gives us an information about all possible extensions. If an extension is invalid because the approximation relation is not preserved, it will be possible to explain it to the user in terms of an abstraction being violated (with syntactic restrictions about when it is possible to declare implementation). The formal definition of the approximation relation is the following:

We say that t_C approximates to t'_C relatively to i_C ($t_C \xrightarrow{i_C} t'_C$) if and only if there exists some interface k_C above i_C and some class t''_C above t_C that abstracts k_C, and t'_C is the least class above t_C that implements i_C.

This quite complicated definition is the generalization of two more intuitive cases:

- When t_C is a final class[19]: then t_C approximates to the least class above it that implements i_C.

  For instance, with a *final* class `NativeFloat` below `Float`, we would have $\texttt{NativeFloat} \xrightarrow{\texttt{Num}} \texttt{Float}$.

- When t'_C abstracts i_C[20]: then all class below t'_C approximate to t'_C for i_C.

  For instance, $\texttt{NativeInteger} \xrightarrow{\texttt{Num}} \texttt{Integer}$ since `Integer abstracts Num`.

[16] This condition is mostly technical.

[17] This reads "class `NativeInteger` approximates to class `Integer` relatively to interface `Num`".

[18] An extension can declare new classes and their relation to the current classes, new interfaces, and – in certain cases – declare that an old class implements or abstracts an interface.

[19] This case can be considered as a special case of the previous general rule in the following way: we can just take $k_C = \texttt{Top}$ and $t''_C = t_C$. The definition of `Top` can be found in section 2.2.

[20] For the general rule, take $k_C = i_C$ and $t''_C = t'_C$.

Now that this approximation relation is defined, it is possible to describe the changes to the axiomatization of constraint implication:

- Implementing an interface automatically implies implementing all its super-interfaces.
- When $t_C \xrightarrow{i_C} t'_C$, if a class ϕ_C is below t_C and below some (variable) class ϕ'_C that implements i_C, then it is true that ϕ'_C is above t'_C.

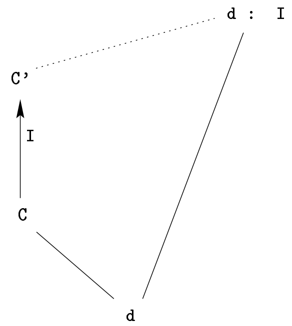

Figure 1: Axiom [21]

Once again, this second property is not very intuitive at the first sight ! Let us see is an example its meaning, with the unary negation method on the numerical hierarchy:

```
T minus < Num T > (T);

minus(x@NativeInteger)
{ ... }
```

When type-checking this method definition, one does the following:

- Introduce a type variable `X` which represents the runtime-type of `x`;
- From the definition of `minus`, we can suppose that `X <: T`;
- If this method body is chosen at runtime by the dispatch method, it will be true that `X <: NativeInteger`, so we assert this.

Since we know that $\texttt{NativeInteger} \xrightarrow{\texttt{Num}} \texttt{Integer}$ and that `T : Num`, we are exactly in the conditions of the above axiom. Thus it follows that `T` is above `Integer`. The conclusion is that the requirement on the return type is only that it should be below `Integer` – just what we expected.

[21] The lines without arrows represent ordering, the greatest classes being at the top. The arrows further represent the approximation relation, indexed by the related interface. The solid lines are hypothesis, and dashed lines are conclusions.

Appendix A is a formal presentation of those ideas, with the proofs of soundness, completeness and decidability.

In the next section, we will see how those concepts can be used in a real programming language, besides the need for types with interfaces we saw in section 2.2.

4 Bossa

This enhanced version of $ML_{\leq}$ with interfaces had to face the tough reality of the real world. To this end, we have designed a language, called Bossa, with the following goals:

- use the new type system to show that it can be implemented simply and efficiently, and that the use of interfaces is worth the pain in practice;
- show how advanced typing techniques can be integrated smoothly with well-known aspects of object-oriented languages.

To meet the second goal, it has been decided that Bossa should remain as close as possible to Java, both for syntax and semantics. It is thus simpler to learn Bossa for a Java programmer, since he only has to understand new concepts such as interfaces [22] and precise types.

4.1 Uses of interfaces

We saw in section 2.2 that interfaces are useful to type some methods. There are also other features of object-oriented languages that can be modeled with interfaces, even when it is convenient to have some kind of syntactic sugar to hide this fact.

Final classes

Many object-oriented languages have of notion of *final* classes. Final classes may not be extended further. One advantage to define a class as final is that it may represent a hint that allows some kind of optimization of the compiled code. A more fundamental use, in the case of $ML_{\leq}$, is that in a method alternative, if a parameter is matched against a final class, we know the type of the parameter will be exactly this class, not a subclass [23]. It may be of importance to type-check the body of the method.

Our point here is to remark that there is no need to explicitly treat final classes in the type system once it has interfaces. We use the `Top` interface that has been

[22] which in our vocabulary are very different from Java interfaces. This may be a source of confusion.

[23] We make here an abuse of language by confusing a class and the related type.

defined in section 2.2. Since declaring a class as final is only syntactic sugar for asserting that it abstracts `Top`, it becomes impossible to have a class strictly below it [24]. Under those hypothesis, it is easy to show that the axiom in $\mathrm{ML}_{\leq}$ for final classes in a consequence of the new axiom for interfaces, so it can be removed from the theory.

It is a relief to see that we can get rid of the concept of minimality – that was a part of $\mathrm{ML}_{\leq}$ – when we introduce interfaces. It allows the number of concepts to remain relatively small, keeping them orthogonal. It is also a good hint for the generality of interfaces.

Exact matching

Definition of method alternatives is an essential part of a language with multi-methods like Bossa. In many cases, the intended meaning of a pattern is to match any arguments that are in it. Taking an example with a polymorphic mono-method:

```
T f < T | T <: Float > (T);
f(x@Float) { ... }
```

In the alternative, we just know that x has some type below `Float`. In particular, the type of `f` makes it impossible to return a newly allocated `Float`, because in this context `T` represents any class below `Float` – so also `Integer` for instance.

But suppose an alternative of `f` is (or will be) defined for each class below `Float`. Then it would be sensible to have some way of declaring "I want only exactly this class to be matched, not one of its subclasses". In our example, this would be useful since it would then become type safe to return a `Float` [25].

However, it would be painful to modify the type system – and in particular the form of the patterns – since it would require to review all the proofs, especially the self-reduction proof, which is quite sensitive to such modifications. Once again, for our luck and relief, it appears that there is no need to modify the type system: interfaces plus syntactic sugar do the job. We get thus this new behavior for free.

Here is how it works. We use the same `Top` interface as before. Whenever a class `C` is defined, it is considered as syntactic sugar for the definition of two $\mathrm{ML}_{\leq}$ classes:

- One class `C` that has exactly the characteristics of the definition. In addition, it implements `Top`.
- One class `#C` that extends `C`, implements nothing and just abstracts `Top`. This is the class for runtime objects created with the construct `new C()`.

[24] It is indeed the way it is implemented in Bossa at the very moment.

[25] This example is inspired by a comment from a user of Jazz (Jean Vuillemin, an early and enthusiastic adept).

This implies that #C is equivalent to C relatively to Top – which means that those classes are indistinguishable from the language user point of view. But, if an alternative matches some parameter against #C, the parameter belongs exactly to class #C – it has runtime type #C.

In our example, it is now valid to write

```
f(x@#Float)
{ return new Float(1.23); }
```

In fact, this behavior would have been possible even without interfaces, if #Float had been declared to be a final class extending Float. But it that case, a counter-intuitive effect would arise: since a subclass of Float – say Integer – would not be a subclass of #Float, it would not be possible to return an Integer. That is

```
f(x@#Float)
{ return new Integer(42); }
```

would be ill-typed. This would really disturb the programmer, since, not being aware of the existence of the #Float class, he cannot see any good reason to reject this example. In our proposal, the expected behavior is obtained. The "# class" is really hidden. It would indeed be a good thing to have a syntax for x@#Float – maybe x#Float ? – so that the trick of the "# class" remains completely hidden.

So far, we have seen the utility of exact matching for writing polymorphic methods. There is at least another use for it, which has more to do with software-engineering. There are methods that should be re-implemented for each class that belong to its domain. A simple example is a getClassName method, which returns a string with the name of the class of the argument. Suppose we have the following hierarchy:

```
String getClassName <T> (T);

class A {}

getClassName(x@A) { return "A"; }
```

Now if we add a new class B below A, we should redefine getClassName for it. However, in all object-oriented languages, it is not possible to let the type-system check that for you. Indeed:

```
class B extends A

print(getClassName(new B()));
```

would be well-typed, and output "A". Now with exact-matching, it becomes possible to express "I define this method for argument of class A, but remind me of redefine it if I sub-class A." In the example, one just has to write:

```
String getClassName <T> (T);

class A {}

getClassName(x#A) { return "A"; }

class B extends A {}
```

At link time – when the method coverage test occurs – it will be detected that `getClassName` has not been redefined on `B`.

`getClassName` is a kind of "basic" method, but we think that this possibility is also useful for user-defined methods. For instance, one may think that the `hashCode` method should be redefined for every sub-class, since it should take into account every new field that could have been introduced.

Those two examples show the power of interfaces, since they illustrated very different uses that are needed in practice. This may advocate for the fact that interfaces are a basic building block for type systems, that may make them simpler since they reduce the number of necessary concepts. It is the task of the language level to partially hide their relative complexity with well chosen syntactic sugar constructs.

4.2 Particularities in the type system

Polymorphic references

Roughly speaking, the $\mathrm{ML}_{\leq}$ type system is used in Bossa. However, a particular problem had to be tackled: as a Java "descendant", Bossa is an imperative language. Since $\mathrm{ML}_{\leq}$ has polymorphic types, the well-known problem of polymorphic references arises. The basic example is the following:

```
class Ref<T>
{
  T content;
}

Ref<T> create <T> (T);
create<T>(value)
{
  return new Ref<T>(content=value);
}

void main(Array<String>);
main(args)
```

```
{
  <T> Ref<T> polyref=create(null);
  polyref.content=3;
  // well-typed, since polyref.content has type T for all T

  String s=polyref.content;
  // well-typed, but runtime error
}
```

The problem in this case is that we are able to declare a variable `polyref` of type $\forall$T `Ref<T>`. It is thus possible to view it both with type `Ref<String>` and type `Ref<Integer>`. Combined with side-effects (assignment), this is unsound.

Our solution, which is not really original, is to restrict types with a notion of well-formedness. We cannot completely forbid type parameters to appear in type constructors (like `T` in `Ref<T>`) since it would prevent to write functions on generic types (like create, or the map function on containers). So we make the distinction between normal type parameters and imperative type parameters, which are supposed to be ground. The later ones are introduced using angle brackets instead of curly brackets for the former ones.

So now we have to write:

```
class Ref<T>
{
  T content;
}

Ref<T> create <T> (T);
create<T>(value)
{
  return new Ref<T>(content=value);
}

void main(Array<String>);
main(args) {
  Ref<Integer> polyref=create<Integer>(0);
  polyref.content=3;
  // well-typed, since polyref.content has type Integer

  String s=polyref.content;  // ill-typed
}
```

It is not possible any more to store a polymorphic references, since its type being ill-formed, no varaible can be declared this type. In our example, we must now instantiate the type of `create` with a ground type parameter, and `polyref` is not a polymorphic reference anymore.

This restriction has not been the object of a formalization, so its only our belief that it is sufficient to guaranty self-reduction, that is the absence of runtime error for a well typed program. This formalization should be done in a near future.

5 Future work

- A self contained presentation of the extended version of $ML_{\leq}$ with interfaces as yet to be done. The current presentation is a "patch" to the first presentation of $ML_{\leq}$ [BM96]. We look forward the final version of Alexander Frey's PhD thesis [Fre99] for it will contain a new view on $ML_{\leq}$ that promises to be a nice framework in which to present our work.

- A formalization of imperative effects should be added to $ML_{\leq}$ in order to cover all theoretical aspects behind the Bossa programming language. We expect to find more work in the academic literature about it than about our kind of interfaces, since interaction between imperative aspects and polymorphic functions is a classical subject.

- While the interfaces presented here are structural, it may also be interesting to have some kind of *type classes* – that is non-structural grouping of classes, defined by an inference predicate *à la* Haskell – in $ML_{\leq}$. We anticipate that non-structural interfaces define on each type constructor class a structural interface, which means that it should be possible to add type classes to $ML_{\leq}$ without changing the whole theory.

- The implementation of Bossa now lacks at least an evaluation mechanism to make it possible to begin using the language and write more consequent examples. We expect this work to be more straightforward than the implementation of type-checking.

6 Conclusion

The extension of $ML_{\leq}$ with interfaces has been successful since it has been possible to formalize the theory and to modify the proofs accordingly, without a complete rewrite of the whole type system.

The system remains decidable, which leads to a naive algorithm for deciding constraint implication. Some simple optimizations of the algorithm have been found, even if a more detailed study and complexity computations are yet to be done. In practice, the addition of interfaces seems to keep $ML_{\leq}$ tractable.

The design and implementation of Bossa – still in progress – tend to show the possibility and the utility of using $\mathrm{ML}_{\leq}$ with interfaces in an higher-order object-oriented language.

All in all, this work on $\mathrm{ML}_{\leq}$ showed that its modular structure enables it to incorporate extensions quite easily. Together with the possible applications in language design, this makes it a promising and exciting field to work in.

A $\mathrm{ML}_{\leq}$ with interfaces

In this appendix, we present the extended theory of $\mathrm{ML}_{\leq}$ with interfaces. Since $\mathrm{ML}_{\leq}$ itself is a big theory, it was not possible to include a self-contained presentation here. The reader should consequently have knowledge of the initial presentation of $\mathrm{ML}_{\leq}$ which can be found in [BM96]. We present here the main differences introduced by interfaces, and the proofs that the properties remain valid.

A.1 Type structures

Definition of a type structure

$$C = (N_C, T_C, I_C, \__C^{IMP}, \__C^{ABS}, \sqsubseteq, \leq, \partial_C)$$

Definition : $\forall i_C \in I_C,\ \forall t_C \in T_C,\ i_C(t_C) = \{t'_C \mid t'_C \in i_C^{IMP} \wedge t_C \sqsubseteq t'_C\}$

1. $\__C^{ABS} \subseteq \__C^{IMP}$

   Each type constructor that abstracts an interface implements it as well.

2. $\forall i_C \in I_C,\ \forall t_C \in T_C,\ \forall t'_C \in i_C^{ABS},\ \forall t''_C \in i_C^{IMP}$,
   $t_C \sqsubseteq t'_C$ and $t_C \sqsubseteq t''_C \Rightarrow t'_C \sqsubseteq t''_C$

3. $j_C \leq i_C \Rightarrow j_C^{IMP} \subseteq i_C^{IMP}$

   If j_C extends i_C, then implementing j_C implies implementing i_C.

4. $\exists \perp_C \in T_C,\ \forall i_C \in I_C,\ \perp_C \in i_C^{ABS}$

   There exists a type constructor that abstracts all interfaces.

The approximation relation

For all $C \in \mathcal{T}$, $i_C \in I_C$, we define a binary relation $\xrightarrow{i_C}$ over T_C by :
$t_C \xrightarrow{i_C} t'_C$ iff

- $\exists k_C \geq i_C\ \exists t''_C \in k_C^{ABS}\ t_C \sqsubseteq t''_C$
- $t'_C = \min(i_C(t_C))$ (note that $t''_C \sqsubseteq t'_C$ by condition TS(2))

Lemma 1.

$$t_C \in i_C^{ABS} \Rightarrow t_C \xrightarrow{i_C} t_C$$

Proof:
Take $k_C = i_C$. Furthermore, condition TS(1) shows that $t_C \in i_C^{IMP}$, so $t_C = \min(i_C(t_C))$.

Definition of a type structure extension

A type structure $\mathcal{T}^*$ is an extension of $\mathcal{T}$ iff

1. $I_C^* \supseteq I_C$ and $\leq^* \supseteq \leq$
2. $\_{}_C^{IMP*} \supseteq \_{}_C^{IMP}$
   $\_{}_C^{ABS*} \supseteq \_{}_C^{ABS}$
3. $(t_C \xrightarrow{i_C} t'_C)_{\mathcal{T}} \Rightarrow (t_C \xrightarrow{i_C} t'_C)_{\mathcal{T}^*}$

Constraints

$$\kappa ::= \phi_C \sqsubseteq \phi_C | \phi_C : i_C | \kappa \wedge \kappa$$

We define inductively when a type structure $\mathcal{T} = \{(N_C, T_C, I_C, \_{}_C^{IMP}, \_{}_C^{ABS}, \sqsubseteq, \leq, \partial_C)\}$ satisfies a ground constraints κ :

$$\mathcal{T} \vdash t_C \sqsubseteq t'_C \text{ iff } t_C \sqsubseteq t'_C$$

$$\mathcal{T} \vdash t_C : i_C \text{ iff } t_C \in i_C^{IMP}$$

$$\mathcal{T} \vdash \kappa \wedge \kappa' \text{ iff } \mathcal{T} \vdash \kappa \text{ and } \mathcal{T} \vdash \kappa'$$

A.2 Constraint implication

[*ITrivImp*]	$\forall\vartheta.\ \kappa \models \kappa \wedge t_C : i_C$	$(t_C : i_C)$
[*IExtImp*]	$\forall\vartheta.\ \kappa\{\phi_C : j_C\} \models \kappa \wedge \phi_C : i_C$	$(j_C \leq i_C)$
[*IAbs*]	$\forall\vartheta.\ \kappa\{\phi_C \sqsubseteq t_C, \phi_C \sqsubseteq \phi'_C : i_C\} \models \kappa \wedge t'_C \sqsubseteq \phi'_C$	$(t_C \xrightarrow{i_C} t'_C)$
[*IEqImp*]	$\forall\vartheta.\ \kappa\{\phi_C = \phi'_C : i_C\} \models \kappa \wedge \phi_C : i_C$	

Figure 2: Axiomatization of constraint implication

A.3 Soundness proof

Lemma 2 (Soundness of implication). Let κ_1 and κ_2 be constraints and ϑ be a variable set such that $\forall\vartheta.\ \kappa_1 \models \kappa_2$ is one of the axioms of figure 2. Then for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$ and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 for the variables in ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.

For all axioms we take $\sigma_2 = \sigma_1$, written σ.

We have to distinguish between the side-conditions, which are supposed to hold in $\mathcal{T}$, and the constraints, wich concern $\mathcal{T}^*$.

[*ITrivImp*] Since $t_C : i_C$ holds in $\mathcal{T}$, it also holds in $\mathcal{T}^*$ by definition of type structure extension. Thus $(\kappa \wedge t_C : i_C)[\sigma]$ holds in $\mathcal{T}^*$ whenever κ holds in $\mathcal{T}^*$.

[*IExtImp*] Since $j_C \leq i_C$ in $\mathcal{T}$, $j_C \leq i_C$ in $\mathcal{T}^*$ by definition of a type structure extension. So if $\phi_C[\sigma] : j_C$ and $j_C \leq i_C$ in $\mathcal{T}^*$, then $\phi_C[\sigma] : i_C$ by condition TS(3).

[*IAbs*] Since $t_C \xrightarrow{i_C} t'_C$ holds in $\mathcal{T}$, it also holds in $\mathcal{T}^*$. So by definition $\exists t''_C \sqsupseteq t_C, \exists k_C \geq i_C$ such that $t''_C :: k_C$. By hypothesis, $\phi_C[\sigma] \sqsubseteq t_C$, so $\phi_C[\sigma] \sqsubseteq t''_C$. Since $k_C \geq i_C$, $\phi'_C[\sigma] : k_C$ by condition TS(3). So we can apply condition TS(2), and $t''_C \sqsubseteq \phi'_C[\sigma]$. So $t_C \sqsubseteq \phi'_C[\sigma]$ and since $\phi'_C[\sigma] : i_C$, $\phi'_C[\sigma] \in i_C(t_C)$. So by minimality, $t'_C \sqsubseteq \phi'_C[\sigma]$.

[*IEqImp*] Trivial since $\sqsubseteq$ is an order over T_C.

A.4 Completeness proof

$$\begin{array}{rl}
[\phi_C] \sqsubseteq^1 [\phi'_C] & \text{iff} \quad \forall\vartheta.\ \kappa_1 \models \phi_C \sqsubseteq \phi'_C \\
[\phi_C] \in i_C^{IMP^1} & \text{iff} \quad \forall\vartheta.\ \kappa_1 \models \phi_C : i_C \\
[\phi_C] \in i_C^{ABS^1} & \text{iff} \quad \exists t_C \in [\phi_C]\ t_C \in i_C^{ABS}
\end{array}$$

$\mathcal{T}^\vartheta = \{(N_C, C^\vartheta / \overset{\sim}{=}_C, I_C, \__C^{IMP^1}, \__C^{ABS^1}, \sqsubseteq^1, \leq, \partial_C)\} + T$ where T is basically unchanged : $T = (N_T, T^\vartheta / \overset{\sim}{=}_T, \emptyset, x \rightarrow \emptyset, x \rightarrow \emptyset, \sqsubseteq^1, \emptyset, ())$.

Lemma 3. $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$.

$\mathcal{T}^\vartheta$ is a type structure :

1. $[\phi_C] \in i_C^{ABS^1}$ Then $\exists t_C \in [\phi_C]\ t_C \in i_C^{ABS}$ in $\mathcal{T}$. So by condition TS(1), $t_C \in i_C^{IMP}$. Thus by [*ITrivImp*], $\forall\vartheta.\ \kappa_1 \models t_C : i_C$. That is to say $[t_C] \in i_C^{IMP^1}$ so $[\phi_C] \in i_C^{IMP^1}$.

2. Let $[\phi'_C] \in i_C^{ABS^1}$ and $[\phi''_C] \in i_C^{IMP^1}, [\phi_C] \sqsubseteq^1 [\phi'_C]$, $[\phi_C] \sqsubseteq^1 [\phi''_C]$.
   By definition of $\__C^{ABS^1}$, $\exists t_C \in [\phi_C]\ , t_C \in i_C^{ABS}$ in $\mathcal{T}$. So by lemma 1, $t_C \xrightarrow{i_C} t_C$. So by [*IAbs*], $\forall\vartheta.\ \kappa_1 \models t_C \sqsubseteq \phi''_C$. So $[\phi_C] = [t_C] \sqsubseteq [\phi''_C]$.

3. Suppose $j_C \leq i_C$.
   $\forall[\phi_C] \in j_C^{IMP^1}, \forall\vartheta.\ \kappa_1 \models \phi_C : j_C$. By [*IExt*], we have $\forall\vartheta.\ \kappa_1 \models \phi_C : i_C$. So $[\phi_C] \in i_C^{IMP^1}$.

4. $\perp_C \in i_C^{ABS^1}$ (identifying $\perp_C$ and $[\perp_C]$) by definition of $\_{}_C^{ABS^1}$.

Let us show that $\mathcal{T}^\vartheta$ is an admissible extension of $\mathcal{T}$.

1. The names of interfaces remain the same.
2. $\forall i_C, \forall t_C \in i_C^{IMP}$, we must show $[t_C] \in i_C^{IMP^1}$. Axiom $[ITrivImp]$ implies $\forall\emptyset.\ true \models t_C : i_C$. So $\forall\vartheta.\ \kappa_1 \models t_C : i_C$ and $[t_C] \in i_C^{IMP^1}$.

   If $t_C \in i_C^{ABS}$, then $[t_C] \in i_C^{ABS^1}$ by definition of $\_{}_C^{ABS^1}$. So $i_C^{ABS} \subseteq i_C^{ABS^1}$.
3. We suppose $t_C \xrightarrow{i_C} t'_C$ in $\mathcal{T}$.
   It follows that $t'_C = \min(i_C(t_C))$ and $\exists k_C \geq i_C\ \exists t''_C \sqsupseteq t_C\ t''_C \in k_C^{ABS}$.

   Let us show that $[t_C] \xrightarrow{i_C} [t'_C]$ in $\mathcal{T}^\vartheta$:

   - $[t''_C] \sqsupseteq [t_C]$ and $[t''_C] \in k_C^{ABS}$
   - $\forall[\phi_C]$ such that $[t_C] \sqsubseteq [\phi_C] \in i_C^{IMP}$, $\forall\vartheta.\ \kappa_1 \models t_C \sqsubseteq \phi_C : i_C, t_C \sqsubseteq t_C$. Since $t_C \xrightarrow{i_C} t'_C$, $[IAbs]$ implies that $\forall\vartheta.\ \kappa_1 \models t'_C \sqsubseteq \phi_C$, so $[t'_C] \sqsubseteq [\phi_C]$. This proves $[t'_C] = \min(i_C([t_C]))$.

   So $[t_C] \xrightarrow{i_C} [t'_C]$.

Lemma 4. κ holds in $\mathcal{T}^\vartheta$ iff $\forall\vartheta.\ \kappa_1 \models \kappa$

if part: By lemma 3, $\mathcal{T}^\vartheta$ is a type structure extension of $\mathcal{T}$. We choose the implicit substitution σ that maps variables ϕ_C to $[\phi_C]$. By construction, $\mathcal{T}^\vartheta \vdash \kappa_1$. Since κ is ϑ-closed, by lemma 2, $\mathcal{T}^\vartheta \vdash \kappa$.

only if part: $\kappa = \phi_C : i_C$
The assertion follows immediately from the definition of $i_C^{IMP^1}$.

A.5 Decidability of constraint implication

The structure of the proof is unchanged, since it is easy to see that all lemmas remain valid.

References

[BM96] François Bourdoncle and Stephan Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathmatiques Appliques, Ecole des Mines de Paris, Paris, March 1996.

[BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, January 1997. ACM.

[Fre99] Alexandre Frey. *Jazz*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 1999. To appear.

[LC98] Vassily Litvinov and Craig Chambers. Constraint-based polymorphism in cecil. Technical Report UW-CSE-98-01-01, University of Washington, 1998.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.