Type-checking multi-methods in ML (A modular approach)

Daniel Bonniot
INRIA Rocquencourt, France

daniel.bonniot@inria.fr

ABSTRACT

We propose a framework to describe type systems with type inference for ML-like languages. Our type systems are parameterized by the *type algebras* from which types are drawn. Type soundness is proved once-and-for-all under minimal assumptions on the type algebra.

Observing that $\mathrm{ML}_{\leq}$ types form a type algebra, we propose a decidable type system for a language *à la* ML with type inference and polymorphic multi-methods, thus solving a long standing problem. Furthermore, we define a sub-language to express classes (constructors), multi-methods (operators), and their static and dynamic semantics in a consistent manner, independently of the base language.

1. INTRODUCTION

Several propositions have been made to type-check multi-methods. Cecil [19] has a powerful type system, and the issue of modular type-checking has been explored extensively in that framework [20]. However, its polymorphic type system is not even known to be decidable. $\mathrm{ML}_{\leq}$ [4] is a decidable type system for multi-methods for which type-checking is known to be tractable. Unfortunately, it does not infer types for lambda-expressions. In this paper, we present a decidable type system for a language *à la* ML with polymorphic multi-methods. Although templates must be declared for multi-methods, type-inference is performed for all core ML expressions, as well as for definitions and invocations of multi-methods.

Additionally, we investigate a new approach to present type systems. We start from the observation that type systems for advanced features like multi-methods, type-classes, etc, are becoming increasingly complex. For instance, $\mathrm{ML}_{\leq}$, while only adding multi-methods to ML, used a presentation and proof scheme completely different from the well-known Hindley-Milner type system for ML. We think this is partially due to the monolithic presentation of type systems: types and expressions have a closed syntax. This implies that extensions of these type systems require a fresh proof of the whole new system; it is not obvious what properties an extension must meet not to compromise type soundness, thus making design choices more difficult; the resulting systems become large, and difficult to understand in their entirety; and no mechanism is provided to integrate seemingly orthogonal extensions in a straightforward way. Standard proof techniques for type soundness [26] ease this process. However, the full system still has to be reproved each time. Intuitively, it should be possible to parameterize the type system for a core language and its soundness proof by the type language. Thus, any specific type language meeting adequate requirements could be used to provide a sound type system for a specific language extension. This decomposition of the system in several layers makes the proofs easier to understand and reuse.

Another motivation for this work is to show that it is possible to model many language features as extensions of a small core, through the use of operators. This way, the presentation is easier to understand, and different extensions can be studied independently, without affecting the essential soundness properties. In particular, this approach applies to object-orientation. It can be made independent from the core language by considering methods as user-defined operators.

We follow these ideas to formalize a general framework for type systems parameterized by a *type algebra* in Section 2; we state how it can be instantiated, and we prove type soundness under minimal assumptions on the type algebra, following a standard proof scheme. We first instantiate the framework on Core ML and simple extensions of Core ML using the Hindley-Milner type system, to show the expressivity and simplicity of our system. In Section 3, we present our main result; that is, a decidable type system with type inference and polymorphic multi-methods, which is obtained by instantiation of our framework on the $\mathrm{ML}_{\leq}$ type system. Therefore, we claim that our approach is modular for three reasons:

- the general framework modularizes the presentation of type systems by abstracting over its concrete type algebra;
- we instantiate this framework to obtain a type system for multi-methods in gradual steps, formalizing the main notions behind multi-methods with the notion of *generic function*, independently of the concrete proposal and its corresponding type algebra;
- programs with multi-methods and modules can be type-checked separately.

Section 4 compares our approach to related work. In particular, we compare the general framework to $\mathrm{HM}(X)$ [21, 24], since we believe that our framework can also be used in different contexts than multi-methods. Full proofs are given in the appendices.

$$
\begin{array}{llcll}
\textsc{Var} & (\Gamma \vdash x) & = & \tau & (x : \tau \in \Gamma) \\
\textsc{Err} & (\Gamma \vdash x) & = & \mathbb{E} & (x \notin \mathsf{dom}(\Gamma)) \\
\textsc{Cst} & (\Gamma \vdash c) & = & \mathsf{type}(c) & \\
\textsc{Lam} & (\Gamma \vdash \lambda x.e) & = & \lambda t.\,(\Gamma, x : t \vdash e) & (t \notin FV(\Gamma)) \\
\textsc{App} & (\Gamma \vdash e_1\ e_2) & = & (\Gamma \vdash e_1)\ (\Gamma \vdash e_2) & \\
\textsc{Let} & (\Gamma \vdash \text{let } x_1 = e_1 \text{ in } e_2) & = & \big(\Gamma, x_1 : (\Gamma \vdash e_1) \vdash e_2\big) &
\end{array}
$$

Figure 1: The typing function

2. ALGEBRAIC TYPE SYSTEM

We present type-checking, type inference and soundness proofs for a core functional language. Instead of exposing the syntax and structure of types, we treat them as an abstract data type with two visible operations: the construction of functional types, and the application of one type to another.

Note that this application is not the application of a type constructor to a type. It is a meta-operator that, given the type of a function f and the type of an argument v, returns the type of the expression $f\ v$. For instance, suppose f and v have respectively the types `int` $\to$ `bool` and `int`. We shall give the expression $f\ v$ the type (`int` $\to$ `bool`) `int`. That is to say that (`int` $\to$ `bool`) `int` is the type of the results of a function of type `int` $\to$ `bool` applied to a value of type `int`.

Since we want to reason about the construction of types, we shall distinguish the *algebraic types*, that are provided by the instantiation of the system, and the *syntactic types*, that are constructed by our type system on top of the algebraic types. The instantiation of the system must interpret these syntactic types by providing the sub-type relation. Following up on the above example, we may assume that algebraic types are simple ground types built over the constants `int` and `bool`. That is `int`, `bool`, `int` $\to$ `bool` are algebraic types[1]. Conversely, (`int` $\to$ `bool`) `int` is a syntactic type which is equal to `bool` with respect to the sub-type relation.

2.1 Type Algebras

We postulate an arbitrary set $\mathcal{A}$ of *algebraic types.* We define *syntactic types* over $\mathcal{A}$ as the expressions τ of a small λ-calculus built over an infinite set of type variables t and over constants a of $\mathcal{A}$. That is:

$$
\begin{array}{lcll}
\tau & ::= & & \textit{Type} \\
 & & t & \textit{Type variable} \\
 & | & \lambda t.\tau & \textit{Functional type} \\
 & | & \tau\ \tau & \textit{Application type} \\
 & | & a & \textit{Algebraic type} \\
 & | & \mathbb{E} & \textit{Error type}
\end{array}
$$

Figure 2: Syntactic types

$\lambda t.\tau$ acts as a binder of variable t with scope τ. The set of free variables of a type τ, written $FV(\tau)$, and the substitution of type τ' for type variable t in type τ, written $\tau\,[t \leftarrow \tau']$, are then defined as usual. In the following, we use "type" as a shorthand for syntactic type.

A *type algebra* is a pair $(\mathcal{A}, \leq)$, where $\leq$ is a pre-order on the syntactic types built over $\mathcal{A}$ whose maximum element is $\mathbb{E}$. The relations $\geq$, $=$ and $\neq$ are defined as usual in terms of the $\leq$ relation.

In order to guarantee the soundness of our type system, we demand this pre-order satisfy:

Requirement 1 (Covariance).

$$
\frac{\tau' \leq \tau}{\tau_0\ \tau' \leq \tau_0\ \tau \qquad \tau'\ \tau_0 \leq \tau\ \tau_0 \qquad \lambda t.\tau' \leq \lambda t.\tau}
$$

Requirement 2 (Reduction).

$$
(\lambda t.\tau)\ \tau' \geq \tau\,[t \leftarrow \tau']
$$

The covariance requirement states that type application is covariant on both arguments. This is not in contradiction with the contravariance of the $\to$ type constructor. For instance with `int` $\leq$ `float`, the contravariance of $\to$ implies that `float` $\to$ `int` $\leq$ `int` $\to$ `float`. Therefore, it is required by our covariance rule that (`float` $\to$ `int`) `int` $\leq$ (`int` $\to$ `float`) `int`. This is indeed the case in a type algebra where the intuitive equalities (`float` $\to$ `int`) `int` $=$ `int` and (`int` $\to$ `float`) `int` $=$ `float` hold.

There is a strong correspondence in structure between these syntactic types $\tau\ ::=\ t \mid \lambda t.\tau \mid \tau\ \tau \mid a$ and the expressions of the lambda-calculus $e\ ::=\ x \mid \lambda x.e \mid e\ e \mid c$. In the same way, the reduction requirement corresponds to the β-reduction rule $(\lambda x.e)\ e' \longrightarrow e\,[x \leftarrow e']$ at the level of types. This parallel is natural if one considers types as an abstraction of expressions, the type of an expression giving partial knowledge of this expression. Thus, the typing function that maps an expression to its (principal) type is an abstraction function. This parallel will carry on in the study of the language and its typing.

2.2 Core Language

We consider the set of Core-ML expressions in Figure 3. The language of expressions is parameterized by a set of *constants* c. Each constant comes with an algebraic type noted $\mathsf{type}(c)$, and with a strictly positive integer arity noted $\mathsf{arity}(c)$. Constants are either *data constructors* C, or *operators* f. A data constructor C applied to $arity(C)$ arguments

[1] They happen to be also syntactic types, which contain all algebraic types, see Figure 2.

$$
\begin{array}{llll}
e & ::= & x \mid \lambda x.e \mid e\ e \mid \text{let } x = e \text{ in } e \mid c & \textit{Expression} \\
c & ::= & C \mid f & \textit{Constant} \\
v & ::= & \lambda x.e & \textit{Value} \\
 & \mid & C\ v_1\ ...\ v_n \quad \text{with } n \leq \mathsf{arity}(C) & \\
 & \mid & f\ v_1\ ...\ v_n \quad \text{with } n < \mathsf{arity}(f) & \\
\mathcal{E} & ::= & []\ e \mid e\ [] \mid \text{let } x = [] \text{ in } e & \textit{Context}
\end{array}
$$

Figure 3: Language syntax

or less is a value, while an operator f applied to exactly $arity(f)$ arguments is not a value, but must reduce. The (possibly non-deterministic) reductions of an operator f are parameterized by the set of $(arity(f)+1)$-tuples $R(f)$ used by rule rule Op in Figure 4.

$$(\lambda x.e)\ e' \longrightarrow e\,[x \leftarrow e'] \quad \beta$$

$$\text{let } x_1 = e_1 \text{ in } e_2 \longrightarrow e_2\,[x_1 \leftarrow e_1] \quad \beta\text{-Let}$$

$$\frac{(v_1, ..., v_n, e) \in R(f)}{f\ v_1\ ...\ v_n \longrightarrow e}\ \text{Op} \qquad \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}\ \text{Ctxt}$$

Figure 4: Semantics

Definition 1. *A* typing context Γ *is a list* $x_1 : \tau_1, ..., x_n : t_n$ *of bindings. An expression variable can appear at most once in a typing context. The free type variables of* Γ *are* $FV(\Gamma) = \cup_{x:\tau \in \Gamma} FV(\tau)$

Given a typing context, we define a function computing the syntactic type of any expression. This typing function is defined recursively in Figure 1. Note that it is valid to use $\mathsf{type}(c)$ as a syntactic type, since the grammar of Section 2.1 implies that syntactic types are a super-set of algebraic types.

Definition 2.
An expression e *is* well-typed *if* $(\emptyset \vdash e) \neq \mathbb{E}$.

To ensure type soundness, we impose the following requirements on constants:

Requirement 3 (Constants).

1. *If* $n = \mathsf{arity}(f)$ *and* $f\ v_1\ ...\ v_n$ *is well-typed then there exists an expression* e' *such that* $f\ v_1\ ...\ v_n \longrightarrow e'$
2. *If* $f\ v_1\ ...\ v_n \longrightarrow e'$ *then* $(\emptyset \vdash f\ v_1\ ...\ v_n) \geq (\emptyset \vdash e')$
3. *The expression* $C\ v_1\ ...\ v_n$ *is not well-typed for any* $n > \mathsf{arity}(C)$

The type system is sound with respect to the semantics of Figure 4.

Theorem 1 (Soundness).
If e *is well-typed then either all reductions of* e *are infinite, or there exists a value* v *such that* $e \longrightarrow ... \longrightarrow v$ *and* v *is well-typed. Furthermore,* $(\emptyset \vdash e) \geq (\emptyset \vdash v)$.

The proof scheme is standard. As usual, soundness is proved by combination of subject-reduction and progress lemmas. However, a major novelty is that the whole proof can be done by using the typing functions and the two requirements on type algebras. This is mere calculation on types, as opposed to the creative reasoning usually required. This simplification is possible because we have a calculus of type expressions. The proof is thus short and easy to check step by step. It should furthermore ease automatic proof checking.

Two lemmas describe the properties of the typing function. The substitution lemma is standard. The context lemma is more unusual: the type of an expression e in the context $x : \tau$ is the same as its type in context $x : t$ in which the fresh type variable t is afterwards replaced by τ. Because of the covariance requirement on type algebras, an interesting consequence is that the type of e can be seen as a covariant function of τ.

Lemma 1 (Substitution).

$$\big(\Gamma \vdash e\,[x \leftarrow e']\big) = \big(\Gamma, x : (\Gamma \vdash e') \vdash e\big)$$

Lemma 2 (Context).

$$(\Gamma, x : \tau \vdash e) = (\Gamma, x : t \vdash e)\ [t \leftarrow \tau] \quad (t \notin FV(\Gamma))$$

Lemma 3 (Subject reduction).

$$\frac{e \longrightarrow e'}{(\emptyset \vdash e) \geq (\emptyset \vdash e')}$$

Lemma 4 (Progress). *If* e *is well-typed then either* e *is a value, or there exists an expression* e' *such that* $e \longrightarrow e'$.

The full proofs are given in Appendix A. Note that we chose to present semantics with a full-beta reduction. The call-by-value or call-by-name semantics fit equally well and are also sound. Furthermore, it is straightforward to impose a deterministic evaluation order. It is easy to see that these semantics have fewer reductions than ours, so that subject-reduction still holds. The progress lemmas for call-by-name and call-by-value can be easily proved with the usual arguments[2].

This finishes the presentation of the general framework. It can be instantiated by providing a domain for (algebraic) types and a pre-order on syntactic types that satisfies requirement 1 (Covariance) and requirement 2 (Reduction). The resulting type system is clearly decidable if this pre-order is decidable.

Additionally, this presentation suggests an implementation parameterized by the chosen domain for algebraic types and by the implementation of operations over algebraic types. The types can thus be manipulated by a compiler as the fol-

[2]for call-by-value, the usual requirement that the value bound be well-typed must additionally be required.

lowing abstract data type, written with the Ocaml syntax[3]:

```
module type TypeAlgebra =
  sig
    type t
    val apply : t → t → t
    val lambda : t → t → t
    val fresh_var : unit → t

    val error : t
    type constant = {name : string, constant_type : t}
    val constants : constant list
  end
```

We present in this paper two such instantiations with their full proofs: the Hindley-Milner type system in the next section, and the $\mathrm{ML}_{\leq}$ type system in Section 3.3. This shows that our system is general enough to express complex and various type systems, while factorizing a substantial part of the soundness proofs.

2.3 Hindley/Milner type system

We show here as an example that Hindley/Milner polymorphic types form a type algebra. Our framework can thus be used to present the type system of ML with a novel and concise proof of its correctness.

The syntax for types is recalled in Figure 5. Type variables are denoted by α as usual, while types are denoted by θ instead of τ to avoid confusion with syntactic types. Type schemes are considered equal modulo renaming of bound type variables and removal of quantified type variables not present in the type.

$$
\begin{array}{rcll}
\iota & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{Base type} \\
\theta & ::= & \iota \mid \theta \to \theta \mid \alpha & \textit{Type} \\
\sigma & ::= & \forall \overline{\alpha}.\theta & \textit{Type scheme}
\end{array}
$$

Figure 5: Type syntax

2.3.1 Instantiation of the framework

To instantiate the framework, we have to provide a pre-order on the syntactic types build over type schemes that verifies the requirements of Section 2.1. To this end, we define in Figure 6 the auxiliary function $\langle\cdot\rangle$ that translates syntactic types into *constrained* type schemes. Every syntactic type variable t is mapped to a distinct type variable α_t. The translation of $\lambda t.\tau$ is basically $\langle t\rangle \to \langle\tau\rangle$, except that generalization on α_t is done immediately. It should not be surprising that unification occurs during type application. In particular this fact makes explicit that in the usual ML rule

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

the two occurrences of τ in the premises amount to an equality constraint between two *a priori* different types.

[3]The `lambda` function is allowed to be partial in its first argument, since it is only applied to the results of `fresh_var` (i.e. to type variables), not to arbitrary types.

$$\langle\forall\overline{\alpha}.\theta\rangle = \forall\overline{\alpha}.\theta, \emptyset \qquad \langle t\rangle = \forall\emptyset.\alpha_t, \emptyset$$

$$\langle\mathbb{E}\rangle = \forall\alpha.\alpha, \texttt{int} = \texttt{bool} \qquad \frac{\langle\tau\rangle = \forall\overline{\alpha}.\theta, C}{\langle\lambda t.\tau\rangle = \forall\overline{\alpha}\alpha_t.\, \alpha_t \to \theta, C}$$

$$\frac{\langle\tau_1\rangle = \forall\overline{\alpha}_1.\theta_1, C_1 \qquad \langle\tau_2\rangle = \forall\overline{\alpha}_2.\theta_2, C_2}{\langle\tau_1\ \tau_2\rangle = \forall\overline{\alpha}_1\overline{\alpha}_2\alpha.\alpha, C_1 \cup C_2 \cup \{\theta_1 = \theta_2 \to \alpha\}}\ \alpha \text{ fresh}$$

Figure 6: Translation for Hindley/Milner

We define the order on the type algebra in the following way. We shall denote ground substitutions by ρ, and write $\rho(C)$ if for each $\theta_1 = \theta_2$ in C, $\rho(\theta_1)$ is syntactically equal to $\rho(\theta_2)$. Let $\langle\tau_1\rangle$ be $\forall\overline{\alpha}_1.\theta_1, C_1$ and $\langle\tau_2\rangle$ be $\forall\overline{\alpha}_2.\theta_2, C_2$. The relation $\tau_1 \leq \tau_2$ holds if and only if for all ground substitution ρ_2 such that $\rho_2(C_2)$, there exists a substitution ρ_1 equal to ρ_2 outside $\overline{\alpha}_1$ such that $\rho_1(C_1)$ and $\rho_1(\theta_1) = \rho_2(\theta_2)$.

It is worth noting that this relation always holds if C_2 is not satisfiable. Since the error type is maximal, this means that syntactic types that generate unsatisfiable constraints are equal to the error type. Therefore, values of these types are not well-typed.

We shall consider constrained type schemes modulo the equivalence relation implied by this pre-order. This opens the opportunity for type simplification, that follows directly from the definition of the order:

Lemma 5 (Simplification). *The constrained type scheme $\forall\overline{\alpha}.\theta, C$ is equivalent to $\forall\overline{\alpha}'.\theta', C'$ where $\theta' = \mathsf{mgu}(C)(\theta)$, $\overline{\alpha}' = \overline{\alpha} \cap FV(\theta')$ and $\mathsf{mgu}(C')$ is equal to $\mathsf{mgu}(C)$ point-wise outside $\overline{\alpha}$.*

We can now assert that we have formed a type algebra in the sense of Section 2.1.

Theorem 2 (ML). *The set of Hindley-Milner type schemes together with the pre-order on the syntactic types built over them form a type algebra.*

The full proof is given in A.2. It only deals with technical properties of the pre-order we introduced. This fact shows that we indeed achieved to factorize substantial parts of the soundness proof.

2.3.2 Example

As an example, let us consider the typing of $(\lambda x.x)\ 1$. By the rules of the algebraic type system, this expression has syntactic type:

$$
\begin{array}{cl}
 & (\emptyset \vdash (\lambda x.x)\ 1) \\
= & (\emptyset \vdash \lambda x.x)\ (\emptyset \vdash 1) \\
= & \lambda t.\,(x : t \vdash x)\ \texttt{int} \\
= & (\lambda t.t)\ \texttt{int}
\end{array}
$$

This type can now be translated to a constrained type scheme. We first translate each side of the application: $\langle\lambda t.t\rangle =$

$\forall \alpha_t . \alpha_t \rightarrow \alpha_t, \emptyset$ and $\langle \texttt{int} \rangle = \texttt{int}, \emptyset$. The system of constraint generated by the application is therefore $\alpha_t \rightarrow \alpha_t = \texttt{int} \rightarrow \alpha$. The considered expression has the constrained type scheme $\forall \alpha_t \alpha . \alpha, \{ \alpha_t \rightarrow \alpha_t = \texttt{int} \rightarrow \alpha \}$. By lemma 5 (SIMPLIFICATION) this type simplifies to $\texttt{int}, \emptyset$.

2.3.3 Beyond Core-ML

So far, we have only considered the features of Core-ML. A real ML language usually has much more features. Data-types and pattern matching can be added easily as in ML. Each data-type declaration can be considered as syntactic sugar for the introduction of a new type name, constructors for the different cases and a matching operator. Pattern-matching can then be seen as syntactic sugar for the application of the matching operator, including matching arbitrarily deep structures, default cases and textual ordering of the branches.

Exceptions and references could also be added to the core language. However, references are not a simple instance of the framework, but require an extension to the framework. As for ML, one could augment the semantics with a global store, and provide references via primitives; this would also require restriction of polymorphism to values. This restriction can be handled in the type algebra, through the translation function defined in this section to instantiate the framework. Therefore the typing part of the framework can be left untouched. It would actually be interesting to study how the semantics itself could be made a parameter of our system, so that features requiring specific semantics could be added without modifying the framework at all.

3. MULTI-METHODS

Traditionally, new language features are presented by enriching the type system, extending the expression language and modifying the semantics. In this section, we try to show how this can be avoided. In Section 2, we proposed a type system that is parametric in the domain of types, which already allows for enrichment of the type system.

It remains to show how to extend Core-ML without changing the expression language. It is already standard to define primitive operations on base types using *constants*. More precisely, constructors are used to build values of base types, while operators are used to destruct them, as described by specific reduction rules. For instance, instead of adding a case $e ::= ... \mid e + e$, a special typing rule

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

and a reduction rule for $e + e$ nodes, an operator $+$ of type $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$ is defined together with the appropriate reduction.

However, more complex features involve the use of more complex operators with semantics that depend on the program. For instance, the reduction of method calls in object-oriented languages depends on the user-defined class hierarchy. Thus, we believe that formalizing these features reduces to providing the programmer a language in which to define new operators and their reduction rules.

We shall now follow this approach to present a simple decidable type system for languages with multi-methods. The presentation is decomposed in several steps so as to introduce the key ideas one by one. Section 3.1 presents a general notion of *generic function*, at first in the closed-world view, then in the open-world view. We propose in Section 3.4, as a particular case of generic functions, an extension for object-orientation with full-fledged multi-methods. This extension is based on the construction of the $\mathrm{ML}_{\leq}$ type algebra, made in Section 3.3.

3.1 Generic functions

3.1.1 Example

For this example we assume a type algebra that has bounded polymorphic types. Base types include `int`, `float`, and `num` such that $\texttt{int} \leq \texttt{float} \leq \texttt{num}$. Type `num` is abstract in the sense that it has no direct instance. Conversely, `int` and `float` are concrete. As this example is only meant to support intuition, we deliberately remain informal.

Assume given two monomorphic functions `opp_float` and `opp_int` that compute the unary minus function on floats and integers. We would like to define a function that computes the opposite of any number. The function `opp_float` could be used, because an integer can be considered as a float by subsumption. However, in that case the result has type `float`, while we know statically that it is `int`. Bounded polymorphism allows to give it the strictly more precise type $\forall t \leq \texttt{num}. t \rightarrow t$. However, `opp_float` cannot be given this type. Furthermore, the only function with this type that can be written in Core-ML is the identity on `num`.

What we need is a *generic function* construct that pattern matches on the runtime-type of its arguments. In our example:

$$\textbf{generic}\ \texttt{opp}\ (\forall t \leq \texttt{num}. t \rightarrow t) = \begin{array}{l} \texttt{float} \Rightarrow \texttt{opp\_float} \\ \texttt{int} \Rightarrow \texttt{opp\_int} \end{array}$$

The intent is that one of the implementations of the generic function will be selected and applied, at runtime, depending on the shape of the argument. Each implementation branch uses a list of *patterns* to declare the shape of arguments it handles, here the single-element lists `float` and `int`. No branch is needed for `num` since it is an abstract type.

3.1.2 Syntax

The syntax for programs with generic functions is:

$$\begin{array}{rcll} G & ::= & \textbf{generic}\ \mathsf{g}\ (\tau) = I_1, ...I_p & \textit{Generic function} \\ I & ::= & \overline{\pi} \Rightarrow e & \textit{Implementation} \\ P & ::= & \textbf{let rec}\ \overline{G}\ \textbf{in}\ e & \textit{Program} \end{array}$$

where π belongs to a language of *patterns*. A pattern represents the set of types that belong to that pattern, as defined by the binary predicate $\tau \in \pi$. Both the set of patterns and this predicate are left abstract at this stage. A semantics for programs with generic functions will first be defined; then we shall present type-checking rules and express the properties these abstract components must verify to ensure type soundness. We will provide a concrete instance of generic functions in Section 3.4.

3.1.3 Semantics

The applicability of implementations is formalized in the following definition:

DEFINITION 3. *An implementation* $\pi_1 ... \pi_n \Rightarrow e$ matches $(\tau_1, ..., \tau_n)$ *if for all* i *in* $1..n$, $\tau_i \in \pi_i$

The declaration **generic g** $(\tau_g) = \overline{\pi}_1 \Rightarrow e_1, ..., \overline{\pi}_p \Rightarrow e_p$ introduces an operator g in the language, with $type(g) = \tau_g$. The reduction rules for this operator are defined by:

$$\frac{\overline{\pi}_i \Rightarrow e_i \text{ matches } ((\emptyset \vdash v_1), ..., (\emptyset \vdash v_n))}{(v_1, ..., v_n, e_i\ v_1\ ...\ v_n) \in R(g)}$$

Hence, we have defined an operator that fits the general framework of Section 2. In particular, the derived rule obtained by combination with the reduction rule for operators OP is then:

$$\frac{\overline{\pi}_i \Rightarrow e_i \text{ matches } ((\emptyset \vdash v_1), ..., (\emptyset \vdash v_n))}{g\ v_1\ ...\ v_n \longrightarrow e_i\ v_1\ ...\ v_n}$$

Note that the semantics is possibly non-deterministic since several implementations might match a given tuple of arguments. In practice, it is often desirable to enforce a deterministic behavior. This can be achieved by requiring a *non-ambiguity* property on generic functions. This extends the coverage property by demanding that there exist a *most specific* matching implementation for all possible calls. Reduction then happens unambiguously by selection of the most specific implementation. This aspect is well-known from the study of multi-methods [6, 4], and is orthogonal to type-soundness.

The semantics is typed in the sense that the types of the values are used in the definition of the reduction rule. However, this does not mean that expressions have to carry a type at run-time. In opposition to dynamics values (values carrying runtime types on which type matching can be performed, as in [18, 1]), types involved here are not static types carried at runtime, but types computed at runtime. This computation might be straightforward as only values have to be typed. Furthermore, the whole type might be unnecessary to perform the match test, depending on the actual pattern language used.

Note that since g is an operator, it is made available globally in the program by CST. It can therefore appear in e, so that generic functions are globally mutually recursive. In particular, this allows for polymorphic recursion. This does not raise undecidability problems since the type of the generic functions is declared and not inferred.

3.1.4 Type-checking

We now present a static type system to detect errors in programs with generic functions. Thanks to our general soundness results, theorem 1, this reduces to proving that generic functions define operators that satisfy requirement 3 (CONSTANTS).

Informally, the type of a generic function poses two requirements on its implementations. Firstly, it can be seen as a specification that every implementation must meet. However, since the patterns restrict the domain of the arguments, this type can be specialized, taking into account the information in the patterns. Thus, we require a notion of *restriction of a type to a list of patterns*, and we demand that each implementation has a type which is equal to the restriction of the generic type to the implementation's patterns. For instance, the restriction of the type of `opp` to pattern `float` is `float` $\rightarrow$ `float`. Since it would not make sense to define an implementation for a case that is never used, we also demand that the restricted type is not $\mathbb{E}$. Thus, an implementation with pattern `string` would be rejected.

Formally, we first demand a total function $\mathsf{restrict}$ such that $\mathsf{restrict}(\tau, \overline{\pi})$ is the restriction of a type τ to a list of patterns $\overline{\pi}$. The given components must satisfy the following requirement:

REQUIREMENT 4 (RESTRICTION).

$$\Downarrow \frac{\tau_i \in \pi_i \ (i = 1..n)}{\mathsf{restrict}(\tau, \overline{\pi})\ \tau_1\ ...\ \tau_n \leq \tau\ \tau_1\ ...\ \tau_n}$$

DEFINITION 4. *The declaration* **generic g** $(\tau) = \overline{\pi}_1 \Rightarrow e_1, ...\overline{\pi}_p \Rightarrow e_p$ *is valid if for all* i,

- $\mathsf{restrict}(\tau, \overline{\pi}_i) \neq \mathbb{E}$
- $(\emptyset \vdash e_i) \leq \mathsf{restrict}(\tau, \overline{\pi}_i)$

The first condition prevents from using patterns that are incompatible with the domain of the generic function and that would therefore never be used. Hence it is not mandatory, but good sense to require this first condition. Conversely, the last condition is essential to ensure subject-reduction.

The second requirement on the implementations of a generic function is fulfill the "progress" part of requirement 3, that is its first point. To achieve this, it must be noticed that the type of the generic function defines its domain. The implementation branches, taken together, must cover this domain. Back to the introductory example, the generic function `opp` must possess an implementation for any non-abstract subtype of `num`. It is indeed not necessary that an implementation exists for types that have no runtime values. We therefore define run-time types and *covered generic functions*:

DEFINITION 5. *A type* τ *is a* run-time type *if there exists a value* v *such that* $(\emptyset \vdash v) = \tau$.

DEFINITION 6. *A generic function* **generic g** $(\tau) = I_1, ...I_p$ *is* covered *if for all run-time types* $\tau_1, ..., \tau_n$ *such that* $\tau\ \tau_1\ ...\ \tau_n \neq \mathbb{E}$, *there exists an index* i *such that* I_i *matches* $(\tau_1, ..., \tau_n)$.

It is now possible to express a sufficient condition for programs with generic functions to be sound.

THEOREM 3 (GENERIC FUNCTIONS). *The operators defined by valid and covered generic functions satisfy requirement 3 (*CONSTANTS*).*

The first part of the requirement is directly implied by definition 6 of covered generic functions together with the reduction rules. The second part is a consequence of definition 4 of a valid implementation, using requirement 4 on the restriction operator.

A program **let rec** $\overline{G}$ **in** e is well-typed if every generic in $\overline{G}$ is valid and covered, and if e is well-typed. The result of the program is the evaluation of e. By theorem 1 (Soundness) and theorem 3 (Generic functions), e reduces to a value v such that $(\emptyset \vdash e) \geq (\emptyset \vdash v)$.

In [9], Chambers and Leavens present an efficient algorithm for testing coverage (and non-ambiguity, to ensure determinism). They also consider a conformance test. However this is unnecessary in our framework. Moreover it would break the modularity property, as we discuss in Section 4.

3.2 Open generic functions

It has already been recognized [11] that the activity of programming has two main sides: defining operations and defining data structures. The functional paradigm mainly uses sum and product types as its data structures, and functions defined by pattern-matching on data-types as its operations. The object-oriented paradigm provides classes to structure data, and methods to operate on it. However, both paradigms introduce an asymmetry between the two concepts. In a functional program, data-types can be defined independently of functions, while functions need knowledge about the data-type constructors. Conversely, methods are defined locally to a class, while classes include the list of all their methods.

This asymmetry is problematic when it comes to modular programming, that is, programming using pre-existing libraries, without changing them [17]. Following the above dualism, modular programming is thus both defining new operations on existing data structures, and defining new data structures to be handled by existing operations. In the functional paradigm, defining new functions is straightforward. On the other hand, extending existing data-types is not possible since it would break existing functions defined by pattern matching on this type: they would miss the new cases. Conversely, extending data structures amounts to writing new classes, which is precisely the object-oriented paradigm, while defining new methods on existing classes is not allowed.

In Section 3.1, all implementations of a generic function had to be syntactically present together with the definition of the function. This is similar to pattern matching in a functional language, and thus fails in the same way for modular programming. The solution is to *open* generic functions, so that implementation branches can be defined independently, which brings back symmetry. An open generic `opp` can thus be defined as:

$$
\begin{array}{l}
\textbf{generic}\ \texttt{opp}\ :\ \forall t \leq \texttt{num}.\, t \rightarrow t \\
\textbf{implementation}\ \texttt{opp float} \Rightarrow \texttt{opp\_float} \\
\textbf{implementation}\ \texttt{opp int} \Rightarrow \texttt{opp\_int}
\end{array}
$$

If the new numeric type of complex numbers is added in a different module that imports the above one, a new implementation of `opp` can —and must— be provided:

$$\textbf{implementation}\ \texttt{opp complex} \Rightarrow \ldots$$

For the rest of the paper, we shall assume a simple class-based object-oriented setting. In particular, we consider that a class definition `class C` introduces a type `C`. An abstract class has no direct instance. A non abstract class `C` can be constructed using the `new_C` operator. Class members are solely fields, and are inherited from super-classes. Finally, subtyping follows inheritance.

We illustrate the two situations encountered with modular programming by taking the example of a small programming language implementation. The structure of this implementation is presented in Figure 7.

```
module CORE
  abstract class Expression = {}
  class Apply inherits Expression =
    { f : Expression, arg : Expression }

  generic eval : Expression → Expression
  generic print : Expression → String

  implementation eval Apply ⇒ ...
  implementation print Apply ⇒ ...
end module

module NUMERIC imports CORE
  class IntegerLiteral inherits Expression = ...

  implementation eval IntegerLiteral ⇒ ...
  implementation print IntegerLiteral ⇒ ...
end module

module COMPILER imports CORE and NUMERIC
  generic compile : Expression → Bytecode

  implementation compile Apply ⇒ ...
  implementation compile IntegerLiteral ⇒ ...
end module
```

Figure 7: Programming with open generic functions

3.2.1 Syntax and semantics

The syntax for programs with open generic functions is:

$$
\begin{array}{rcll}
G & ::= & \textbf{generic}\ \mathsf{g}\ :\ \tau & \textit{Declaration} \\
I & ::= & \textbf{implementation}\ \mathsf{g}\ \overline{\pi} \Rightarrow e & \textit{Implementation} \\
M & ::= & \textbf{imports}\ \overline{M};\ \overline{G \mid I} & \textit{Module} \\
P & ::= & \textbf{imports}\ \overline{M}\ \textbf{eval}\ e & \textit{Program}
\end{array}
$$

A natural semantics is defined by translation into the generic functions of Section 3.1. The generic functions are reconstructed by grouping the implementations with their respective declarations. The program can then be evaluated as before.

3.2.2 Type-checking

A module type-checks if all its implementations are *valid*. Since the definition of *valid* is independent of the other implementations, modules can be type-checked separately. Conversely, checking coverage and non-ambiguity can only be done for the (whole) program, that is when all the implementations and types are known.

However, this scheme for checking coverage is problematic because it leaves much freedom about the module in which to define a certain implementation. This is arguably a problem from the software-engineering point of view. For example, it would be possible in our compiler example of Figure 7 to omit the implementation of `eval` in module `NUMERIC`, while this module would then pass type-checking. At the point of linking the modules together, the coverage test would fail for `eval`. Solving this failure would amount to adding the missing implementation. But since it logically belongs to module `NUMERIC`, this solution breaks modularity by forcing the update of an already compiled module.

Therefore, we believe it is a good design to check coverage in every module. This check is done in the typing context made of all declarations present in this module and all the modules it imports transitively. Thus, apart from errors local to the module, the coverage test can only fail when a module imports a generic function, and defines new types in its domain without providing the corresponding implementation, or when it imports two modules leading to the same situation. These are indeed the situation where an implementation logically belong to this module. The error can then be solved locally by including this implementation in the module, without breaking any existing code. This coverage testing scheme also enforces an intuitive organization of code:

- when a module extends an existing data structure, it must define the implementations for all the generic functions that operate on this data structure;
- when a module defines a new generic function, it must define its implementations for all the data structures in its domain.

The list of all generic functions and data structures is drawn from the typing context defined at the beginning of this paragraph. An example of the first point is the implementation of `eval` in module `NUMERIC`; the implementations of `compile` in module `COMPILER` illustrates the second point.

Furthermore, it is possible to make this coverage test unnecessary, at the price of a loss of generality for generic functions. A well-known case is mono-methods, used in object-oriented languages with single dispatch. These methods select an implementation based on the runtime type of a distinguished "receiver" argument. Furthermore they are always defined together with the class for which they provide an implementation. The coverage test then reduces to checking that an implementation is present if none is inherited from the parent classes – which could happen if the parents are abstract. This check can therefore be done with the sole knowledge of the class definition and of its parents. In our presentation, this corresponds to restrict implementations to only one pattern on the first argument. It is then sufficient to check that whenever a new type is defined, all necessary implementations are also provided. More elaborate restrictions for multi-methods are studied in [20].

We discuss briefly the issue of inference for open generic functions in Appendix A.3.1.

3.3 ML$_\leq$

ML$_\leq$ [3, 4] is a rank-1 polymorphic constrained type system. It has been developed to type an extension of ML with multi-methods and object-orientation. However, it requires type annotations on lambda-expression, and thus lacks type inference. Furthermore, the presentation in [3, 4] is *ad-hoc* and rather unusual, making it difficult to study.

Here, we give a new presentation of ML$_\leq$ as an instance of our framework. We observe that ML$_\leq$ types[4] form a type algebra that can be used with the algebraic type system of Section 2. We shall use this instance in section 3.4 to present a concrete instantiation of generic functions to model multi-methods.

3.3.1 Definition of ML$_\leq$

The syntax for constrained types is given in Figure 8. The definition of the constraint implication relation $\models$ is included in Appendix A.4.

Let $\tau_1 \equiv \forall \bar{t}_1.\, \kappa_1 \Rightarrow \theta_1$ and $\tau_2 \equiv \forall \bar{t}_2.\, \kappa_2 \Rightarrow \theta_2$ be two types. The subtyping relation $\tau_1 \leq \tau_2$ holds if and only if

$$\forall FV(\tau_1), FV(\tau_2), t.\ \kappa_2 \wedge \theta_2 \leq t \models \kappa_1 \wedge \theta_1 \leq t$$

holds for a fresh variable t. The constrained type $\forall \bar{t}.\, \kappa \Rightarrow \theta$ is *well-formed* if and only if $\forall \emptyset.\ \mathit{true} \models \kappa$ holds. Subtyping and well-formedness have been proved decidable in [3].

$$\begin{array}{rcll} \tau & ::= & \forall \bar{t}.\, \kappa \Rightarrow \theta & \mathit{Type} \\ \theta & ::= & t \mid \theta \rightarrow \theta & \mathit{Monotype} \\ \kappa & ::= & \mathit{true} \mid \theta \leq \theta \mid \kappa \wedge \kappa & \mathit{Constraint} \end{array}$$

Figure 8: Constraint syntax

3.3.2 Instantiation

We have to define the pre-order on syntactic types built over ML$_\leq$ types. Contrarily to ML, ML$_\leq$ types include constraints. This makes them powerful enough to represent all the typing information present in syntactic types. Formally, we define a translation function from syntactic types back to algebraic, ML$_\leq$ types. The order on syntactic types is then simply the order on their translations. This translation function is defined by:

$$\langle t \rangle = \forall \emptyset.\ \mathit{true} \Rightarrow t \qquad \frac{\langle \tau \rangle = \forall \bar{t}.\, \kappa \Rightarrow \theta}{\langle \lambda t.\tau \rangle = \forall \bar{t}, t.\, \kappa \Rightarrow\ t \rightarrow \theta}$$

$$\frac{\langle \tau_1 \rangle = \forall \bar{t}_1.\, \kappa_1 \Rightarrow \theta_1 \qquad \langle \tau_2 \rangle = \forall \bar{t}_2.\, \kappa_2 \Rightarrow \theta_2}{\langle \tau_1\ \tau_2 \rangle = \forall \bar{t}_1, \bar{t}_2, v.\, \kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq (\theta_2 \rightarrow v) \Rightarrow v}$$

Theorem 4 (ML-Sub). *ML$_\leq$ is a type algebra.*

[4]We omit here type constructors for clarity purposes, but they do not raise any difficulties.

3.4 Multi-methods

Multi-methods are less popular than mono-methods used in single-dispatch languages (C++ [23], Java [16], Ocaml [22], ...). They have nevertheless been studied and used in several programming languages (CLOS [2, 15], Dylan [13], Cecil [6, 7], ...). However, their type-checking with polymorphic types, in a decidable and modular way that preserves type inference on ML expressions is still an issue.

We present here multi-methods as a particular case of open generic functions (Section 3.2), using the $\mathrm{ML}_{\leq}$ instance.

3.4.1 Definitions

The abstract syntax for a multi-method declarations is adapted from open generic functions:

$$\textbf{method } m : \tau_m$$

The language of patterns is defined by:

$$\pi \ ::= \ \_ \mid @C \mid \#C$$

The intent is that pattern _ matches any expression. Pattern $@C$ matches any instance of class C or of any of its subclasses. This means that an implementation with pattern $@C$ can also be used for sub-classes of C, which is usual in object-oriented languages. On the other hand, the pattern $\#C$ only matches instances of class C. This pattern is necessary to implement some polymorphic multi-methods, which we shall illustrate on an example in Section 3.4.3. Definitions 7 and 8 formalize this informal presentation of patterns.

3.4.2 Type-checking

We define multi-methods as an instance of open generic functions, and check that they meet their requirements.

Definition 7. *For any pattern π and type τ we define the* pattern constraint $\pi(\tau)$ *by:*

$$\begin{array}{rcl} \_\,(\forall \overline{t}.\, \kappa \Rightarrow \theta) & = & \mathit{true} \\ @C(\forall \overline{t}.\, \kappa \Rightarrow \theta) & = & \kappa \wedge \theta \leq C \\ \#C(\forall \overline{t}.\, \kappa \Rightarrow \theta) & = & \kappa \wedge \theta = C \end{array}$$

Definition 8. *The relation $\tau \in \pi$ holds if and only if $\mathit{true} \models \pi(\tau)$ holds.*

Definition 9. *The restriction*

$$\mathsf{restrict}(\forall \overline{t}.\, \kappa \Rightarrow \theta_1 \to \ldots \to \theta_n \to \theta, \pi_1 \ldots \pi_n)$$

is equal to

$$\forall \overline{t} t_1 \ldots t_n .\, \kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \Rightarrow t_1 \to \ldots \to t_n \to \theta$$

and $\mathsf{restrict}(\tau, \pi_1 \ldots \pi_n)$ is the error type if τ is not a functional type of textual arity n.

Note that if $\mathsf{restrict}(\tau, \overline{\pi}, i)$s not well-formed it is equal to $\mathbb{E}$ by the identification introduced in Section 3.3.

Lemma 6 (Restriction).

$$\Downarrow \frac{\tau_i \in \pi_i \quad (i = 1..n)}{\mathsf{restrict}(\tau_m, \overline{\pi})\ \tau_1 \ldots \tau_n = \tau_m\ \tau_1 \ldots \tau_n}$$

Theorem 3 (Generic functions) thus ensures type soundness for programs with valid and covered multi-methods.

3.4.3 Examples

This section illustrates the implementation and typing aspects of multi-methods. Firstly, we consider the definition of a generic method `equals` for structural equality of values. We assume given a function `identical` for testing pointer equality, to handle the general case, and a record type `Point` with a component x.

$$\begin{array}{l} \textbf{method}\ \mathtt{equals}\ :\ \forall t.\ t \to t \to \mathtt{bool} \\ \textbf{implementation}\ \mathtt{equals}\ \_\ \_ = \mathtt{identical} \\ \textbf{implementation}\ \mathtt{equals}\ @\mathtt{Point}\ @\mathtt{Point} = \\ \quad \lambda pq.\ \mathtt{equals}\ p.x\ q.x \end{array}$$

This last implementation is valid since it has inferred type `Point` $\to$ `Point` $\to$ `bool` and calculation in the $\mathrm{ML}_{\leq}$ type algebra shows that:

$$\begin{array}{l} \mathsf{restrict}(\forall t.\ t \to t \to \mathtt{bool}, @\mathtt{Point}\ @\mathtt{Point}) \\ = \forall t \overrightarrow{t_i}.\ \overrightarrow{t_i} \leq t \wedge \overrightarrow{t_i} \leq \mathtt{Point} \Rightarrow t_1 \to t_2 \to \mathtt{bool} \\ = \forall \overrightarrow{t_i}.\ \overrightarrow{t_i} \leq \mathtt{Point} \Rightarrow t_1 \to t_2 \to \mathtt{bool} \\ = \mathtt{Point} \to \mathtt{Point} \to \mathtt{bool} \end{array}$$

We can now implement the `opp` example of Section 3.1 in the concrete case of multi-methods with:

$$\begin{array}{l} \textbf{method}\ \mathtt{opp}\ :\ \forall t.\, t \leq \mathtt{num} \Rightarrow t \to t \\ \textbf{implementation}\ \mathtt{opp}\ \#\mathtt{float} = \mathtt{opp\_float} \\ \textbf{implementation}\ \mathtt{opp}\ \#\mathtt{int} = \mathtt{opp\_int} \end{array}$$

This illustrates that # patterns are necessary to implement methods with precise types like `opp`: if we try to use pattern `@float` we find:

$$\mathsf{restrict}(\forall t.\, t \leq \mathtt{num} \Rightarrow t \to t, @\mathtt{float}) = \forall t.\, t \leq \mathtt{float} \Rightarrow t \to t$$

It would thus be invalid to use pattern `@float` with a function that returns a `float`. That would be unsound indeed, since this implementation could be called with a strict subtype (say `rat`) of `float`, in which case `opp`'s type requires that the result be type `rat`. On the other hand,

$$\mathsf{restrict}(\forall t.\, t \leq \mathtt{num} \Rightarrow t \to t, \#\mathtt{float}) = \mathtt{float} \to \mathtt{float}$$

The reason why the implementation with pattern `#float` is sound is that this pattern does not match the strict subtypes of `float`. The coverage test will therefore ensure that an implementation exists for any such subtype, and each of these implementations will be required to return a result of the same type.

The # patterns are useful outside base type and operators. Consider the case of a generic container hierarchy. Container classes `List`, `Vector`, etc, derive from an abstract `Container` class. It is natural to define functional operations on containers that return the same kind of container as the argument. These can only be implemented using # patterns:

$$\begin{array}{l} \textbf{method}\ \mathtt{filter}\ :\ \forall t, c \leq \mathtt{Container}[t].\ (c, t \to \mathtt{bool}) \to c \\ \textbf{implementation}\ \mathtt{filter}\ \#\mathtt{List}\ \_ = \ldots\textbf{new}\ \mathtt{List}\ldots \\ \textbf{implementation}\ \mathtt{filter}\ \#\mathtt{Vector}\ \_ = \ldots\textbf{new}\ \mathtt{Vector}\ldots \end{array}$$

3.4.4 Semantics

The semantics is implied by the matching relation, as defined in Section 3.1. We can now follow up on the discussion about the typedness of this semantics. With the patterns described in this section, only the head type constructor is needed to specify reductions. In an object-oriented language, this tagging information is already commonly present at runtime. It corresponds to the `new_C` data constructor of Section 3.2. Multi-methods can thus be implemented without runtime typing.

Furthermore, it is not fundamental to give an eager semantics. The patterns presented in this section only require the head type constructor to reduce. Object fields could thus be computed lazily. This is much similar to the semantics of pattern matching in lazy languages, which would make multi-methods well-behaved in lazy languages.

Algorithms for efficient reduction of multi-methods operators (that is, implementation of multiple-dispatch) can be found in [10, 8, 12].

The multi-methods in this section can form the basis of a realistic programming language with a sound and tractable type system. An explicitly typed version of this language has a preliminary implementation as a partially bootstrapped compiler. The documentation and the compiler are available at `http://cristal.inria.fr/~bonniot/nice`. The results of this paper, and particularly Section 2.1, form the theoretical basis for the extension of the language with type inference.

4. RELATED WORK

The instantiation of our algebraic type system with the type language of $\mathrm{ML}_{\leq}$ in Section 3.3 together with the multi-method extension in Section 3.4 produces a system similar to the whole of $\mathrm{ML}_{\leq}$ ([3, 4]). These presentations used a highly non-standard operational semantics and soundness proofs through the use of a typed abstract machine, which made the presentation and the proof of the system tedious and hard to understand. This paper uses a more conventional approach, which consists in describing an untyped language and untyped small-step semantics, and proving soundness through subject-reduction and progress lemmas. Another major achievement of this work is to separate the object-oriented part from the core language, both in presentation and in the proofs. Since the latter part is already well understood, this makes the study of the theoretical aspects of multi-methods much easier.

Unlike $\mathrm{ML}_{\leq}$, we do not consider nested methods, that is declaration of methods inside other methods. The need for these is not clear from practice, especially as nested methods do not fit well with open multi-methods. Indeed, the addition of new classes usually needs to define new implementations for existing multi-methods. This is not possible if nested methods are allowed, since this would mean textually adding the cases in the same scope as for the nested method definition. The omission of nested multi-methods also makes the system much simpler. Of course, lambda-expressions can be nested inside methods and other lambda-expressions.

We formalize the type-checking of multi-methods, while [3] tackled formally only methods with one argument, and described how to extend the theory to methods with multiple arguments only informally. This is unfortunate since multi-methods were a key feature in this work.

We also remove the need to annotate lambda-expressions with their domain, and show that their type can be inferred. Thus, only top-level declarations of multi-methods require type annotations. We believe that these top-level annotations are acceptable in a programming language, and that they cannot be avoided if modular type-checking is needed. Indeed, implementations of methods might not be known when type-checking of code making use of these methods is performed.

$\mathrm{ML}_{\leq}$ had no notion of $\#C$ pattern. This feature is especially useful with polymorphic multi-methods: it makes it possible to implement methods of type $\forall t.\ t \to t$ or $\forall t. t \leq C \Rightarrow\ t \to t$ that are not the identity function but that actually return a newly constructed object. Useful concrete examples include the `clone` method, the `union` method on a hierarchy of collections with the precise type $\forall t. t \leq \mathtt{Collection} \Rightarrow\ (t,t) \to t$, etc.

The algebraic type system of Section 2 can be compared with $\mathrm{HM}(X)$ [21, 24]: both provide a generic framework to define new type systems, while factorizing the typing rules, type inference and soundness proof. $\mathrm{HM}(X)$ did so by taking a constraint system X as parameter. The syntax of types was fixed and exposed in the typing rules. The original presentation of $\mathrm{HM}(X)$ [21] only enjoyed complete type inference under unduly strong hypotheses. This has been relaxed in a newer technical report [24]. Our algebraic type system enjoys a higher level of abstraction than $\mathrm{HM}(X)$ since we do not expose the structure of types, which allows us to prove soundness at a more concise way. In particular we do not impose that the type system used be constraint based. Therefore, our framework is potentially applicable in more situations. Indeed, we proved in a previous draft that the revised version of $\mathrm{HM}(X)$ is equivalent to an instance of the algebraic type system with constrained types where constraint are extensible and contain at least subtyping inequalities. Finally, $\mathrm{HM}(X)$ has so far remained in the type-theoretic ground; it has not been used with advanced language features. They did not consider constants or operators in the expression language. We showed how our system can be used to model multi-methods using operators.

$\mathrm{HM}(X)$ uses the usual non-deterministic presentation of typing rules as a logical-oriented specification of typing, and proves that their type inference rules are sound and complete. We do not provide these two set of rules, but make use of our level of abstraction to present typing as a simple recursive function that returns the principal type of any well-typed expression, or the error type. While the logical-based presentation is certainly useful, we believe it is equally important to present type systems in a more compact way and with simple typing rules.

Cecil has a polymorphic type system for multi-methods [19]. However, this system is not known to be decidable. Chambers and Leavens [9] proposed decidable type-checking rules and algorithms for multi-methods in Cecil with only monomorphic types. Their system was more complex but also more powerful than ours in that it allowed the separation of inheritance from subtyping. Their notion of *signature* corresponds to our method declaration, with the difference that several signatures were allowed for the same generic function. This partially made up for the absence of parametric polymorphism. Each method implementation needed type annotations, and in particular a declared return type. This put a burden on the programmer, even if it can be argued that their syntactic sugar for methods partially relieved it by declaring simultaneously a signature and an implementation. In this perspective, our function restrict infers the type of method implementations. Furthermore, their "client-side type-checking" only checked the implementation against this declared return type. The declared return type was only compared to the signatures return types during the conformance test. However, this conformance check could only be performed with a global knowledge of the program, like the other parts of the "implementation-side type-checking". This made separate compilation impossible. This amounts to the fact that their predicate `ComputeIsConforming` was defined with a quantification on all signatures and concrete classes. We illustrate this modularity issue by a concrete example with two modules `M1` and `M2`. They separately type-checked, but they could not be imported together because they failed the conformance check when taken together. Furthermore, this situation cannot be resolved in any importing module:

```
module M1 {
  signature f(float) : int;
  implementation f(x@:float) : int = 0;
}
module M2 {
  signature f(int) : float;
  implementation f(x@:int) : float = 0.5;
}
module Main {
  import M1;
  import M2;
}
```

Module `Main` fails the conformance check with signature `f(float) : int` and class `int`: the most precise implementation is `f(x@:int) : float` which does not conform to return type `int`. The situation cannot be solved without modifying the pre-existing module `M2`, since that implementation for this problematic case is already as precise as can be. This system did therefore fail to achieve reasonable modularity. On the opposite, our type-checking rule checks modularly the inferred type of an implementation against the corresponding method declaration. This is made possible in particular by the presence of a *single* polymorphic signature, instead of several monomorphic ones possibly spread in different modules.

Millstein and Chambers [20] proposed a decidable type-system for monomorphic multi-methods. They studied a class-less language with modules, and proved type-soundness. Additionally, they studied several restrictions to the ability to define new objects and implementations that allowed coverage and non-ambiguity to be performed modularly. Their client-side check differed from [9], so that it could be performed modularly. Thus, they provided several interesting fully modular type-checking strategies for multi-methods. We follow their "global" approach to type-checking modules, which gives the full power of multi-methods; we also argue in Section 3.2 that the necessity to perform coverage and ambiguity checks when linking modules together does not break modularity especially if these test are performed aggressively and not only when the full program is known. Additionally, it should be easy to apply their other strategies in our presentation since we have the same notion of coverage test. On the other hand, they only allowed multi-methods to have a single signature, which in conjunction with monomorphism, results in a limited expressive power. For instance, precise types couldn't be given to numerical operators, and neither could multi-methods returning "self-type" be defined. The language didn't support the dynamic allocation of new objects, but an extension was sketched and proposed as future work. It also lacked function closures, and did not investigate type inference. Another proposed extension was the $@=$ pattern, but its usage was restricted to implementations where all patterns had this form. Our # pattern has the same semantics, but it can be mixed arbitrarily with other patterns. This full generality is useful, for instance in the `filter` example of Section 3.4.3.

The $\lambda\&$ calculus [5] proposed an extension of the lambda-calculus with functions that dispatch on their arguments runtime types. They argued that considering methods as first-class elements rather than object components significantly simplifies the theory over record based models that require recursive types. However, this presentation was not aimed at modeling directly programming languages: they did not allow programmer's definable base types; the use of the & operator to build methods does not fit directly the incremental definition of method implementations.

5. CONCLUSION AND FUTURE WORK

We made two main contributions. Firstly, we proposed the notion of a type algebra to parameterize a generic type system by the language of types. Secondly, we used this framework to solve a long-standing issue by presenting a decidable type system for languages *à la* ML with type inference and polymorphic multi-methods. Furthermore, this presentation demonstrated our claim that the algebraic type system can be extended modularly, that is without proving a new global soundness theorem. Indeed, we used two forms of extension: the instantiation by a type language suited to a particular domain, and the extension of the expression language using constants.

Future work includes proving that other existing or new type languages are type algebras. This will enable one to consider other language features besides multi-methods, and thus benefit from the potentially easy integration of orthogonal features made possible by our framework. For instance, it would be interesting to describe a type algebra with recursive predicates on types in the style of type classes [25] in Haskell; we are currently studying richer constraint domains to give more precise types to certain multi-methods. The fact that patterns are kept abstract in the generic methods can be used to handle more general patterns like those in [14]. It would also be instructive to explore how our framework can be extended to model language features that require different operational semantics, in particular references. A bigger challenge would be to parameterize the system not only in the type algebra, but also in the actual semantics used.

ACKNOWLEDGMENTS

My gratitude goes to Didier Rémy and François Pottier for their inspired comments and suggestions on this work. I also thank the anonymous reviewers for their useful remarks on earlier versions of this article.

6. REFERENCES

[1] M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Also appeared as SRC Research Report 120. Preliminary version appeared in the Proceedings of the ACM SigPlan Workshop on ML and its Applications, June 1992.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification x3j13. *SIGPLAN Notices*, 23(Special Issue), September 1998.

[3] F. Bourdoncle and S. 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, Mar. 1996.

[4] F. Bourdoncle and S. 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, Jan. 1997. ACM.

[5] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, 1992. Extended and revised version in *Information and Computation* 117(1):115-135, 1995.

[6] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[7] C. Chambers. The Cecil language: Specification and rationale, version 2.1. Technical report, Department of Computer Science and Engineering University of Washington, Box 352350, Seattle, Washington 98195-2350 USA, March 1997.

[8] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 238–255, Denver, CO, November 1999. ACM.

[9] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *ACM Transactions on Programming Languages (TOPLAS)*, volume 17(9). ACM, November 1995.

[10] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *ECOOP '94, European Conference on Object-Oriented Programming, Bologna, Italy*, volume 821 of *Lecture Notes in Computer Science*, pages 408–431, New York, N.Y., July 1994. Springer-Verlag.

[11] W. R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, N.Y., 1991.

[12] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-dispatch in the Java virtual machine: Design and implementation. In *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, USA, January 2001.

[13] Eastern Research Apple Computer and Technology. Dylan, an object-oriented dynamic language. Apple Computer, Inc., April 1992.

[14] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. Technical report, Seattle, Washington, 1998.

[15] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):28–38, September 1991.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sunsoft Java Series. Addison Wesley Developers Press, second edition, 2000.

[17] U. Holzle. Integrating independently-developed components in object-oriented languages. In *ECOOP '93*, pages 36–56, 1993.

[18] X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[19] V. Litvinov. Constraint-based polymorphism in Cecil. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Aplications*, pages 388–411, Vancouver, Canada, October 1998.

[20] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303, Lisbon, Portugal, June 1999. Springer Verlag.

[21] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[22] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[24] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

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

APPENDIX

A. PROOFS

A.1 Algebraic type system

PROOF OF LEMMA 1 (SUBSTITUTION). The proof is by induction on e.

$\boldsymbol{e \equiv x}$

$$(\Gamma \vdash x\,[x \leftarrow e']) = (\Gamma \vdash e')$$
$$\big(\Gamma, x : (\Gamma \vdash e') \vdash x\big) = (\Gamma \vdash e')$$

$\boldsymbol{e \equiv x'}$

$$(\Gamma \vdash x'\,[x \leftarrow e']) = (\Gamma \vdash x')$$
$$\big(\Gamma, x : (\Gamma \vdash e') \vdash x'\big) = (\Gamma \vdash x')$$

$\boldsymbol{e \equiv c}$ Same as $e \equiv x'$

$\boldsymbol{e \equiv \lambda x_1 . e_1}$

$$\begin{array}{lll}
 & (\Gamma \vdash (\lambda x_1.e_1)\,[x \leftarrow e']) & \\
\equiv & (\Gamma \vdash \lambda x_1.e_1\,[x \leftarrow e']) & (x_1 \notin FV(e')) \\
= & \lambda t_1.\,(\Gamma, x_1 : t_1 \vdash e_1\,[x \leftarrow e']) & (\textsc{Lam}) \\
= & \lambda t_1.\big(\Gamma, x_1 : t_1, x : (\Gamma, x_1 : t_1 \vdash e') \vdash e_1\big) & (\text{Ind. Hyp.}) \\
= & \lambda t_1.\big(\Gamma, x_1 : t_1, x : (\Gamma \vdash e') \vdash e_1\big) & (x_1 \notin FV(e')) \\
= & \big(\Gamma, x : (\Gamma \vdash e') \vdash \lambda x_1.e_1\big) & (\textsc{Lam})
\end{array}$$

$\boldsymbol{e \equiv e_1\ e_2}$

$$\begin{array}{lll}
 & (\Gamma \vdash (e_1\ e_2)\,[x \leftarrow e']) & \\
\equiv & (\Gamma \vdash e_1\,[x \leftarrow e']\ e_2\,[x \leftarrow e']) & \\
= & (\Gamma \vdash e_1\,[x \leftarrow e'])\ (\Gamma \vdash e_1\,[x \leftarrow e']) & (\textsc{App}) \\
= & \big(\Gamma, x : (\Gamma \vdash e') \vdash e_1\big)\ \big(\Gamma, x : (\Gamma \vdash e') \vdash e_2\big) & (\text{I.H.}) \\
= & \big(\Gamma, x : (\Gamma \vdash e') \vdash e_1\ e_2\big) & (\textsc{App})
\end{array}$$

$\boldsymbol{e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2}$

$$\begin{array}{lll}
 & (\Gamma \vdash (\text{let } x_1 = e_1 \text{ in } e_2)\,[x \leftarrow e']) & \\
\equiv & (\Gamma \vdash \text{let } x_1 = e_1\,[x \leftarrow e'] \text{ in } e_2\,[x \leftarrow e']) & \\
= & \big(\Gamma, x_1 : (\Gamma \vdash e_1\,[x \leftarrow e']) \vdash e_2\,[x \leftarrow e']\big) & (\textsc{Let}) \\
= & \big(\Gamma, x_1 : (\Gamma, x : (\Gamma \vdash e') \vdash e_1), x : (\Gamma \vdash e') \vdash e_2\big) & \\
 & \quad (\text{I.H.}, x_1 \notin FV(e')) & \\
= & \big(\Gamma, x : (\Gamma \vdash e') \vdash \text{let } x_1 = e_1 \text{ in } e_2\big) & (\textsc{Let})
\end{array}$$

□

PROOF OF LEMMA 2 (CONTEXT). The proof is by induction on e.

$\boldsymbol{e \equiv x}$

$$(\Gamma, x : \tau \vdash x) = \tau$$
$$(\Gamma, x : t \vdash x)\ [t \leftarrow \tau] = t\,[t \leftarrow \tau] = \tau$$

$\boldsymbol{e \equiv x'}$

$$(\Gamma, x : \tau \vdash x') = (\Gamma \vdash x')$$
$$(\Gamma, x : t \vdash x')\,[t \leftarrow \tau] = (\Gamma \vdash x')\,[t \leftarrow \tau] = (\Gamma \vdash x')$$

$\boldsymbol{e \equiv c}$ Same as $e \equiv x'$

$\boldsymbol{e \equiv \lambda x_1 . e_1}$

$$\begin{array}{lll}
 & (\Gamma, x : \tau \vdash \lambda x_1.e_1) & \\
= & \lambda t_1.\,(\Gamma, x : \tau, x_1 : t_1 \vdash e_1) & (\textsc{Lam}, \text{ with } t_1 \notin FV(\tau)) \\
= & \lambda t_1.\,(\Gamma, x_1 : t_1, x : t \vdash e_1)\ [t \leftarrow \tau] & (\text{I.H.}) \\
= & \big(\lambda t_1.\,(\Gamma, x_1 : t_1, x : t \vdash e_1)\big)\,[t \leftarrow \tau] & (t_1 \notin FV(\tau)) \\
= & (\Gamma, x : t \vdash \lambda x_1.e_1)\ [t \leftarrow \tau] & (\textsc{Lam})
\end{array}$$

$\boldsymbol{e \equiv e_1\ e_2}$

$$\begin{array}{lll}
 & (\Gamma, x : \tau \vdash e_1\ e_2) & \\
= & (\Gamma, x : \tau \vdash e_1)\ (\Gamma, x : \tau \vdash e_1) & (\textsc{App}) \\
= & (\Gamma, x : t \vdash e_1)\ [t \leftarrow \tau]\ (\Gamma, x : t \vdash e_2)\ [t \leftarrow \tau] & (\text{I.H.}) \\
\equiv & \big((\Gamma, x : t \vdash e_1)\ (\Gamma, x : t \vdash e_2)\big)\,[t \leftarrow \tau] & \\
= & (\Gamma, x : t \vdash e_1\ e_2)\ [t \leftarrow \tau] & (\textsc{App})
\end{array}$$

$\boldsymbol{e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2}$ Let Γ_τ and Γ_t denote $\Gamma, x : \tau$ and $\Gamma, x : t$.

$$\begin{array}{lll}
 & (\Gamma_\tau \vdash \text{let } x_1 = e_1 \text{ in } e_2) & \\
= & \big(\Gamma_\tau, x_1 : (\Gamma_\tau \vdash e_1) \vdash e_2\big) & (\textsc{Let}) \\
= & (\Gamma_\tau, x_1 : t_1 \vdash e_2)\ [t_1 \leftarrow (\Gamma_\tau \vdash e_1)] & (\text{I.H.}) \\
= & (\Gamma_t, x_1 : t_1 \vdash e_2)\ [t \leftarrow \tau]\,[t_1 \leftarrow (\Gamma_t \vdash e_1)\ [t \leftarrow \tau]] & (\text{I.H.}) \\
\equiv & (\Gamma_t, x_1 : t_1 \vdash e_2)\ [t_1 \leftarrow (\Gamma_t \vdash e_1)]\,[t \leftarrow \tau] & (*) \\
= & \big(\Gamma_t, x_1 : (\Gamma_t \vdash e_1) \vdash e_2\big)\,[t \leftarrow \tau] & (\text{I.H.}) \\
= & (\Gamma_t \vdash \text{let } x_1 = e_1 \text{ in } e_2)\ [t \leftarrow \tau] & (\textsc{Let})
\end{array}$$

The $(*)$ step is a usual property of substitution.

□

We prove a generalization of the Covariance requirement to arbitrary substitutions:

LEMMA 7 (COVARIANCE).

$$\frac{\tau' \leq \tau}{\tau_0\,[t \leftarrow \tau'] \leq \tau_0\,[t \leftarrow \tau]}$$

PROOF OF LEMMA 7 (COVARIANCE). The proof is by induction on τ_0.

$\boldsymbol{\tau_0 \equiv t}$ hypothesis

$\boldsymbol{\tau_0 \equiv t' \neq t}$ **or** $\boldsymbol{\tau_0 \equiv T}$ $\tau_0\,[t \leftarrow \tau'] = \tau_0\,[t \leftarrow \tau] = \tau_0$

The other cases are application of the the Covariance requirement to the induction hypothesis. □

PROOF OF LEMMA 3 (SUBJECT REDUCTION).

Op requirement 3 (CONSTANTS)

$\boldsymbol{\beta}$

$$\begin{array}{lll}
 & (\emptyset \vdash (\lambda x.e)\ e') & \\
= & (\emptyset \vdash \lambda x.e)\ (\emptyset \vdash e') & (\textsc{App}) \\
= & \lambda t.\,(x : t \vdash e)\ (\emptyset \vdash e') & (\textsc{Lam}) \\
\geq & (x : t \vdash e)\ [t \leftarrow (\emptyset \vdash e')] & (\text{Reduction}) \\
= & \big(x : (\emptyset \vdash e') \vdash e\big) & (\text{lemma } 2) \\
= & (\emptyset \vdash e\,[x \leftarrow e']) & (\text{lemma } 1)
\end{array}$$

$\boldsymbol{\beta}$**-Let**

$$\begin{array}{lll}
 & (\emptyset \vdash \text{let } x_1 = e_1 \text{ in } e_2) & \\
= & \big(x_1 : (\emptyset \vdash e_1) \vdash e_2\big) & (\textsc{Let}) \\
= & (\emptyset \vdash e_2\,[x_1 \leftarrow e_1]) & (\text{lemma } 1)
\end{array}$$

Ctxt $e \equiv \mathcal{E}[e_1]$ with $e_1 \longrightarrow e_1'$.

So by induction hypothesis $(\emptyset \vdash e_1') \leq (\emptyset \vdash e_1)$

$\boldsymbol{\mathcal{E} \equiv []\ e_2}$

$$\begin{array}{lll}
 & (\emptyset \vdash e_1'\ e_2) & \\
= & (\emptyset \vdash e_1')\ (\emptyset \vdash e_2) & (\textsc{App}) \\
\leq & (\emptyset \vdash e_1)\ (\emptyset \vdash e_2) & (\text{Covariance}) \\
= & (\emptyset \vdash e_1\ e_2) & (\textsc{App})
\end{array}$$

$\mathcal{E} \equiv e_2$ [] *idem*

$\mathcal{E} \equiv$ **let** $x_1 = []$ **in** e_2

$$\begin{array}{lll} & (\emptyset \vdash \text{let } x_1 = e_1' \text{ in } e_2) & \\ = & (x_1 : (\emptyset \vdash e_1') \vdash e_2) & (\textsc{Let}) \\ = & (x_1 : t_1 \vdash e_2)\ [t_1 \leftarrow (\emptyset \vdash e_1')] & (\text{lemma } 2) \\ \leq & (x_1 : t_1 \vdash e_2)\ [t_1 \leftarrow (\emptyset \vdash e_1)] & (\text{lemma } 7) \\ = & (x_1 : (\emptyset \vdash e_1) \vdash e_2) & (\text{lemma } 2) \\ = & (\emptyset \vdash \text{let } x_1 = e_1 \text{ in } e_2) & (\textsc{Let}) \end{array}$$

□

PROOF OF LEMMA 4 (PROGRESS). The proof is by induction on e.

$e \equiv x$ Impossible because x is not well-typed.

$e \equiv c$ Then by definition e is a value

$e \equiv \lambda x.e'$ is a value

$e \equiv$ **let** $x_1 = e_1$ **in** e_2 reduces by β-LET

$e \equiv e_1\ e_2$ By APP, e_1 and e_2 are well-typed. By induction hypothesis, if $e_2 \longrightarrow e_2'$ then by CTXT, $e \longrightarrow e_1\ e_2'$. Otherwise e_2 is a value v_2 and there are four cases:

$e_1 \longrightarrow e_1'$ Then by CTXT $e \longrightarrow e_1'\ v_2$

The other cases correspond to e_1 being a value:

$e_1 \equiv \lambda x_1.e_1'$ Then e reduces by β

$e_1 \equiv C\ v_1' \dots v_n'$ By hypothesis, $e \equiv C\ v_1' \dots v_n'\ v_2$ is well-typed. So by requirement 3 (CONSTANTS), $n+1 \leq \mathsf{arity}(C)$. Hence e is a value.

$e_1 \equiv f\ v_1' \dots v_p'$ with $p < \mathsf{arity}(f)$.

If $p+1 = \mathsf{arity}(f)$, then since $e \equiv f\ v_1' \dots v_{n-1}'\ v_2$, requirement 3 (CONSTANTS) imposes that $e \longrightarrow e'$. Else $p+1 < \mathsf{arity}(f)$, so e is a value.

□

PROOF OF THEOREM 1 (SOUNDNESS). Let's suppose there exists a finite reduction of e, that is $e \longrightarrow^* e'$ and $e' \not\longrightarrow$. Lemma 3 (SUBJECT REDUCTION) shows that $\emptyset \vdash e' : \tau' \leq \tau$. Since $e' \not\longrightarrow$, lemma 4 (PROGRESS) shows that e' is a value. □

A.2 Hindley/Milner

PROOF OF THEOREM 2 (ML). Let us first prove requirement 1 (COVARIANCE). We therefore assume $\tau_1 \leq \tau_2$ with $\langle \tau_i \rangle = \forall \overline{\alpha}_i.\theta_i, C_i$ for $i = 1, 2$.

In the case of type application, by definition, $\langle \tau_i\ \tau_0 \rangle = \forall \overline{\alpha}_i \overline{\alpha}_0 \alpha.\alpha, C_i \cup C_0 \cup \{\theta_i = \theta_0 \to \alpha\}$. For all ρ_2 such that $\rho_2(C_2 \cup C_0 \cup \{\theta_2 = \theta_0 \to \alpha\})$, $\rho_2(C_2)$. Therefore, by the hypothesis $\tau_1 \leq \tau_2$, there exists ρ_1 equal to ρ_2 outside $\overline{\alpha}_1$ such that $\rho_1(C_1)$ and $\rho_1(\theta_1) = \rho_2(\theta_2)$. By construction $\rho_1(C_1 \cup C_0)$. Furthermore

$$\begin{array}{lll} & \rho_1(\theta_0 \to \alpha) & \\ = & \rho_1(\theta_0) \to \rho_1'(\alpha) & \\ = & \rho_2(\theta_0) \to \rho_2(\alpha) & (\text{definition of } \rho_1) \\ = & \rho_2(\theta_0 \to \alpha) & \\ = & \rho_2(\theta_2) & (\text{hypothesis on } \rho_2) \\ = & \rho_1(\theta_1) & (\text{hypothesis } \tau_1 \leq \tau_2) \end{array}$$

Therefore $\tau_1\ \tau_0 \leq \tau_2\ \tau_0$.

The other case of type application is similar. For type abstraction: $\langle \lambda t.\tau_i \rangle = \forall \overline{\alpha}_i \alpha_t.\, \alpha_t \to \theta_i, C_i$. For all ρ_2 such that $\rho_2(C_2)$, by the hypothesis $\tau_1 \leq \tau_2$, there exists ρ_1 equal to ρ_2 outside $\overline{\alpha}_1$ such that $\rho_1(C_1)$ and $\rho_1(\theta_1) = \rho_2(\theta_2)$. Furthermore,

$$\begin{array}{lll} & \rho_1(\alpha \to \theta_1) & \\ = & \rho_1(\alpha) \to \rho_1(\theta_1) & \\ = & \rho_2(\alpha) \to \rho_2(\theta_2) & (\text{hypothesis } \tau_1 \leq \tau_2) \\ = & \rho_2(\alpha \to \theta_2) & \end{array}$$

Therefore $\lambda t.\tau_1 \leq \lambda t.\tau_2$.

We now prove requirement 2 (REDUCTION) by case on τ.

$\tau \equiv \lambda t_0.\tau_0$
By definition of the substitution, $(\lambda t.\tau_0)\,[t \leftarrow \tau'] \equiv \lambda t_0.(\tau_0\,[t \leftarrow \tau'])$. We suppose by induction that $\tau_0\,[t \leftarrow \tau'] \leq (\lambda t.\tau_0)\ \tau'$. Since requirement 1 (COVARIANCE) is already proved, we have therefore $\lambda t_0.(\tau_0\,[t \leftarrow \tau']) \leq \lambda t_0.(\lambda t.\tau_0)\ \tau'$. We conclude this case by noting that $\langle (\lambda t.\lambda t_0.\tau_0)\ \tau' \rangle = \langle \lambda t_0.((\lambda t.\tau_0)\ \tau') \rangle$.

$\tau \equiv \tau_1\ \tau_2$
Similarly, induction hypothesis and requirement 1 show $((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau') \geq (\tau_1\ \tau_2)\,[t \leftarrow \tau']$. Furthermore, $((\lambda t.\tau_1)\ \tau')\ ((\lambda t.\tau_2)\ \tau') = (\lambda t.\tau_1\ \tau_2)\ \tau'$, since the translation of the former only differ by a duplicate copy of the constraint in $\langle \tau' \rangle$ due to alpha-conversion.

$\tau \equiv t$

$$\begin{array}{ll} & \langle \tau\,[t \leftarrow \tau'] \rangle \\ = & \langle \tau' \rangle \\ = & \forall \overline{\alpha}'.\theta', C' \\ = & \forall \overline{\alpha}' \alpha_t.\alpha_t, C' \cup \{\alpha_t = \theta'\} \\ = & \langle (\lambda t.\tau)\ \tau' \rangle \end{array}$$

$\tau \equiv \mathbb{E}$ **or** σ **or** $t' \not\equiv t$
Let $\langle \tau \rangle$ be $\forall \overline{\alpha}.\theta, C$ and $\langle \tau' \rangle$ be $\forall \overline{\alpha}'.\theta', C'$.

$$\begin{array}{lll} & \langle \tau\,[t \leftarrow \tau'] \rangle & \\ = & \langle \tau \rangle & (\tau \text{ is a constant type}) \\ = & \forall \overline{\alpha}.\theta, C & \\ \leq & \forall \overline{\alpha}\overline{\alpha}' \alpha_t.\theta, C \cup C' \cup \{\alpha_t = \theta'\} & (\text{see below}) \\ = & \langle (\lambda t.\tau)\ \tau' \rangle & \end{array}$$

The above inequality is trivial: for all ρ_2 such that $\rho_2(C' \cup \{\alpha_t = \theta'\})$, we can take $\rho_1 = \rho_2$ since then $\rho_1(C)$ and $\rho_1(\theta) = \rho_2(\theta)$. Note that the reverse inequality does not hold as soon as C' is not trivial.

□

A.3 Generic functions

PROOF OF THEOREM 3 (GENERIC FUNCTIONS).

- $g\ v_1 \dots v_n$ is well-typed, so each $(\emptyset \vdash v_i)$ is a runtime type. So by definition 6 there exists an implementation $\overline{\pi} \Rightarrow e$ such that $(\emptyset \vdash v_1) \dots (\emptyset \vdash v_n) \in \overline{\pi}$. So by definition of reduction rules, $g\, v_1 \dots v_n \longrightarrow e\ v_1 \dots v_n$
- Let $I \equiv \overline{\pi} \Rightarrow e$ be an implementation that matches $(\emptyset \vdash v_1) \dots (\emptyset \vdash v_n)$, and τ_g the declared type of g.

$$\begin{array}{lll} & (\emptyset \vdash e\ v_1 \dots v_n) & \\ = & (\emptyset \vdash e)\ (\emptyset \vdash v_1) \dots (\emptyset \vdash v_n) & (\textsc{App}) \\ \leq & \mathsf{restrict}(\tau_g, \pi)\ (\emptyset \vdash v_1) \dots (\emptyset \vdash v_n) & (I \text{ is valid}) \\ \leq & \tau_g\ (\emptyset \vdash v_1) \dots (\emptyset \vdash v_n) & (\text{requirement } 4) \\ \equiv & (\emptyset \vdash g\ v_1 \dots v_n) & \end{array}$$

□

A.3.1 *Inference for open generic functions*

It is an open problem to infer types for generic functions. However, we claim here that it does not make sense to try to infer types for open generic functions. In short, the idea is that inference requires the knowledge of the implementation of the function. For open generic functions, only some implementations might be known in a module. The type inferred might then be too precise to capture the intent of the functions, which would either prevent proper implementation in client modules, or break modularity by requiring the imported module to be re-type-checked.

As a degenerate case, it is perfectly legal to declare an open generic function with no implementation at all. This makes sense if the function operates on a hierarchy rooted at an abstract class, with no known subclasses. That is to say that the module could only contain:

generic g

How can we possibly infer a type for g? If we choose $\forall t.t$, one will not be able to implement g in a client module.

A.4 ML$_{\leq}$

$$\textsc{Trans}\ \frac{\forall\overline{t}.\ \kappa_1 \models \kappa_2 \qquad \forall\overline{t}.\ \kappa_2 \models \kappa_3}{\forall\overline{t}.\ \kappa_1 \models \kappa_3} \qquad \textsc{Triv}\ \frac{\kappa' \subset \kappa}{\forall\overline{t}.\ \kappa \models \kappa'}$$

$$\textsc{VarIntro}\ \frac{\forall t \in \overline{t} \quad t\sigma \equiv t}{\forall\overline{t}.\ \kappa\sigma \models \kappa} \qquad \textsc{Ref}\ \frac{}{\forall\overline{t}.\ \kappa \models \kappa \wedge \theta \leq \theta}$$

$$\textsc{Trans}\ \frac{\theta \leq \theta' \in \kappa \qquad \theta' \leq \theta'' \in \kappa}{\forall\overline{t}.\ \kappa \models \kappa \wedge \theta \leq \theta''}$$

$$\rightarrow\textsc{Intro}\ \frac{\theta_1 \leq \theta_1' \in \kappa \qquad \theta_2' \leq \theta_2 \in \kappa}{\forall\overline{t}.\ \kappa \models \kappa \wedge\ \theta_1' \rightarrow \theta_2' \leq\ \theta_1 \rightarrow \theta_2}$$

$$\rightarrow\textsc{Elim}\ \frac{\theta_1' \rightarrow \theta_2' \leq\ \theta_1 \rightarrow \theta_2 \in \kappa}{\forall\overline{t}.\ \kappa \models \kappa \wedge \theta_1 \leq \theta_1' \wedge \theta_2' \leq \theta_2}$$

Figure 9: Axioms of constraint implication

PROOF OF THEOREM 4 (ML-SUB). Requirement Covariance is easily proved using the axioms of the constraint implication. The proof can be found in [3].

Let us prove that $(\lambda t.\tau)\ \tau' \geq \tau\,[t \leftarrow \tau']$. If t appears in τ, then $\langle\tau\,[t \leftarrow \tau']\rangle \equiv \forall\overline{t},\overline{t}'.\kappa\,[t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta\,[t \leftarrow \theta']$, else $\langle\tau\,[t \leftarrow \tau']\rangle \equiv \forall\overline{t}.\kappa \Rightarrow \theta$.

$$\begin{array}{cll} & \langle(\lambda t.\tau)\ \tau'\rangle & \\ = & \forall\overline{t},t,\overline{t}',v.\kappa \wedge \kappa' \wedge\ t \rightarrow \theta \leq\ \theta' \rightarrow v \Rightarrow v & \text{(Definition)} \\ = & \forall\overline{t},t,\overline{t}',v.\kappa \wedge \kappa' \wedge \theta' \leq t \wedge \theta \leq v \Rightarrow v & (\rightarrow\textsc{Elim}) \\ = & \forall\overline{t},t,\overline{t}'.\kappa \wedge \kappa' \wedge \theta' \leq t \Rightarrow \theta & \end{array}$$

By construction, t does not appear in θ' nor in κ', and it appears covariantly in θ (on the right of $\rightarrow$) and in κ (on the left of $\leq$). So for each possible occurrence of t is of the form $t \leq \theta''$, and transitivity with $\theta' \leq t$ implies that $\theta' \leq \theta''$. Thus, $\langle(\lambda t.\tau)\ \tau'\rangle\ \geq\ \forall\overline{t},\overline{t}'.\kappa\,[t \leftarrow \theta'] \wedge \kappa' \Rightarrow \theta\,[t \leftarrow \theta']$. Whether t appears in τ or not, this is a super-type of $\langle\tau\,[t \leftarrow \tau']\rangle$. □

A.5 Multi-methods

PROOF OF LEMMA 6 (RESTRICTION).
Let $\tau_m \equiv \forall\overline{t}.\kappa \Rightarrow \theta_1 \rightarrow \ldots \rightarrow \theta_n \rightarrow \theta$, and $\tau_i' \equiv \forall\overline{t}_i'.\kappa_i' \Rightarrow \theta_i'$.

By definition:

$$\mathsf{restrict}(\tau_m,\pi_i)\ \tau_i' = \forall\overline{t}\mathcal{T}_i t_i \overline{t}_i'.\ \begin{array}{c}\kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \\ \kappa_i' \wedge \theta_i' \leq t_i\end{array} \Rightarrow \theta$$

$$\tau_m\ \tau_1'\ \ldots\ \tau_n' = \forall\overline{t}\overline{t}_i'.\kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \Rightarrow \theta$$

The proof of $\tau_m\ \tau_1'\ \ldots\ \tau_n' \leq \mathsf{restrict}(\tau_m,\pi_i)\ \tau_i'$ is straightforward:

$$\begin{array}{lll} \forall t. & & \\ & \kappa \wedge t_i \leq \theta_i \wedge \pi_i(t_i) \wedge \kappa_i' \wedge \theta_i' \leq t_i \wedge \theta \leq t & \\ \models & \kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \wedge \theta \leq t & (\text{MT}\textsc{rans}) \end{array}$$

Conversely, $\mathsf{restrict}(\tau_m,\pi_i)\ \tau_i' \leq \tau_m\ \tau_1'\ \ldots\ \tau_n'$ because:

$$\begin{array}{lll} \forall t. & & \\ & \kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \wedge \theta \leq t & \\ \models & \kappa \wedge \theta_i' \leq \theta_i \wedge \kappa_i' \wedge \pi_i(\theta_i') \wedge \theta_i' \leq \theta_i' \wedge \theta \leq t & (\tau_i' \in \pi_i) \\ \models & \kappa \wedge t_i \leq \theta_i \wedge \kappa_i' \wedge \pi_i(t_i) \wedge \theta_i' \leq t_i \wedge \theta \leq t & \\ & (\textsc{VarIntro} : t_i \mapsto \theta_i') & \end{array}$$

□