A Type Checker for Annotated OCaml Abstract Syntax Trees,

or

An Effective Type System for OCaml

[Research paper, extended abstract]

Pierrick Couderc[1,2], Michel Mauny[1], Grégoire Henry[2], and Fabrice Le Fessant[3]

[1] LIX, École Polytechnique – U2IS, ENSTA ParisTech
Université Paris-Saclay
828 bd des Maréchaux, 91762 Palaiseau cedex France
[2] OCamlPro
6 Allée de la Croix Saint-Pierre, 91190 Gif-sur-Yvette
[3] Gallium, Inria Paris
75012, Paris, France

Abstract. Checking fully type-annotated abstract syntax trees resulting from type inference is useful for verifying the coherence of supposedly well-typed programs, and for chasing bugs in the implementation of the type inference process. But, more interestingly, a disciplined implementation of this checking mechanism may be read as an executable specification of the type system. We present such a type checking mechanism for the OCaml language, and the corresponding typing rules for a large subset of the language,

1 Introduction

The OCaml language [LD14] supports many features coming either from the functional, imperative or object worlds, like modules and applicative functors [XL95], generalized algebraic datatypes [GL11, PR06], mutable variables through records, or objects via structural polymorphism and row variables [DV97].

While all those features have been thoroughly studied, the language still lacks a formal definition of its type system. Actually, the OCaml implementation is its only available definition.

The OCaml compiler, after its parsing and typing phases, produces a completely type-annotated and self-contained abstract syntax tree that represents the program, simply called the *typed AST* (TAST, for short). This TAST can then be used and analyzed, for example as a typing proof, and thus checked without inferring again the type of the program. We devised and implemented a type checker for this TAST, relying on as few as possible of the compiler's internals. Our type checker verifies the coherence of the TAST, and can be seen as a formal and effective definition of the OCaml type system.

2 TASTs: typed-annotated abstract syntax trees

After the type inference of the program, the OCaml compiler produces a fully type-annotated abstract syntax tree, which contains at each node all the information that were inferred, as well as the environment used to perform the inference. Its

[3] This work is part of the first author's ongoing PhD project, and has been partially supported by *Association nationale de la recherche et de la technologie* (ANRT).

$e ::=$	*expressions*
$x \mid c \mid$ **function** $\overline{p \to e} \mid e_1\ e_2 \mid$ **let** $\overline{p = e}$ **in** $e \mid$ **let rec** $\overline{p = e}$ **in** e	*ML classical constructions*
$\mid$ **function** $\sim l{:}p \to e \mid e_1\ (\sim l{:}e_2)$	*labeled functions arguments*
$\mid$ **match** e **with** $\overline{p \to e}$	*pattern matching*
$\mid$ **try** e **with** $\overline{exn \to e}$	*exceptions catching*
$\mid e_1, .., e_n \mid K\ (\overline{e})$	*tuples and variant constructors*
$\mid \grave{}K\ e$	*polymorphic variant constructors*
$\mid \{\overline{l = e;}\} \mid e.l \mid e_1.l \leftarrow e_2$	*records*
$\mid e_1.(e_2) \mid e_1.(e_2) \leftarrow e_3$	*arrays*
$\mid$ **if** e_1 **then** e_2 **else** e_3	*conditional*
$\mid e_1$ **;** e_2	*sequence*
$\mid$ **for** $x = e_1$ **to**/**downto** e_2 **do** e_3 **done** $\mid$ **while** e_1 **do** e_2 **done**	*imperative loops*
$\mid$ **let module** $I = M$ **in** $e \mid$ (**module** M)	*first-class modules*
$\mid$ **lazy** e	*lazyness*
$\mid$ **assert** e	*assertions*
$\mid e^+$	
$e^+ ::=$	*expressions with extra information*
$(e : \tau_1 :> \tau_2) \mid (e : \tau)$	*coercions and constraints*
$\mid$ **fun** (**type** a) $\to$ e	*local abstract type introduction*
$\mid$ **let open** P **in** e	*local module opening*
$p ::=$	*patterns*
$x \mid c \mid p$ **as** x	*variable, constant and aliases pattern*
$\mid (p_0, .., p_n) \mid K\ (\overline{p}) \mid \grave{}K\ (\overline{p})$	*tuple, variant and polymorphic variant constructor pattern*
$\mid \{\ \overline{l = e;}\ \} \mid [\ \overline{p}\ ]$	*record and array pattern*
$\mid (\ p_1 \mid p_2\ )$	*or pattern*
$\mid$ **lazy** p	*lazy pattern*

Fig. 1. The core OCaml expressions and patterns language

lightweight syntax, given at figure 1, is almost the same as the syntax of OCaml, the only difference being that some of the constructions have been desugared. For instance, the classical abstraction construct **fun** $x \to e$ is fused with the "by case" abstraction **function** $\overline{p \to e}$. The language includes the classical ML constructs, as well as some others that are more peculiar such as functions with labelled arguments (**function** $\sim$l:p $\to$ e), imperative loops and lazyness.

Local opening of modules, coercions, type constraints (provided as user annotations) and local abstract types are extra informations attached to expressions, noted e^+ in figure 1. Instead of being represented with their own TAST node, they are registered as extra attributes of the expression they enclose in the source code: this is how OCaml compiler distinguishes between the nodes that represent effective computations and those that only refine the type of expressions when generating the TAST.

The type algebra is given in figure 2. Types are stratified into core types and signatures/module types, where the latter may contain the former.

Core types, in the OCaml implementation, are also annotated with a "level" that is used by the inference process for generalization [DR92] and detection of (existential) types escaping their scope in the context of pattern-matching over

$\tau ::= \text{'}a$	*free type variable*
$\mid (l : \tau) \to \tau$	*function with labeled argument*
$\mid \tau_0 * .. * \tau_n$	*tuple*
$\mid (\overline{\tau})\ \mathtt{t}$	*type constructor*
$\mid [(\rho)\ K_0\ \mathbf{of}\ \tau_0; ..;\ K_n\ \mathbf{of}\ \tau_n > K_i .. K_j\ ]$	*polymorphic variant*
$\mid \alpha$	*universally quantified variable*
$\mid \forall\overline{\alpha}.\tau$	*universally quantified type*
$\mid (\ \mathbf{module}\ \mathtt{P}\ \mathbf{with}\ \overline{\mathtt{t} = \tau})$	*first-class module pack*
$\rho ::= \text{'}a \mid \epsilon$	*row variable*

Fig. 2. The OCaml type algebra

GADTs, and first-class modules. Core types can either be type variables, functions whose argument can be labeled [JG01], tuples, type constructors that designate either datatypes or locally abstract types for the most common types. They can also be variant types, universally quantified types, and packs for first class modules. Since variant types are polymorphic, they contain a so-called "row variable" that can either be a type variable or the special empty type ϵ, to encode a closed variant type. Types schemes σ are actually completely implicit in the OCaml internal type algebra, but we make them explicit in the presentation in our typing rules.

Finally, the TAST can also contain parametric type definitions for (possibly generalized) algebraic datatypes, record types, type abbreviations (that is, type aliases introduced by the programmer) and extensible types, that generalize the exception type. These type definition nodes are checked during type inference and also contain their own type and environment.

As a last note, we will not consider object and classes in our language and formalism, and let them for future work.

3 From type checking to typing rules

As we said earlier, the TAST is self-contained: checking its consistency should be possible without using any form of inference. In order to check TASTs, we wrote a type checker reusing as few compiler's internals and algorithms as possible. There are multiple reasons to do so. First of all, we want to avoid reusing levels and check them: we consider that they are part of the type inference implementation and not of the semantics. Using and checking levels would force us to maintain them all along our type checking algorithm, taking the risk of simply reproducing the corresponding part of the type inference implementation. And reusing too much material from the type inference engine could result in a type checking engine that reproduces possible bugs of the implementation of type inference. Finally, we ended up using from the OCaml compiler, only its data structures such as the TAST, the type algebra, the typing environment, and some auxiliary functions that we knew were safe to use.

We implemented all the rules to type check the TAST. The current algorithm for the core OCaml language is given at 3. The rules may look quite verbose, but actually this is a choice to have a system that is as explicit as possible and readable as a type checking algorithm. Judgements have the form $C \vdash (e : \tau)$, where $C = (\Gamma, \Phi)$ is the typing context. Γ is a mapping of paths to types that can be value types, type constructors, module types, module type types or universally quantified types. They can be accessed via projection (i.e. $\Gamma.Value$, or $\Gamma.Module$, etc). The second entity Φ is a mapping from types to sets of types, used to represent ambivalences [GR13]. Under such assumptions, two types with the same ambivalences (noted $\tau_1 \simeq \tau_2$) can be seen as equivalent. They are necessary to type GADTs.

$$\textsc{Const}\ \frac{c \in domain(\tau)}{C \vdash (c : \tau)} \qquad \textsc{Var}\ \frac{C, \Sigma \vdash \tau \leq C.\Gamma.Values(x) \Rightarrow \theta}{C \vdash (x : \tau)}$$

$$\textsc{Abs}\ \frac{\begin{array}{c} C \vdash \tau \prec \tau_d \to \tau_{cd} \qquad \forall i.\ C, \Sigma \vdash (p_i : \tau_i) \Rightarrow (\overline{v_i : \tau_{v_i}}), \Phi_i \\ \forall i.\ C \vdash \tau_d \equiv \tau_i \qquad \forall i.\ C_i \leftarrow C \oplus_{\mathcal{V}} (\overline{v_i : \tau_{v_i}}), \Phi_i \qquad \forall i.\ C_i \vdash (e_i : \tau'_i) \qquad \forall i.\ C \vdash \tau'_i\ wf \qquad \forall i.\ C \vdash \tau_{cd} \equiv \tau'_i \end{array}}{C \vdash (\texttt{function}\ \overline{\mid\ p \to e}\ : \tau)}$$

$$\textsc{Abs-Label}\ \frac{\begin{array}{c} C \vdash \tau \prec l' : \tau_d \to \tau_{cd} \qquad l = l' \qquad C, \Sigma \vdash (p : \tau_{arg}) \Rightarrow (\overline{v : \tau_v}), \Phi' \\ C \vdash \tau_d \equiv \tau_{arg} \qquad C' \leftarrow C \oplus_{\mathcal{V}} (\overline{v : \tau_v}), \Phi' \qquad C \vdash (e : \tau_{res}) \qquad C \vdash \tau_{res}\ wf \qquad C \vdash \tau_{cd} \equiv \tau_{res} \end{array}}{C \vdash (\texttt{function} \sim l : p \to e : \tau)}$$

$$\textsc{App}\ \frac{C \vdash (e_1 : \tau_1) \qquad C \vdash (e_2 : \tau_2) \qquad C \vdash \tau_1 \prec \tau_d \to \tau_{cd} \qquad C \vdash \tau_2 \equiv \tau_d \qquad C \vdash \tau_{cd} \equiv \tau}{C \vdash (e_1\ e_2 : \tau)}$$

$$\textsc{Construct}\ \frac{\begin{array}{c} \forall i.\ C \vdash (e_i : \tau_i) \qquad (\tau_{arg_0}, \ldots, \tau_{arg_n}, \tau_{constr}) \leftarrow find\_constructor(C, T, \tau) \\ C, \Sigma \vdash \tau_0 \leq \tau_{arg_0} \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ C, \theta_{i-1} \vdash \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \qquad \tau_{inst} \leftarrow \theta_n(\tau_{constr}) \qquad C \vdash \tau_{inst} \equiv \tau \end{array}}{C \vdash (T(e_0, \ldots, e_n) : \tau)}$$

$$\textsc{Let}\ \frac{\begin{array}{c} \forall i.\ C, \Sigma \vdash (p_i : \sigma_i) \Rightarrow (\overline{v_i : \sigma_{v_i}}), \Phi_i \qquad \forall i.\ C \vdash (e_i : \tau_i) \qquad \forall i.\ C, \Sigma \vdash \tau_i \leq \sigma_i \Rightarrow \theta_i \\ \forall i.\ check\_gen(C, \sigma_i, e_i) \qquad C' \leftarrow C \oplus_{\mathcal{V}} (v_{00} : \sigma_{v_{00}} .. v_{nm} : \sigma_{v_{nm}}), \Phi_i \qquad C' \vdash (e' : \tau') \qquad C \vdash \tau'\ wf \qquad C \vdash \tau \equiv \tau' \end{array}}{C \vdash (\texttt{let}\ \overline{p = e}\ \texttt{in}\ e' : \tau)}$$

$$\textsc{Letrec}\ \frac{\begin{array}{c} \forall i.\ C, \Sigma \vdash (p_i : \sigma_i) \Rightarrow (\overline{v_i : \sigma_{v_i}}), \Phi_i \qquad \forall i.\ check\_gen(C, \sigma_i, e_i) \qquad C' \leftarrow C \oplus_{\mathcal{V}} (v_{00} : \sigma_{v_{00}} .. v_{nm} : \sigma_{v_{nm}}), \Phi_i \\ \forall i.\ C' \vdash (e_i : \tau_i) \qquad \forall i.\ C, \Sigma \vdash \tau_i \leq \sigma_i \Rightarrow \theta_i \qquad C' \vdash (e' : \tau') \qquad C \vdash \tau'\ wf \qquad C \vdash \tau \equiv \tau' \end{array}}{C \vdash (\texttt{let rec}\ \overline{p = e}\ \texttt{in}\ e' : \tau)}$$

Fig. 3. Type checking core-OCaml expressions

In each judgement, the notation $(e : \tau)$ represents the information provided by the TAST. The type τ needs then to be *matched* against the expected type for the expression, *modulo abbreviations and ambivalence*, using the operator $\prec$. The matching operation $\tau \prec \tau_p$ uses the expected type τ_p as a (type) pattern, checks the conformity of τ w.r.t. τ_p and binds the components of τ that have to be used by the checking process. For example, $C \vdash \tau \prec \tau_1 \to \tau_2$ checks that τ can be a function type modulo abbreviations and ambivalences, binds the meta-variables τ_1 and τ_2 to the corresponding parts of the expansion of τ, and puts them into the context of the rule.

We also introduce the equivalence operator $\equiv$, that checks that two types are equivalent, again modulo abbreviations and ambivalences. This equivalence is checked by induction on the structure of the type, expanding abbreviations when necessary. The only tricky case is with rigid type variables[4]: if one appears, we must check that the type against which it is compared also appears in Φ with the same ambivalence.

Patterns (figure 4) in OCaml introduce variables in the current context and generate ambivalences for GADTs. In fact, each construction that binds variables is actually a pattern: the **`let`** binding can deconstruct a value and bind its components, while **`match`** and **`function`** perform full-fledged matching on values and bind value components to variables. As a result, typing a pattern also returns a mapping from variables occurring in the pattern to types. We introduce the $\uplus$ operation that

[4] OCaml provides locally abstract types, which are implemented as rigid type variables.

$$\textsc{Pat-Const}\ \frac{c \in \mathit{domain}(\tau)}{\Gamma, \Phi, \mathcal{V} \vdash (c : \tau) \Rightarrow \mathcal{V}, \Phi} \qquad \textsc{Pat-Wildcard}\ \Gamma, \Phi, \mathcal{V} \vdash (\_ : \tau) \Rightarrow \mathcal{V}, \Phi \qquad \textsc{Pat-Var}\ \frac{v \notin \mathcal{V} \qquad \mathcal{V}' \leftarrow \mathcal{V} \oplus (v, \tau)}{\Gamma, \Phi, \mathcal{V} \vdash (v : \tau) \Rightarrow \mathcal{V}', \Phi}$$

$$\textsc{Pat-Tuple}\ \frac{\begin{array}{c}\Gamma, \Phi \vdash \tau \prec \tau_{p_0} * \ .. \ * \tau_{p_n} \qquad \Gamma, \Phi, \Sigma \vdash (p_0 : \tau_0) \Rightarrow \mathcal{V}_0 \\ \forall i_{\geq 1}.\ \Gamma, \Phi_{i-1}, \Sigma \vdash p_i : \tau_i \Rightarrow \mathcal{V}_i, \Phi_i \qquad \forall i.\ \Gamma, \Phi_i \vdash \tau_{p_i} \equiv \tau_i \qquad \mathcal{V}' \leftarrow \mathcal{V} \uplus \mathcal{V}_0 \uplus \ .. \ \uplus \mathcal{V}_n\end{array}}{\Gamma, \Phi, \mathcal{V} \vdash (p_0, \ .. \ , p_n : \tau) \Rightarrow \mathcal{V}', \Phi_n}$$

$$\textsc{Pat-Or}\ \frac{\begin{array}{c}\Gamma, \Phi, \mathcal{V} \vdash (p_1 : \tau_1) \Rightarrow \mathcal{V}_1, \Phi_1 \\ \Gamma, \Phi_1, \mathcal{V} \vdash (p_2 : \tau_2) \Rightarrow \mathcal{V}_2, \Phi_2 \qquad \Gamma, \Phi_2 \vdash \tau_1 \equiv \tau_2 \qquad \Gamma, \Phi_2 \vdash \tau_2 \equiv \tau \qquad \Gamma, \Phi_2 \vdash \mathcal{V}_1 \equiv \mathcal{V}_2\end{array}}{\Gamma, \Phi, \mathcal{V} \vdash (p_1 \mid p_2 : \tau) \Rightarrow \mathcal{V}_2, \Phi_2}$$

$$\textsc{Pat-Construct}\ \frac{\begin{array}{c}\Gamma, \Phi, \Sigma \vdash (p_0 : \tau_0) \Rightarrow \mathcal{V}_0, \Phi_0 \qquad \forall i_{\geq 1}.\ \Gamma, \Phi_{i-1}, \Sigma \vdash (p_i : \tau_i) \Rightarrow \mathcal{V}_i, \Phi_i \\ (\tau_{arg_0}, \ .. \ , \tau_{arg_n}, \tau_{constr}) \leftarrow \mathit{find\_constructor}(\Gamma, \Phi, T, \tau) \qquad \Gamma, \Phi_n, \Sigma \vdash \tau_0 \leq \tau_{arg_0} \Rightarrow \theta_0 \\ \forall i_{\geq 1}.\ \Gamma, \Phi_n, \theta_{i-1} \vdash \tau_i \leq \tau_{arg_i} \Rightarrow \theta_i \qquad \forall i.\ \Phi_{p_i} \leftarrow \mathit{add\_equation}(\Gamma, \Phi_{p_{i-1}}, \tau_i, \tau_{arg_i}) \\ \tau_{inst} \leftarrow \theta_n(\tau_{constr}) \qquad \Gamma, \Phi_{p_n} \vdash \tau_{inst} \equiv \tau \qquad \mathcal{V}' \leftarrow \mathcal{V} \uplus \mathcal{V}_0 \uplus \ .. \ \uplus \mathcal{V}_n\end{array}}{\Gamma, \Phi, \mathcal{V} \vdash (T(p_0, \ .. \ , p_n) : \tau) \Rightarrow \mathcal{V}', \Phi_{p_n}}$$

Fig. 4. Pattern typechecking rules

merges such mappings, and fails when the same variable occurs twice. This is due to the linearity of OCaml patterns, in which no variable may occur twice. We also need to check the equivalence on such mappings: they must have the same domain and map variables to equivalent types. Equivalence of mappings is necessary for checking that two or-patterns effectively bind exactly the same variables. Typing a pattern also returns an ambivalence environment since matching a (GADT) data constructor can generate type equalities.

We introduce two important check operations: the instantiation $C, \theta \vdash \tau_1 \leq \tau_2 \Rightarrow \theta'$ (fig. 6) and the wellformedness $C \vdash \tau\ wf$ (fig. 5). The first is necessary at multiple occasions: when checking the type of a variable, we must check that its type is an instance of the type scheme provided by the typing environment. Instantiation generates a substitution from type variables to types. Instantiation is also necessary for checking type declarations: the user may constraint parameters, then the instantiation checks that the whole set of constraints is valid. The second detects possible types escaping their scope, in the context of a GADT or first-class modules. Indeed, the compiler generates "fresh" type constructors for existential types that appear in the branch of a pattern matching. The scope of those type constructors is the current branch: they therefore cannot occur in the resulting type. This check is also necessary for local modules (unpacking of first-class modules) that bring in the current context, types that are only valid in this context. Both constraints can be verified by Wf-Construct.

Generalization is checked in the Let and Let-Rec rules: the predicate $\mathit{check\_gen}(C, \sigma, e)$ takes a context C, a type scheme σ and an expression e and checks that it was correct to generalize (following the so-called "relaxed value restriction" [JG04]) the universally quantified type variables of σ. For an expression, being a "*value*" is left to the compiler's internals, and is used abstractly by our system.

Finally, the last abstract operation is $\mathit{add\_equation}(C, \tau_{annot}, \tau_{expected})$, used in the rule Pat-Construct. This operation takes an environment C, the annotated type τ_{annot} and the type expected for this pattern $\tau_{expected}$, which is actually the type expected for each argument of the data constructor. After type inference, τ_{annot} might contain rigid variables that will be used to generate ambivalences if the data constructor belongs to a GADT. The annotated pattern type (and its sub-patterns) can contain "fresh" existential variables, we must therefore add them as abstract types to the current matching branch for

$$\textsc{Wf-Var}\ C \vdash\ 'a\ wf \qquad \textsc{Wf-Fun}\ \frac{C \vdash \tau_1\ wf \qquad C \vdash \tau_2\ wf}{C \vdash (l : \tau_1) \to \tau_2\ wf} \qquad \textsc{Wf-Tuple}\ \frac{\forall \tau_i.\ C \vdash \tau_i\ wf}{C \vdash \tau_0 * .. * \tau_n\ wf}$$

$$\textsc{Wf-Construct}\ \frac{type\ (\tau_{param_0}, .., \tau_{param_n})\ \mathtt{t} \leftarrow \Gamma(\mathtt{t}) \qquad \forall i. C \vdash \tau_i\ wf \qquad C, \Sigma \vdash (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau_{param}})\ \mathtt{t} \Rightarrow \theta}{C \vdash (\overline{\tau})\ \mathtt{t}\ \ wf}$$

$$\textsc{Wf-Poly}\ \frac{\forall i.\ \Gamma, \Phi \vdash \tau_{\alpha_i} \prec \alpha_i \qquad \Gamma \oplus \overline{\alpha}, \Phi \vdash \tau\ wf}{\Gamma, \Phi \vdash \forall \overline{\tau_\alpha}.\ \tau\ \ wf} \qquad \textsc{Wf-Univar}\ \frac{\alpha \in \Gamma}{C \vdash \alpha\ wf}$$

$$\textsc{Wf-Variant}\ \frac{\begin{array}{c}\forall i, j.\ i \neq j \implies T_i \neq T_j \\ \forall i. C \vdash \tau_i\ wf \qquad \forall T_{pres_i} \in \{T_{pres_i} .. T_{pres_j}\}.\ T_{pres_i} \in \overline{T} \qquad C \vdash \rho \prec\ 'a \bigvee C \vdash \rho \prec \epsilon\end{array}}{C \vdash [(\rho)\ \overline{\sim T\ \mathtt{of}\ \tau} > T_{pres_i} .. T_{pres_j}]\ wf}$$

$$\textsc{Wf-Package}\ \frac{\mathtt{P} \in \Gamma.Modtypes \qquad \forall i.\ \mathtt{t_i} \in \Gamma.Modtypes(\mathtt{P}) \qquad \forall i. C \vdash \tau_i\ wf \qquad \forall i\ s.t.\ transparent(\mathtt{P.t_i}).\ C \vdash \tau_i \leq \mathtt{P.t_i}}{C \vdash (\mathtt{module\ P\ with}\ \overline{type\ \mathtt{t} = \tau})\ wf}$$

Fig. 5. The type wellformedness condition

the wellformedness to be correct. Each type that is added to an ambivalence is checked for equivalence with every existing concrete type in this set. The formal definition if this algorithm is left for a future work. Actually, the equations generated by the compiler can be retrieved in the environment, but in the current implementation, we check that those equalities are correct.

4 Results and future work

The current type checker works on OCaml 4.02. Since this version was released, some bugs in the type systems have been discovered, that were related to GADTs. For example, the program given at figure 7 leads to a runtime type error, due to a bug that allows the user to cast any value to an arbitrary type. The reason is that in the functor `Fix`, the type engine inferred that in the function `uniq`, the data constructor `Eq` (used as a pattern here) can have type `((a, a F.f) eq)`, or, in other words, that `(a F.f)` and `a` can be seen as equivalent. It is then possible to instantiate the functor `Fix` with such a type, and use `uniq` to generate an equality witness that is otherwise impossible to produce. Actually, we can easily find such an error when typing the first ocurrence of `Eq` as pattern in `uniq`: we must check that its inferred type is an instance of the type of its declaration (`('a, 'a) eq`). We first associate `'a` with the locally abstract type `a`, then when instantiating the second `'a`, we check that the already bound `'a` can be associated with `(a F.f)`. Since `('a F.f)` is abstract, we cannot deduce that `(a F.f)` and `a` are equivalent. This instantiation is therefore incorrect.

Another recent bug is given at figure 8. The resulting type of `undetected` will contain a type constructor `b#0`, that corresponds to an existential type constructor generated by the compiler. This existential type constructor has no inhabitant, thus this function will never be applied, but `b#0` escapes its scope whereas it should not. The rule `Wf-Construct` (that is called during `Match`) can easily detect this case.

A drawback of type checking annotated TASTs is that they are difficult to produce by other ways than the OCaml compiler: it is therefore difficult to produce inconsistent TASTs, except when there is a regression in the compiler, as we have shown

$$\textsc{Inst-Var-Unbound}\ \frac{'a \notin \theta}{C, \theta \vdash\ \tau \leq\ 'a \Rightarrow \theta \oplus ['a \to \tau]} \qquad \textsc{Inst-Var-Bound}\ \frac{'a \in \theta \qquad C, \theta \vdash \tau_{'a} \equiv \tau}{C, \theta \oplus ['a \to \tau_{'a}] \vdash\ \tau \leq\ 'a \Rightarrow \theta \oplus ['a \to \tau_{'a}]}$$

$$\textsc{Inst-Var-Generalized}\ \frac{generalized('a_1) \implies generalized('a_2)}{C, \theta \vdash\ 'a_1 \leq\ 'a_2 \Rightarrow \theta \oplus ['a_2 \to' a_1]}$$

$$\textsc{Inst-Fun}\ \frac{l_1 = l_1' \qquad C, \theta \vdash \tau_1 \leq \tau_1' \Rightarrow \theta_1 \qquad C, \theta_1 \vdash \tau_2 \leq \tau_2' \Rightarrow \theta_2}{C, \theta \vdash (l_1 : \tau_1) \to \tau_2 \leq (l_1' : \tau_1') \to \tau_2' \Rightarrow \theta_2}$$

$$\textsc{Inst-Tuple}\ \frac{C, \theta \vdash \tau_0 \leq \tau_0' \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ C, \theta_{i-1} \vdash \tau_i \leq \tau_i' \Rightarrow \theta_i}{C, \theta \vdash \tau_0 * .. * \tau_n \leq \tau_0' * .. * \tau_n' \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct}\ \frac{C, \theta \vdash \tau_0 \leq \tau_0' \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ C, \theta_{i-1} \vdash \tau_i \leq \tau_i' \Rightarrow \theta_i}{C, \theta \vdash (\overline{\tau})\ \mathtt{t} \leq (\overline{\tau'})\ \mathtt{p} \Rightarrow \theta_n}$$

$$\textsc{Inst-Construct-Exp-Left}\ \frac{\tau \leftarrow expand(C, \mathtt{t}, \overline{\tau}) \qquad C, \theta \vdash \tau \leq \tau' \Rightarrow \theta'}{C, \theta \vdash (\overline{\tau})\ \mathtt{t} \leq \tau' \Rightarrow \theta'}$$

$$\textsc{Inst-Construct-Exp-Right}\ \frac{\tau' \leftarrow expand(C, \mathtt{t}', \overline{\tau'}) \qquad C, \theta \vdash \tau \leq \tau' \Rightarrow \theta'}{C, \theta \vdash \tau \leq (\overline{\tau'})\ \mathtt{t}' \Rightarrow \theta'}$$

$$\textsc{Inst-Poly}\ \frac{\theta' \leftarrow \forall i.\theta \oplus [\alpha_i' \to \alpha_i] \qquad C, \theta' \vdash \tau \leq \tau' \Rightarrow \theta''}{C, \theta \vdash \forall \overline{\alpha}.\ \tau \leq \forall \overline{\alpha'}.\ \tau' \Rightarrow \theta''} \qquad \textsc{Inst-Univar}\ \frac{[\alpha' \to \alpha] \in \theta}{C, \theta \vdash \alpha \leq \alpha' \Rightarrow \theta}$$

$$\textsc{Inst-Rigid1}\ \frac{C.\Phi(\mathtt{t}) = C.\Phi(\tau)}{C.\Gamma.Rigid \oplus (\mathtt{t}), \theta \vdash \tau \leq \mathtt{t} \Rightarrow \theta} \qquad \textsc{Inst-Rigid2}\ \frac{C.\Phi(\mathtt{t}) = C.\Phi(\tau)}{C.\Gamma.Rigid \oplus (\mathtt{t}), \theta \vdash \mathtt{t} \leq \tau \Rightarrow \theta}$$

$$\textsc{Inst-Variant}\ \frac{T_{pres_i} .. T_{pres_j} \subseteq T'_{pres_i} .. T'_{pres_j} \qquad C, \theta \vdash \rho \leq \rho' \Rightarrow \theta_\rho \qquad \forall i.\ T_i \in \overline{T'} \implies C, \theta_{i-1} \vdash \tau_{var_i} \leq \tau'_{var_i} \Rightarrow \theta_i}{C, \theta \vdash [(\rho)\ \overline{T\ of\ \tau_{var}} > T_{pres_i} .. T_{pres_j}] \leq [(\rho')\ \overline{T'\ of\ \tau'_{var}} > T'_{pres_i} .. T'_{pres_j}] \Rightarrow \theta_n}$$

$$\textsc{Inst-Nil}\ C, \theta \vdash \epsilon \leq \epsilon \Rightarrow \theta$$

$$\textsc{Inst-Package}\ \frac{S \leftarrow Sig(\mathtt{P}\ \mathtt{with}\ \overline{\mathtt{type}\ \mathtt{t} = \tau}) \qquad S' \leftarrow Sig(\mathtt{P}'\ \mathtt{with}\ \overline{\mathtt{type}\ \mathtt{t}' = \tau'}) \qquad C, \theta \vdash S' :> S \Rightarrow \theta'}{C, \theta \vdash (\mathtt{module}\ \mathtt{P}\ \mathtt{with}\ \overline{\mathtt{type}\ \mathtt{t} = \tau}) \leq (\mathtt{module}\ \mathtt{P}'\ \mathtt{with}\ \overline{\mathtt{type}\ \mathtt{t}' = \tau'}) \Rightarrow \theta'}$$

Fig. 6. Type instanciation rules

above. One solution could be to create a subset of the language where every node is annotated and directly generate a TAST after parsing inconsistently annotated such programs.

As for now, we only cover a subset of the full OCaml language: objects and classes are not yet formalized even though their checking is implemented in our typechecker. Variance annotations are not yet checked: we plan to deal with them in the future. As for recursive modules, we chose not to cover them, since they are currently unspecified.

```
type (_, _) eq = Eq : ('a, 'a) eq
let cast : type a b . (a, b) eq -> a -> b = fun Eq x -> x

module Fix (F : sig type 'a f end) = struct
  type 'a fix = ('a, 'a F.f) eq
  let uniq (type a) (type b) (Eq : a fix) (Eq : b fix) : (a, b) eq = Eq
end

module FixId = Fix (struct type 'a f = 'a end)
let bad : (int, string) eq = FixId.uniq Eq Eq
let _ = Printf.printf "Oh dear: %s\n" (cast bad 42)
```

Fig. 7. An unsound OCaml program

```
type +'a n = private int
type nil = private Nil_type
type (_,_) elt =
  | Elt_fine : 'nat n -> ('l, 'nat * 'l) elt
  | Elt : 'nat n -> ('l, 'nat -> 'l) elt
type _ t = Nil : nil t | Cons : ('x, 'fx) elt * 'x t -> 'fx t

let undetected: ('a -> 'b -> nil) t -> 'a n -> 'b n -> unit =
  fun sh i j -> let _ = match sh with Cons(Elt dim, _) -> dim in ()
```

Fig. 8. Another unsound OCaml program

5 Conclusion

We presented in this extended abstract a type checker of type-annotated OCaml abstract syntax trees, such as those produced by the OCaml type inference engine. Our type checker is able to process a large subset of OCaml, and can be presented — almost pretty-printed — as a set of inference rules, providing an effective definition of the language's type system. Although it is well-known that type-annotated ASTs produced by the type inference engine *are* indeed typing proofs, and that, quite obviously, a checker for typing proofs can be presented as a type system, the design and implementation of such a type checker for OCaml is new, and its presentation as a type system provides us with the first effective presentation of the OCaml type system.

Furthermore, having such a type checker as part of the compiler tools, and letting it follow the evolution of the language's static semantics can be helpful for both the compiler implementors (for, e.g. building test suites dedicated to the type inference), and the programmers, since it may constitute a reference for the static semantics of the language.

References

[JG04] Jacques Garrigue. "Functional and Logic Programming: 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004. Proceedings". In: ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. Chap. Relaxing the Value Restriction, pp. 196–213. ISBN: 978-3-540-24754-8. DOI: 10.1007/978-3-540-24754-8_15. URL: http://dx.doi.org/10.1007/978-3-540-24754-8_15 (cit. on p. 5).

[JG01] Jacques Garrigue. “Labeled and optional arguments for Objective Caml”. In: *In JSSST Workshop on Programming and Programming Languages*. 2001 (cit. on p. 3).

[GL11] Jacques Garrigue and Jacques Le Normand. “Adding GADTs to OCaml: a direct approach”. In: ML ’11 (Workshop on ML and its applications) (2011) (cit. on p. 1).

[GR13] Jacques Garrigue and Didier Rémy. “Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings”. In: ed. by Chung-chieh Shan. Cham: Springer International Publishing, 2013. Chap. Ambivalent Types for Principal Type Inference with GADTs, pp. 257–272. ISBN: 978-3-319-03542-0. DOI: 10.1007/978-3-319-03542-0_19. URL: http://dx.doi.org/10.1007/978-3-319-03542-0_19 (cit. on p. 3).

[XL95] Xavier Leroy. “Applicative functors and fully transparent higher-order modules”. In: *22nd symposium Principles of Programming Languages*. ACM Press, 1995, pp. 142–153 (cit. on p. 1).

[LD14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.02): Documentation and user’s manual*. Inria. Sept. 2014. URL: http://caml.inria.fr/pub/docs/manual-ocaml/ (cit. on p. 1).

[PR06] François Pottier and Yann Régis-Gianas. “Stratified type inference for generalized algebraic data types”. In: POPL ’06 (Principle of Programming Languages) (Jan. 2006), pp. 232–244. DOI: http://doi.acm.org/10.1145/1111037.1111058. URL: http://gallium.inria.fr/~fpottier/publis/pottier-regis-gianas-popl06.ps.gz (cit. on p. 1).

[DR92] Didier Rémy. *Extending ML Type System with a Sorted Equational Theory*. Research Report 1766. Institut National de Recherche en Informatique et Automatisme, 1992 (cit. on p. 2).

[DV97] Didier Rémy and Jérôme Vouillon. “Objective ML: A simple object-oriented extension of ML”. In: *Proceedings of the 24th ACM Conference on Principles of Programming Languages*. Paris, France, 1997, pp. 40–53 (cit. on p. 1).

6 Annex

We list here the rules for the rest of the (large) subset of OCaml that is covered by our type checker.

6.1 OCaml expression (without objects)

$$\textsc{Tuple}\ \frac{\forall i.\ C \vdash (e_i : \tau_i') \qquad C \vdash \tau \prec \tau_0\ *\ ..\ *\ \tau_n \qquad \forall i. C \vdash \tau_i \equiv \tau_i'}{C \vdash ((e_0,\ ..\ ,\ e_n) : \tau)}$$

$$\textsc{Match}\ \frac{\begin{array}{c} C \vdash (e : \tau_{scrut}) \qquad \forall i.\ C, \Sigma \vdash (p_i : \tau_i) \Rightarrow (\overline{v_i : \tau_i}), \Phi_i \\ \forall i.\ C_i \leftarrow C.\Gamma \oplus_{\mathcal{V}} (\overline{v_i : \tau_i}), \Phi_i \qquad \forall i.\ C_i \vdash (e_i : \tau_{e_i}) \qquad \forall i.\ C \vdash \tau_{e_i}\ wf \qquad \forall i.\ C_i \vdash \tau \equiv \tau_{e_i} \end{array}}{C \vdash (\texttt{match}\ e\ \texttt{with}\ \overline{\mid p \rightarrow e} : \tau)}$$

$$\textsc{Record}\ \frac{\begin{array}{c} \tau_{rec} \leftarrow C.\Gamma.Types(\tau) \qquad \forall i.\ \tau_{l_i} \leftarrow find\_label(C, l_i, \tau_{rec}) \qquad \forall i.\ C \vdash (e : \tau_i) \\ C, \Sigma \vdash \tau_0 \leq \tau_{l_0} \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ C, \theta_{i-1} \vdash \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \qquad \tau_{inst} \leftarrow \theta_n(\tau_{rec}) \qquad C \vdash \tau_{inst} \equiv \tau \end{array}}{C \vdash (\{l_0 = e_0;\ ..\ ; l_n = e_n\} : \tau)}$$

$$\textsc{Field}\ \frac{\begin{array}{c} C \vdash (e : \tau_e) \qquad \tau_{rec} \leftarrow C.\Gamma.Types(\tau_e) \\ \tau_l \leftarrow find\_label(C, l, \tau_{rec}) \qquad C, \Sigma \vdash \tau \leq \tau_l \Rightarrow \theta \qquad \tau_{inst} \leftarrow \theta(\tau_{rec}) \qquad C \vdash \tau_{inst} \equiv \tau_e \end{array}}{C \vdash (e.l : \tau)}$$

$$\textsc{Set-Field}\ \frac{\begin{array}{c} C \vdash (e_1 : \tau_1) \qquad C \vdash (e_2 : \tau_2) \\ \tau_{rec} \leftarrow C.\Gamma.Types(\tau_1) \qquad \tau_l \leftarrow find\_label(C, l, \tau_{rec}) \qquad label\_kind(C, l, \tau_{rec}) = Mutable \\ C, \Sigma \vdash \tau_2 \leq \tau_l \Rightarrow \theta \qquad \tau_{inst} \leftarrow \theta(\tau_{rec}) \qquad C \vdash \tau_{inst} \equiv \tau_1 \qquad C \vdash \tau \equiv \texttt{unit} \end{array}}{C \vdash (e_1.l \leftarrow e_2 : \tau)}$$

$$\textsc{Array}\ \frac{\forall i.\ C \vdash (e_i : \tau_i) \qquad \forall i_{\geq 1}.\ C \vdash \tau_{i-1} \equiv \tau_i \qquad C \vdash \tau \prec \tau_{arg}\ \texttt{array} \qquad C \vdash \tau_0 \equiv \tau_{arg}}{C \vdash ([|e_0;\ ..\ ; e_n|] : \tau)}$$

$$\textsc{Sequence}\ \frac{C \vdash (e_1 : \tau_1) \qquad C \vdash (e_2 : \tau_2) \qquad C \vdash \tau_1 \equiv \texttt{unit} \qquad C \vdash \tau \equiv \tau_2}{C \vdash (e_1;\ e_2 : \tau)}$$

$$\textsc{While}\ \frac{C \vdash (e_1 : \tau_1) \qquad C \vdash (e_2 : \tau_2) \qquad C \vdash \tau_1 \equiv \texttt{bool} \qquad C \vdash \tau_2 \equiv \texttt{unit} \qquad C \vdash \tau \equiv \texttt{unit}}{C \vdash (\texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{done} : \tau)}$$

$$\textsc{For}\ \frac{\begin{array}{c} C \vdash (e_1 : \tau_1) \qquad C \vdash (e_2 : \tau_2) \qquad C \vdash \tau_1 \equiv \texttt{int} \\ C \vdash \tau_2 \equiv \texttt{int} \qquad C.\Gamma \oplus_{\mathcal{V}} (x, \tau_1), C.\Phi \vdash (e_3 : \tau_3) \qquad C \vdash \tau_3 \equiv \texttt{unit} \qquad C \vdash \tau \equiv \texttt{unit} \end{array}}{C \vdash (\texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} : \tau)}$$

$$\textsc{Raise}\ \frac{C \vdash (e : \tau_e) \qquad C \vdash (\tau_e \equiv \texttt{exn}) \qquad C \vdash (\tau \equiv \texttt{unit})}{C \vdash (\textbf{assert}\ e : \tau)}$$

$$\textsc{Assert}\ \frac{C \vdash (e : \tau_e) \qquad C \vdash (\tau_e \equiv \texttt{bool}) \qquad C \vdash (\tau \equiv \texttt{unit})}{C \vdash (\textbf{assert}\ e : \tau)}$$

$$\textsc{Assert-False}\ \frac{C \vdash (\mathit{false} : \tau_e) \qquad C \vdash (\tau_e \equiv \texttt{bool})}{C \vdash (\textbf{assert}\ \mathit{false} : \tau)}$$

$$\textsc{Lazy}\ \frac{C \vdash (e : \tau_e) \qquad C \vdash \tau \prec \tau_{arg}\ \texttt{Lazy.t} \qquad C \vdash \tau_e \equiv \tau_{arg}}{C \vdash (\textbf{lazy}\ e : \tau)} \qquad \textsc{Variant-Const}\ \frac{C \vdash \tau \prec [(\rho)\ ..\ T\ ..\ >\ ..\ T\ ..\ ]}{C \vdash (\text{`}T : \tau)}$$

$$\textsc{Variant}\ \frac{C \vdash \tau \prec [(\rho)..\ T\ of\ \tau_{arg}\ ..\ >\ ..\ T\ ..\ ] \qquad C \vdash (e : \tau_e) \qquad C \vdash \tau_{arg} \equiv \tau_e}{C \vdash (\text{`}T\ e : \tau)}$$

$$\textsc{Constraint}\ \frac{C, \Sigma \vdash (t : \tau_{cstr}) \Rightarrow \mathcal{V} \qquad C \vdash (e : \tau_e) \qquad C, \Sigma \vdash \tau_e \leq \tau_{cstr} \Rightarrow \theta \qquad C \vdash \tau \equiv \tau_e}{C \vdash ((e : t) : \tau)}$$

$$\textsc{Newtype}\ \frac{C.\Gamma.(\mathit{Rigid} \oplus \texttt{t}, \mathit{Types} \oplus (\texttt{t}, \mathit{Abstract}) \vdash (e : \tau_e) \qquad C \vdash \tau_e\ wf \qquad C \vdash \tau \equiv \tau_e}{C \vdash (\textbf{fun}\ (\textbf{type}\ \texttt{t}) \rightarrow e : \tau)}$$

$$\textsc{Let-Module}\ \frac{C \vdash (M : \mathcal{M}) \qquad C.\Gamma.\mathit{Module} \oplus (\texttt{I}, M) \vdash (e : \tau_e) \qquad C \vdash \tau_e\ wf \qquad C \vdash \tau \equiv \tau_e}{C \vdash (\textbf{let module}\ \texttt{I} = M\ \textbf{in}\ e : \tau)}$$

6.2 OCaml patterns

$$\textsc{Pat-Record}\ \frac{\begin{array}{c}\tau_{rec} \leftarrow find\_record((\Gamma,\Phi),\tau) \qquad \forall i.\ \tau_{l_i} \leftarrow find\_label((\Gamma,\Phi),l_i,\tau_{rec}) \\ \forall i.\ \Gamma,\Phi_{i-1},\Sigma\ \vdash (p_i:\tau_i) \Rightarrow \mathcal{V}_i,\Phi_i \qquad \Gamma,\Phi_0 \vdash \tau_0 \leq \tau_{l_0} \Rightarrow \theta_0 \qquad \forall i_{\geq 1}.\ \Gamma,\Phi_i,\theta_{i-1} \vdash \tau_i \leq \tau_{l_i} \Rightarrow \theta_i \\ \tau_{inst} \leftarrow \theta_n(\tau_{rec}) \qquad \Gamma,\Phi_n \vdash \tau_{inst} \equiv \tau \qquad \mathcal{V}' \leftarrow \mathcal{V} \uplus \mathcal{V}_0 \uplus\ ..\ \uplus \mathcal{V}_n\end{array}}{\Gamma,\Phi,\mathcal{V}\ \vdash (\{l_0 = p_0;\ ..\ ; l_n = p_n\} : \tau) \Rightarrow \mathcal{V}',\Phi_n}$$

$$\textsc{Pat-Array}\ \frac{\begin{array}{c}\forall i.\ \Gamma,\Phi,\Sigma \vdash (p_i:\tau_i) \Rightarrow \mathcal{V}_i,\Phi_i \\ \forall i_{\geq 1}.\ \Gamma,\Phi_i \vdash \tau_{i-1} \equiv \tau_i \qquad \Gamma,\Phi \vdash \tau \prec \tau'\ \texttt{array} \qquad \Gamma,\Phi_0 \vdash \tau_0 \equiv \tau \qquad \mathcal{V}' \leftarrow \mathcal{V} \uplus \mathcal{V}_0 \uplus\ ..\ \uplus \mathcal{V}_n\end{array}}{\Gamma,\Phi,\mathcal{V} \vdash ([|p_0;\ ..\ ; p_n|] : \tau) \Rightarrow \mathcal{V}',\Phi_n}$$

$$\textsc{Pat-Lazy}\ \frac{\Gamma,\Phi,\mathcal{V} \vdash p : \tau_p \Rightarrow \mathcal{V}',\Phi_p \qquad \Gamma,\Phi \vdash \tau \prec \tau'\ \texttt{Lazy.t} \qquad \Gamma,\Phi_p \vdash \tau' \equiv \tau_p}{\Gamma,\Phi,\mathcal{V} \vdash (\texttt{lazy}\ p : \tau) \Rightarrow \mathcal{V}',\Phi_p}$$

$$\textsc{Pat-Variant-Const}\ \frac{\Gamma,\Phi_p \vdash \tau \prec [(\rho)\ T > ..\ ]}{\Gamma,\Phi,\mathcal{V} \vdash (`T : \tau) \Rightarrow \mathcal{V},\Phi}$$

$$\textsc{Pat-Variant}\ \frac{\Gamma,\Phi \vdash \tau \prec [(\rho)\ T\ of\ \tau_{arg} > ..\ ] \qquad \Gamma,\Phi,\mathcal{V} \vdash p : \tau_p \Rightarrow \mathcal{V}',\Phi_p \qquad \Gamma,\Phi_p \vdash \tau_{arg} \equiv \tau_p}{\Gamma,\Phi,\mathcal{V} \vdash (`T\ p : \tau) \Rightarrow \mathcal{V}',\Phi_p}$$

6.3 OCaml types annotations

These annotations can appear as constraints on expressions, as type of labels from record declaration or arguments of constructor declarations.

$$\textsc{Core-Var-Bound}\ \frac{C \vdash \tau \equiv \tau_a}{C, \mathcal{V} \oplus ('a, \tau_a) \vdash\ ('a : \tau) \Rightarrow \mathcal{V} \oplus ('a, \tau_a)} \qquad \textsc{Core-Var-Unbound}\ \frac{}{C, \mathcal{V} \vdash\ ('a : \tau) \Rightarrow V \oplus ('a, \tau)}$$

$$\textsc{Core-Any}\ C, \mathcal{V} \vdash (\_ : \tau) \Rightarrow \mathcal{V}$$

$$\textsc{Core-Arrow}\ \frac{C, \mathcal{V} \vdash t_1 : \tau_1 \Rightarrow \mathcal{V}_1 \qquad C, \mathcal{V}_1 \vdash t_2 : \tau_2' \Rightarrow \mathcal{V}_2 \qquad l = l' \qquad C \vdash \tau_d \equiv \tau_1 \qquad C \vdash \tau_{cd} \equiv \tau_2}{C, \mathcal{V} \vdash ((l : t_1) \to t_2 :\ (l' : \tau_d) \to \tau_{cd}) \Rightarrow \mathcal{V}_2}$$

$$\textsc{Core-Tuple}\ \frac{C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \qquad \forall i_{\geq 1}.\, C, \mathcal{V}_{i-1} \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i'}{C, \mathcal{V} \vdash (t_0 * .. * t_n : \tau_0 * .. * \tau_n) \Rightarrow \mathcal{V}_n}$$

$$\textsc{Core-Constr}\ \frac{\begin{array}{c} C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \\ \forall i_{\geq 1}.\, C, \mathcal{V}_{i-1}, \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i' \qquad \mathtt{t} = \mathtt{t}' \qquad C \vdash (\overline{\tau})\ \mathtt{t}'\ wf \end{array}}{C, \mathcal{V} \vdash ((\overline{t})\ \mathtt{t} : (\overline{\tau})\ \mathtt{t}') \Rightarrow \mathcal{V}_n}$$

$$\textsc{Core-Poly}\ \frac{\forall i.\, C \vdash \tau_i \prec \alpha_i \qquad C, \mathcal{V} \oplus (\overline{'a, \alpha}) \vdash (t : \tau_{poly}) \Rightarrow \mathcal{V}_{poly} \qquad C \vdash \tau \equiv \tau_{poly}}{C, \mathcal{V} \vdash (\overline{'a}.\, t : \overline{\tau}.\, \tau) \Rightarrow \mathcal{V}_{poly}}$$

$$\textsc{Core-Alias-Nonrec}\ \frac{C, \mathcal{V} \vdash (t : \tau_{al}) \Rightarrow \mathcal{V}_{al} \qquad 'a \notin \mathcal{V}_{al} \qquad C \vdash \tau \equiv \tau_{al}}{C, \mathcal{V} \vdash (t\ \mathtt{as}\ 'a) : \tau \Rightarrow \mathcal{V}_{al}}$$

$$\textsc{Core-Variant-Static}\ \frac{\begin{array}{c} C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \\ \forall i.\, C, \mathcal{V}_{i-1} \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \qquad \forall i.\, T_i \in [T_{pres_i} .. T_{pres_j}] \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i' \qquad C \vdash \rho \prec \epsilon \end{array}}{C, \mathcal{V} \vdash [\overline{T\ of\ t}] : [(\rho)\ \overline{|\ T\ ?of\ \tau} > T_{pres_i} .. T_{pres_j}] \Rightarrow \mathcal{V}_n}$$

$$\textsc{Core-Variant-Closed}\ \frac{\begin{array}{c} C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \qquad \forall i.\, C, \mathcal{V}_{i-1} \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \\ \forall i.\, T_{pres_i} \in [T'_{pres_i} .. T'_{pres_j}] \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i' \qquad C \vdash \rho \prec' a \end{array}}{C, \mathcal{V} \vdash [\overline{T\ of\ t} > T_{pres_i} .. T_{pres_j}] : [(\rho)\ \overline{|\ T\ ?of\ \tau} > T_{pres_i} .. T_{pres_j}] \Rightarrow \mathcal{V}_n}$$

$$\textsc{Core-Variant-Open}\ \frac{\begin{array}{c} C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \\ \forall i.\, C, \mathcal{V}_{i-1} \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \qquad \forall i.\, T_i \in [T_{pres_i} .. T_{pres_j}] \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i' \qquad C \vdash \rho \prec' a \end{array}}{C, \mathcal{V} \vdash [> \overline{T\ of\ t}] : [(\rho)\ \overline{|\ T\ ?of\ \tau} > T_{pres_i} .. T_{pres_j}] \Rightarrow \mathcal{V}_n}$$

$$\textsc{Core-Package}\ \frac{\begin{array}{c} \mathtt{S} = \mathtt{S}' \qquad \forall i.\, \mathtt{t_i} = \mathtt{t'_i} \qquad C, \mathcal{V} \vdash (t_0 : \tau_0') \Rightarrow \mathcal{V}_0 \\ \forall i_{\geq 1}.\, C, \mathcal{V}_{i-1} \vdash (t_i : \tau_i') \Rightarrow \mathcal{V}_i \qquad \forall i.\, C \vdash \tau_i \equiv \tau_i' \qquad C \vdash (\mathtt{module}\ S'\ \mathtt{with\ type}\ \overline{\mathtt{t}' = \tau})\ wf \end{array}}{C, \mathcal{V} \vdash ((\mathtt{module}\ \mathtt{S}\ \mathtt{with\ type}\ \overline{\mathtt{t} = t}) : (\mathtt{module}\ \mathtt{S}'\ \mathtt{with\ type}\ \overline{\mathtt{t}' = \tau})) \Rightarrow \mathcal{V}_n}$$