Chapter 1

A polymorphic applicative language

The full ML language is too complex to allow easy formal reasoning about it. This chapter introduces a small applicative language with a polymorphic type discipline, that presents the main features of ML, while remaining much simpler. This language is the basis for the extensions studied in the next chapters.

The present chapter contains nothing that has not been already written several times [62, 21, 22, 92, 17, 59, 60, 93]. Nonetheless, I attempt to give a self-contained presentation with few prerequisites. This presentation introduces most of the notations and proof techniques that are widely used in the remainder of this work.

1.1 Syntax

The EXPRESSIONS of the language (written a, possibly with a prime or a subscript) are the elements of the term algebra generated by the following grammar:

$$
\begin{array}{rcll}
a & ::= & cst & \text{constant} \\
 & \mid & x & \text{identifier} \\
 & \mid & op(a) & \text{primitive application} \\
 & \mid & f \ \texttt{where} \ f(x) = a & \text{recursive function} \\
 & \mid & a_1(a_2) & \text{application} \\
 & \mid & \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 & \text{the } \texttt{let} \text{ binding} \\
 & \mid & (a_1, a_2) & \text{pair construction}
\end{array}
$$

In this grammar, x and f range over an infinite set `Ident` of IDENTIFIERS. Expressions of the form cst are CONSTANTS, taken from a set `Cst` that we do not specify here, for the sake of generality. For instance, `Cst` might contain integer numbers, the true values `true` and `false`, character strings, etc.

In the expressions $op(a)$, op ranges over a set of OPERATORS `Op` which is also left unspecified. Typically, `Op` contains the usual arithmetic operations over integers, the comparisons between integers, the two projections `fst` and `snd`; even the usual conditional constructs such as `if`...`then`...`else`... can be treated as operators.

The expression f `where` $f(x) = a$ defines the function whose parameter is x and whose result is the value of a. The identifier f is the internal name for the function. Inside the expression a, the identifier f is considered bound to the function being defined. The defined function can therefore be recursive. In the examples below, we write $\lambda x.\, a$ for non-recursive functions. This is an abbreviation for f `where` $f(x) = a$, where f is any identifier that does not appear in a.

Context. The toy languages studied in the literature usually do not feature recursion. As a consequence, they are unrealistic: it is hard to write interesting examples in these languages. Also, they often possess properties that do not hold for any "real" programming language: "all well-typed programs terminate", for instance. Moreover, it is not easy to add recursion as an afterthought: the proofs, and sometimes even the formalism, must be entirely revised. That's why I have chosen to consider a recursive language from the beginning.

In the ML language, recursion is presented as an extension of the `let` construct: `let rec`, `let val rec`, `let fun` depending on the dialects. I prefer to present recursion as an extension of λ-abstraction; this way, it is apparent that only functions can be defined recursively, and not any value. (In ML, extra restrictions are required to rule out definitions such as $\texttt{let rec x} = \texttt{x} + \texttt{1}$, that cannot be executed.) In Standard ML syntax, f `where` $f(x) = a$ is written `let fun` $f\ x = a$ `in` f `end`; in Caml syntax, `let rec` $f\ x = a$ `in` f, or f `where rec` $f\ x = a$. □

Example. Here are two syntactically correct expressions of the language:

$$\texttt{let fact} = \texttt{f where f(n)} = \texttt{ifthenelse}(\leq(\texttt{n},\texttt{0}),(\texttt{1},\times(\texttt{n},\texttt{f}(-(\texttt{n},\texttt{1}))))) \ \texttt{in fact(2)}$$

$$\texttt{let double} = \lambda\texttt{f}.\,\lambda\texttt{x}.\,\texttt{f(f(x))} \ \texttt{in double}(\lambda\texttt{x}.\,+(\texttt{x},\texttt{true}))(\texttt{1})$$

For the sake of readability, we take the liberty to use the familiar infix notation for binary and ternary operators, as well as the usual precedence and associativity rules. We therefore write the expressions as:

$$\texttt{let fact} = \texttt{f where f(n)} = \texttt{if n} \leq \texttt{0 then 1 else n} \times \texttt{f(n} - \texttt{1)} \ \texttt{in fact(2)}$$

$$\texttt{let double} = \lambda\texttt{f}.\,\lambda\texttt{x}.\,\texttt{f(f(x))} \ \texttt{in double}(\lambda\texttt{x.x} + \texttt{true}))(\texttt{1})$$

□

We write $\mathcal{I}(a)$ for the set of the FREE IDENTIFIERS of expression a. The binding constructs are `let` and `where`. The precise definition of $\mathcal{I}$ is:

$$\begin{array}{rclrcl}
\mathcal{I}(cst) & = & \emptyset & \mathcal{I}(x) & = & \{x\} \\
\mathcal{I}(op(a)) & = & \mathcal{I}(a) & \mathcal{I}(f \ \texttt{where} \ f(x) = a) & = & \mathcal{I}(a) \setminus \{f, x\} \\
\mathcal{I}(a_1(a_2)) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & \mathcal{I}(\texttt{let} \ x = a_1 \ \texttt{in} \ a_2) & = & \mathcal{I}(a_1) \cup (\mathcal{I}(a_2) \setminus \{x\}) \\
\mathcal{I}(a_1, a_2) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & & &
\end{array}$$

1.2 Semantics

We are now going to give a meaning to the language expressions, by defining an evaluation mechanism that maps expressions to the results of their evaluations. To do so, we use the formalism of relational semantics. It consists in defining, by a system of axioms and inference rules, a predicate between expressions and results, the EVALUATION JUDGEMENT, telling whether a given expression can evaluate to a given result.

Context. This method is known as "structured operational semantics" (SOS) in Northern Great Britain [73], and as "natural semantics" in Southern France [43]. In contrast with denotational semantics, relational semantics require no complex mathematical framework; it also remains closer to direct execution on a computer [16]. Other mathematically simple approaches include rewriting semantics (for the extensions in next chapter, simple rewriting does not suffices, rewriting rules with contexts are required [27, 26, 100]), and also translating expressions into code for an abstract machine [46, 18]. □

Actually, the evaluation judgement depends on extra arguments, that represent the context in which evaluation takes place. For the language considered in the present chapter, only one such extra argument is required: the evaluation environment, that records the bindings of identifiers to values. The evaluation judgement is therefore $e \vdash a \Rightarrow r$, read: "in the evaluation environment e, the expression a evaluates to the result r".

1.2.1 Semantic objects

The time has come to define precisely the various kinds of objects that appear in the semantics: the RESULTS, the VALUES, and the EVALUATION ENVIRONMENTS. These semantic objects range over the term algebra defined by the grammar below:

Results:	r	$::=$	v	normal result (a value)
		$\mid$	`err`	error result
Value:	v	$::=$	cst	base value
		$\mid$	(v_1, v_2)	value pair
		$\mid$	(f, x, a, e)	functional value (closure)
Environments:	e	$::=$	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	

In environment expressions e, we assume the identifiers $x_1 \ldots x_n$ to be distinct. Here and elsewhere, the term $e = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ is interpreted as a partial mapping with finite domain from identifiers to values: the mapping that associates v_i to x_i, for all i from 1 to n, and that is undefined on the other identifiers. We improperly call e a finite mapping from identifiers to values. The empty mapping is written $[\,]$. We write $\mathrm{Dom}(e)$ for the DOMAIN of e, that is $\{x_1, \ldots, x_n\}$, and $\mathrm{Codom}(e)$ for its RANGE, that is $\{v_1, \ldots, v_n\}$. If x belongs to $\mathrm{Dom}(e)$, we write $e(x)$ for the value associated to x in e. Finally, we define the extension of e by v in x, written $e + x \mapsto v$, by:

$$\begin{aligned}
[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v \quad &= \quad [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v] \\
&\quad\quad \text{if } x \notin \{x_1, \ldots, x_n\} \\
[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v \quad &= \quad [x_1 \mapsto v_1, \ldots, x_{i-1} \mapsto v_{i-1}, x \mapsto v, x_{i+1} \mapsto v_{i+1}, \ldots, x_n \mapsto v_n] \\
&\quad\quad \text{if } x = x_i
\end{aligned}$$

Hence, we have as expected

$$\begin{aligned}
\mathrm{Dom}(e + x \mapsto v) \quad &= \quad \mathrm{Dom}(e) \cup \{x\} \\
(e + x \mapsto v)(x) \quad &= \quad x \\
(e + x \mapsto v)(y) \quad &= \quad e(y) \text{ for all } y \in \mathrm{Dom}(e),\ y \neq x.
\end{aligned}$$

1.2.2 Evaluation rules

Once these definitions are set up, we are able to define the evaluation judgement $e \vdash a \Rightarrow r$. The definition is presented as a set of axioms and inference rules. Directions for use: an axiom P allows to conclude that proposition P holds; an inference rule such as

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

allows to prove that the conclusion P holds as soon as all premises $P_1 \dots P_n$ have been proved. Axioms and rules can contain (meta-) variables. These variables are implicitly quantified universally at the beginning of each rule.

$$e \vdash cst \Rightarrow cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \Rightarrow e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \Rightarrow \texttt{err}}$$

The axiom on the left states that a constant evaluates to itself. The middle rule states that an identifier evaluates to the value bound to it in the environment e, provided that this identifier does belong to the domain of e. Otherwise, there is no sensible value for the expression x. In this case, the right rule therefore returns the response "an error has occurred", denoted by `err`.

$$e \vdash (f \;\texttt{where}\; f(x) = a) \Rightarrow (f, x, a, e)$$

A function evaluates to a CLOSURE: an object that combines the unevaluated body of a function (the triple f, x, a) with the environment e at the time the function was defined.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

A pair expression normally evaluates to a pair of values, in the obvious way. However, if the evaluation of one of the two pair components causes an error (such as evaluating an unbound identifier), then the evaluation of the whole pair also returns an error.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{let}\; x = a_1 \;\texttt{in}\; a_2 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash \texttt{let}\; x = a_1 \;\texttt{in}\; a_2 \Rightarrow \texttt{err}}$$

A `let` binding evaluates its first argument, then associates the value obtained to the bound identifier, and evaluates its second argument in the environment thus enriched. The result obtained is also the result for the whole `let` expression. In the case where the evaluation of the first argument causes an error, the evaluation of the `let` expression immediately stops on an error.

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r_0}{e \vdash a_1(a_2) \Rightarrow r_0}$$

The rule for applications $a_1(a_2)$ is the most complex. The expression a_1 must evaluate to a closure (f, x, a_0, e_0). The argument a_2 must evaluate to a value v_2. We then evaluate the function body a_0 from the closure in the environment e_0 from the closure, after having enriched e_0 by two bindings: the function parameter x is bound to the value v_2 of the argument; the function name f is bound to the closure (f, x, a_0, e_0) itself. The latter binding ensures that the evaluation of a_0, which can refer to f, will find the right value for f in the environment.

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash a_1(a_2) \Rightarrow \texttt{err}}$$

Here are the two error cases for function application. The application is $a_1(a_2)$ is meaningless if a_1 evaluates in a value other than a closure. Example: $1(2)$. The leftmost rule returns `err` in this case. The rightmost rule ensures that the response `err` propagates through applications: if the evaluation of the argument part stops on an error, the evaluation of the whole application also stops on an error, even if the function does not use its argument.

It remains to give evaluation rules for each of the primitives in the set `Op`. As an example, here are the rules for integer addition, and for the conditional construct:

$$\frac{e \vdash a \Rightarrow (cst_1, cst_2) \qquad cst_1 \in \texttt{Int} \qquad cst_2 \in \texttt{Int} \qquad cst_1 + cst_2 = cst}{e \vdash +(a) \Rightarrow cst}$$

$$\frac{e \vdash a \Rightarrow r \qquad r \text{ is not } (cst_1, cst_2) \text{ with } cst_1 \in \texttt{Int} \text{ and } cst_2 \in \texttt{Int}}{e \vdash +(a) \Rightarrow \texttt{err}}$$

$$\frac{e \vdash a_1 \Rightarrow \texttt{true} \qquad e \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{false} \qquad e \vdash a_3 \Rightarrow r_3}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow r_3}$$

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \notin \texttt{Bool}}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow \texttt{err}}$$

Here and elsewhere, we associate to these rules a relation between environments, expressions and values, written $e \vdash a \Rightarrow v$, which is the least defined relation that satisfies the inference rules. In the following, we frequently use the following characterization of this relation: (e, a, v) are in relation by $\cdot \vdash \cdot \Rightarrow \cdot$ if and only if there exists a finite derivation of the judgement $e \vdash a : v$ from the axioms, by repeated applications of the inference rules [73].

Context. Introducing the `err` response and the rules that return this response (the so-called "error rules") is the traditional answer to the problem of distinguishing erroneous programs from non-terminating programs: without error rules, when an expression does not admit any (finite) evaluation derivation, we do not know whether this is due to the fact that at some point no regular evaluation rule match (case of a run-time type error), or to the fact that the only possible derivations are infinite (case of a non-terminating program).

Besides complicating the semantics, the error rules put unfortunate constraints over the evaluation strategy — which should not interfere at this level of description. For instance, the two error rules for pairs proposed above:

$$\frac{e \vdash a_1 \Rightarrow \mathtt{err}}{e \vdash (a_1, a_2) \Rightarrow \mathtt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \mathtt{err}}{e \vdash (a_1, a_2) \Rightarrow \mathtt{err}}$$

imply the parallel evaluation of the two pair components ("angelic non-determinism"), since we must return an error in all cases where one component does not terminate while the other causes an error. To allow sequential implementation, we must break the symmetry between the two error rules, and take for instance:

$$\frac{e \vdash a_1 \Rightarrow \mathtt{err}}{e \vdash (a_1, a_2) \Rightarrow \mathtt{err}} \qquad \frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow \mathtt{err}}{e \vdash (a_1, a_2) \Rightarrow \mathtt{err}}$$

But, then, we have imposed the evaluation order for the pair: left to right. Alternate, more recent approaches that do not require error rules avoid this difficulty. These approaches make use of richer notions of derivations: partial derivations [32], or possibly infinite derivations [20].

This indirect specification of the evaluation order through error rules is less of a problem for statically-typed languages: when we evaluate well-typed terms only, the error cases never occur at run-time; hence, we can adopt an evaluation strategy that does not respect the error rules. That's why I indulged in non-sequential error rules for pairs and function applications: these rules are simpler and more elegant than the sequential rules. □

1.3 Typing

We are now going to equip the language with a type system. The typing rules associate types (more precisely, type expressions) to expressions, in the same way as the evaluation rules associate evaluation results to expressions. We use Milner's type system [62, 21], which is the basis for the type systems of the ML language and several functional languages (Miranda, Hope, Haskell). The distinctive feature of Milner's system is polymorphism, taking into account the fact that an expression can belong to several different types.

1.3.1 Types

We assume given a set `TypBas` of BASE TYPES (such as `int`, the type of integers, and `bool`, the type of truth values), and an infinite set `VarTyp` of TYPE VARIABLES.

$$\begin{array}{rcll} \iota & \in & \mathtt{TypBas} = \{\mathtt{int};\mathtt{bool};\ldots\} & \text{base types} \\ \alpha,\beta,\gamma & \in & \mathtt{VarTyp} & \text{type variables} \end{array}$$

We define the set `Typ` of TYPE EXPRESSIONS (or simply TYPES), with typical element τ, by the following grammar:

$$\begin{array}{rcll} \tau & ::= & \iota & \text{base type} \\ & | & \alpha & \text{type variable} \\ & | & \tau_1 \to \tau_2 & \text{function type} \\ & | & \tau_1 \times \tau_2 & \text{product type} \end{array}$$

We write $\mathcal{F}(\tau)$ for the set of type variables that are free in the type τ. This set is defined by:

$$\begin{aligned}
\mathcal{F}(\iota) &= \emptyset \\
\mathcal{F}(\alpha) &= \{\alpha\} \\
\mathcal{F}(\tau_1 \to \tau_2) &= \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\
\mathcal{F}(\tau_1 \times \tau_2) &= \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)
\end{aligned}$$

1.3.2 Substitutions

The substitutions considered here are finite mappings from type variables to type expressions. They are written φ, ψ.

$$\text{Substitutions:} \quad \varphi, \psi \quad ::= \quad [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$$

A substitution φ naturally extends to an homomorphism of type expressions, written $\overline{\varphi}$, and defined by:

$$\begin{aligned}
\overline{\varphi}(\iota) &= \iota \\
\overline{\varphi}(\alpha) &= \varphi(\alpha) \text{ if } \alpha \in \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\alpha) &= \alpha \text{ if } \alpha \notin \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\tau_1 \to \tau_2) &= \overline{\varphi}(\tau_1) \to \overline{\varphi}(\tau_2) \\
\overline{\varphi}(\tau_1 \times \tau_2) &= \overline{\varphi}(\tau_1) \times \overline{\varphi}(\tau_2)
\end{aligned}$$

From now on, we do not distinguish anymore between φ and its extension $\overline{\varphi}$, and we write φ for both.

The following property shows the effect of a substitution over the free variables of a type.

Proposition 1.1 (Substitution and free variables) *For all types τ and all substitutions φ, we have:*

$$\mathcal{F}(\varphi(\tau)) = \bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha))$$

Proof: straightforward structural induction over τ. □

1.3.3 Type schemes

We define the set `SchTyp` of TYPE SCHEMES, with typical element σ, by the following grammar:

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n . \tau \qquad \text{type scheme}$$

In this syntax, the quantified variables $\alpha_1 \ldots \alpha_n$ are treated as a set of variables: their relative order is not significant, and they are assumed to be distinct. We do not distinguish between the type τ and the trivial type scheme $\forall . \tau$, and we write τ for both. We identify two type schemes that differ only by a renaming of the variables bound by $\forall$ (alpha-conversion operation), and by the introduction or suppression of quantified variables that are not free in the type part. More precisely, we quotient the set of schemes by the following two equations:

$$\begin{aligned}\forall \alpha_1 \ldots \alpha_n . \tau &= \forall \beta_1 \ldots \beta_n . [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau) \\ \forall \alpha \alpha_1 \ldots \alpha_n . \tau &= \forall \alpha_1 \ldots \alpha_n . \tau \quad \text{if } \alpha \notin \mathcal{F}(\tau)\end{aligned}$$

The free variables in a type scheme are:

$$\mathcal{F}(\forall \alpha_1 \ldots \alpha_n . \tau) = \mathcal{F}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}.$$

This definition is compatible with the two equations over types schemes, as can easily be checked using proposition 1.1.

We extend substitutions to type schemes as follows:

$$\varphi(\forall \alpha_1 \ldots \alpha_n . \tau) = \forall \alpha_1 \ldots \alpha_n . \varphi(\tau),$$

assuming, after renaming the α_i if necessary, that the α_i are OUT OF REACH for φ; that is, none of the α_i is in the domain of φ, and none of the α_i is free in one of the types in the range of φ. We easily check that this definition is compatible with the two equations over schemes.

Proposition 1.1 also holds with the type τ replaced by a scheme σ.

1.3.4 Typing environments

A typing environment, written E, is a finite mapping from identifiers to type schemes:

$$E \quad ::= \quad [x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$$

The image $\varphi(E)$ of an environment E by a substitution φ is defined in the obvious way:

$$\varphi([x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]) = [x_1 \mapsto \varphi(\sigma_1), \ldots, x_n \mapsto \varphi(\sigma_n)].$$

The operator $\mathcal{F}$ is extended to typing environment, by taking that a type variable is free in E if and only if it is free in $E(x)$ for some x:

$$\mathcal{F}(E) = \bigcup_{x \in \mathrm{Dom}(E)} \mathcal{F}(E(x))$$

1.3.5 Typing rules

We are now going to give the typing rules for the language, using the same formalism as for the evaluation rules. The following inference rules define the TYPING JUDGEMENT $E \vdash a : \tau$, read: "in environment E, the expression a has type τ". The environment E associates a type scheme to each identifier that can appear in expression a.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

The typing rule for identifiers states that an identifier x in an expression can be given any type that is an instance of the type scheme associated to x in the typing environment. We say that the type τ is an INSTANCE of the type scheme $\sigma = \forall\alpha_1 \ldots \alpha_n.\, \tau_0$, and we write $\tau \leq \sigma$, if and only if there exists a substitution φ whose domain is a subset of $\{\alpha_1 \ldots \alpha_n\}$ such that τ is equal to $\varphi(\tau_0)$. The relation $\tau \leq \sigma$ is obviously compatible with the two equations over schemes (alpha-conversion, and useless quantified variable elimination).

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2} \qquad \frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

A function definition has type $\tau_1 \to \tau_2$ as soon as the function body has type τ_2, under the extra assumptions that its formal parameter x has type τ_1, and its internal name f has type $\tau_1 \to \tau_2$. A function application is well-typed as soon as the type of the argument corresponds to the type of the function parameter; the type of the function result is the type of the whole application node.

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2}$$

The `let` construct is the only one that introduces polymorphic types (that is, non-trivial type schemes) in the environment. This is due to the generalization operator `Gen`, which builds a type scheme from a type and a type environment, as follows:

$$\texttt{Gen}(\tau, E) = \forall\alpha_1 \ldots \alpha_n.\, \tau \quad \text{where} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E).$$

We shall see later that if a_1 belongs to the type τ under the assumptions E, and if α is not free in E, then a_1 also belongs to the type $[\alpha \mapsto \tau_1](\tau)$ for all types τ_1. This intuitively explains why α can be universally quantified, since it can be substituted by any type.

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \qquad \frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

The typing of pair expressions is obvious. For constants and primitives, we assume given two functions $\texttt{TypCst} : \texttt{Cst} \to \texttt{SchTyp}$ and $\texttt{TypOp} : \texttt{Op} \to \texttt{SchTyp}$, that assign type schemes to constants and operators. For instance, we can take:

$$\begin{array}{rcll}
\texttt{TypCst}(i) & = & \texttt{int} & (i = 0, 1, \ldots) \\
\texttt{TypCst}(b) & = & \texttt{bool} & (b = \texttt{true} \text{ or } \texttt{false}) \\
\texttt{TypOp}(+) & = & \texttt{int} \times \texttt{int} \to \texttt{int} & \\
\texttt{TypOp}(\texttt{ifthenelse}) & = & \forall\alpha.\ \texttt{bool} \times \alpha \times \alpha \to \alpha &
\end{array}$$

Example. The expression `let id = (f where f(x) = x) in id(1)` has type `int`. That's because the following derivation is valid:

$$\dfrac{\dfrac{\dfrac{t \le t}{[\mathtt{f} \mapsto t \to t,\, \mathtt{x} \mapsto t] \vdash \mathtt{x} : t}}{[\,] \vdash \mathtt{f\ where\ f(x) = x} : t \to t} \qquad \dfrac{\dfrac{\mathtt{int} \to \mathtt{int} \le \forall t.\, t \to t}{E \vdash \mathtt{id} : \mathtt{int} \to \mathtt{int}} \qquad \dfrac{\mathtt{int} \le \mathtt{int}}{E \vdash 1 : \mathtt{int}}}{E = [\mathtt{id} \mapsto \forall t.\, t \to t] \vdash \mathtt{id(1)} : \mathtt{int}}}{[\,] \vdash \mathtt{let\ id = (f\ where\ f(x) = x)\ in\ id(1)} : \mathtt{int}}$$

□

Context. The presentation given above is essentially the one of the definition of Standard ML [59], and slightly differs from Damas and Milner's presentation [21]. The Damas-Milner system assigns type schemes to expressions, not simple types, and features two separate rules for generalization and instantiation. In the presentation above, these two operations are integrated into the `let` rule (for generalization) and into the rules for identifiers and constants (for instantiation). The presentation above has one advantage: the rules are syntax-directed — that is, the shape of the expression determines the unique rule that applies; and the premises contain only strict subexpressions of the expression in the conclusion. This makes many proofs easier, especially those for the type inference algorithm. □

1.3.6 Properties of the typing rules

As we said above about the typing rule for `let`, the typing judgement is stable under substitution: if we can prove $E \vdash a : \tau$, then the judgements obtained by substituting any types for any variables in E and τ are still provable.

Proposition 1.2 (Typing is stable under substitution) *Let a be an expression, τ be a type, E be a typing environment and φ be a substitution. If $E \vdash a : \tau$, then $\varphi(E) \vdash a : \varphi(\tau)$.*

Proof: by structural induction on a. We show the two cases that do not immediately follow from the induction hypothesis.

• **Case** $a = x$. We must have $\tau \le E(x)$, with $E(x) = \forall \alpha_1 \ldots \alpha_n .\, \tau_0$. After renaming if necessary, we can assume that the α_i are out of reach for φ. Let ψ be a substitution over the α_i such that $\tau = \psi(\tau_0)$. Define a substitution θ with domain $\{\alpha_1 \ldots \alpha_n\}$ by $\theta(\alpha_i) = \varphi(\psi(\alpha_i))$. We have:

$$\begin{array}{rclcll} \theta(\varphi(\alpha_i)) & = & \theta(\alpha_i) & = & \varphi(\psi(\alpha_i)) & \text{for all } i \\ \theta(\varphi(\beta)) & = & \varphi(\beta) & = & \varphi(\psi(\beta)) & \text{for all } \beta \text{ that is not an } \alpha_i \end{array}$$

Hence $\theta(\varphi(\tau_0)) = \varphi(\psi(\tau_0)) = \varphi(\tau)$, showing that $\varphi(\tau)$ is an instance of $\varphi(E(x))$. We can therefore derive $\varphi(E) \vdash x : \varphi(\tau)$.

• **Case** $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$. The typing derivation ends up with:

$$\dfrac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \mathtt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

By definition, $\texttt{Gen}(\tau_1, E)$ is equal to $\forall\alpha_1 \ldots \alpha_n.\, \tau_1$, with $\{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{F}(E)$. Let $\beta_1, \ldots, \beta_n$ be type variables out of reach for φ, and not free in E. Let ψ be the substitution $\varphi \circ [\alpha_i \mapsto \beta_i]$. We have $\psi(E) = \varphi(E)$, since the variables α_i are not free in E.

We apply the induction hypothesis twice: once to the left premise, with the substitution ψ; once to the right premise, with the substitution φ. We obtain proofs of:

$$\psi(E) \vdash a_1 : \psi(\tau_1) \qquad\qquad \varphi(E) + x \mapsto \varphi(\texttt{Gen}(\tau_1, E)) \vdash a_2 : \varphi(\tau_2)$$

We now compute $\texttt{Gen}(\psi(\tau_1), \psi(E))$, using proposition 1.1.

$$\mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E)) = \left(\bigcup_{\alpha \in \mathcal{F}(\tau_1)} \mathcal{F}(\psi(\alpha)) \right) \setminus \left(\bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha)) \right)$$

By construction of ψ, we have $\psi(\alpha_i) = \varphi(\beta_i) = \beta_i$. Moreover, for all variables α that are not an α_i, the term $\psi(\alpha)$ is equal to $\varphi(\alpha)$ and does not contain any of the β_i. Since the α_i are free in τ_1, but not free in E, we therefore have:

$$\beta_i \in \bigcup_{\alpha \in \mathcal{F}(\tau_1)} \mathcal{F}(\psi(\alpha)) \qquad\qquad \beta_i \notin \bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha)).$$

Hence $\{\beta_1 \ldots \beta_n\} \subseteq \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$. We now show the converse inclusion. Let $\beta \in \mathcal{F}(\psi(\tau_1))$, such that β is not one of the β_i. Let $\alpha \in \mathcal{F}(\tau_1)$ such that $\beta \in \mathcal{F}(\psi(\alpha))$ (the existence of α follows from proposition 1.1). Necessarily, α is not one of the α_i; otherwise, β would be one of the β_i. Hence $\alpha \in \mathcal{F}(E)$, and $\beta \in \bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha))$ and $\beta \notin \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$. Hence $\{\beta_1 \ldots \beta_n\} = \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$, and

$$\texttt{Gen}(\psi(\tau_1), \psi(E)) = \forall\beta_1 \ldots \beta_n.\, \psi(\tau_1) = \varphi(\forall\alpha_1 \ldots \alpha_n.\, \tau) = \varphi(\texttt{Gen}(\tau_1, E))$$

by definition of the image of a scheme by a substitution. Since $\psi(E)$ and $\varphi(E)$ are identical, the two derivations obtained by applying the induction hypothesis therefore prove that:

$$\varphi(E) \vdash a_1 : \psi(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\psi(\tau_1), \varphi(E)) \vdash a_2 : \varphi(\tau_2).$$

Applying the `let` typing rule to these premises, we obtain the desired result:

$$\varphi(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2).$$

□

Another property of the typing rules, relevant to the type inference algorithm: if we can prove that a has type τ under the assumptions E, then we can prove it under stronger assumptions E'. To make this notion of "strength" more precise, we take that a scheme σ' is MORE GENERAL than a scheme σ, and we write $\sigma' \geq \sigma$, if all instances of σ are also instances of σ'.

Proposition 1.3 *Let E, E' be two typing environments such that* $\mathrm{Dom}(E) = \mathrm{Dom}(E')$, *and* $E'(x) \geq E(x)$ *for all* $x \in \mathrm{Dom}(E)$. *If* $E \vdash a : \tau$, *then* $E' \vdash a : \tau$.

Proof: easy structural induction on a. The base case $a = x$ is straightforward by hypothesis over E and E'. For the case $a = (\texttt{let}\ x = a_1\ \texttt{in}\ a_2)$, we first show that $\mathcal{F}(E') \subseteq \mathcal{F}(E)$, implying $\texttt{Gen}(\tau_1, E') \geq \texttt{Gen}(\tau_1, E)$, where τ_1 is the type of a_1. We can therefore apply the induction hypothesis to the typing derivation for a_2; the result follows. □

1.4 Type soundness

One of the main goals of static typing is to exclude the programs that can cause a type error at run-time. We have defined the typing rules independently from the evaluation rules. It therefore remains to show that the initial goal is reached, that is, that the typing rules are *sound* with respect to the evaluation rules. The main result in this section is the following: no closed, well-typed expression can evaluate to `err`.

Proposition 1.4 (Weak soundness) *Let a be an expression, τ be a type, and r be a response. If $[\,] \vdash a : \tau$ and $[\,] \vdash a \Rightarrow r$, then r is not* `err`.

Context. This claim does not state that there always exists a value v to which a evaluates: a well-typed program may fail to terminate. The aim of a type system for a programming language is not to ensure termination, but to guarantee the structural conformity of data. □

It is difficult to prove directly the weak soundness claim, because it does not lend itself to an inductive proof: to show that $a_1(a_2)$ cannot evaluate to `err`, it is not enough to show that neither a_1 nor a_2 can evaluate to `err`; even then, the application $a_1(a_2)$ can cause a run-time type error, for instance if a_1 does not evaluate to a closure. We are therefore going to show a stronger result: the evaluation of a closed term with type τ, if it terminates, not only does not return `err`, but also returns a value that, semantically, does belong to the type τ. For instance, an expression with type `int` can only evaluate to an integer value.

1.4.1 Semantic typing

Before going any further, we must therefore define precisely what it means for a value to semantically belong to a type. We define the following three semantic typing predicates:

$\models v : \tau$	the value v belongs to the type τ
$\models v : \sigma$	the value v belongs to the scheme σ
$\models e : E$	the values contained in the evaluation evironment e belong to the corresponding type schemes in E

These predicates are defined as follows:

- $\models cst :$ `int` if cst is an integer
- $\models cst :$ `bool` if cst is `true` or `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$
- $\models (f, x, a, e) : \tau_1 \to \tau_2$ if there exists a typing environment E such that

$$\models e : E \quad \text{and} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2$$

- $\models v : \sigma$ if for all types τ such that $\tau \le \sigma$, we have $\models v : \tau$
- $\models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, and for all $x \in \mathrm{Dom}(e)$, we have $\models e(x) : E(x)$.

This definition is well-founded by structural induction over the value part v. In all cases except for schemes, the definition of $\models$ over the value v appeals to $\models$ over proper subterms of v. The case for type schemes appeals to $\models$ on the same value, but with the type scheme replaced by a simple type; hence, another case of the definition is going to be used just after, and this case considers only proper subterms of v.

Context. Appealing to the type judgement to define $\models$ over functional values is a technical trick due to Tofte [92, 93]. A more traditional definition would be "a functional value belongs to the type $\tau_1 \rightarrow \tau_2$ if it maps any value of type τ_1 to a value of type τ_2". More formally: $\models (f, x, a, e) : \tau_1 \rightarrow \tau_2$ if, for all values v_1 and results r_2 such that

$$\models v_1 : \tau_1 \quad \text{and} \quad e + f \mapsto (f, x, a, e) + x \mapsto v_1 \vdash a \Rightarrow r_2,$$

we have $r_2 \neq \texttt{err}$ and $\models r_2 : \tau_2$. That's the continuity condition that appears, for instance, in ideal models [55]. However, such a definition is problematic in the presence of recursive functions: the soundness proof cannot be proved by simple induction, because in the case of a `where` construct we cannot conclude $\models (f, x, a, e) : \tau_1 \rightarrow \tau_2$ unless we have already proved that $\models (f, x, a, e) : \tau_1 \rightarrow \tau_2$. It seems that we would have to revert to more complex proof principles, such as co-induction [61], or to richer mathematical frameworks, such as domains [33]. By appealing to the typing judgement for defining the semantic typing relation over functions, we are back to an elementary proof by induction. Moreover, the semantic typing relation defined in terms of the typing judgement turns out to be stable under substitution of type variables (proposition 1.5 below), which is not the case with the classical relation defined over functions by continuity. This stability property is important for the soundness proof. □

1.4.2 Semantic generalization

The following proposition is the key lemma that shows the soundness of the `let` typing rule.

Proposition 1.5 *Let v be a value and τ be a type such that $\models v : \tau$. Then, for all substitutions φ, we have $\models v : \varphi(\tau)$. As a consequence, for all sets of variables $\alpha_1 \ldots \alpha_n$, we have $\models v : \forall \alpha_1 \ldots \alpha_n . \tau$.*

Proof: by structural induction over v (the same induction as in the definition of the $\models$ relation).

• **Case $v = cst$ and $\tau =$ `int` ou $\tau =$ `bool`.** Obvious, since $\varphi(\tau) = \tau$ in this case.

• **Case $v = (v_1, v_2)$ and $\tau = \tau_1 \times \tau_2$.** We apply the induction hypothesis to v_1 and v_2. It follows $\models v_1 : \varphi(\tau_1)$ and $\models v_2 : \varphi(\tau_2)$. Hence the result.

• **Case $v = (f, x, a, e)$ and $\tau = \tau_1 \rightarrow \tau_2$.** Let E be a typing environment such that $\models e : E$ and $E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \rightarrow \tau_2$. By proposition 1.2, we have

$$\varphi(E) \vdash (f \ \texttt{where} \ f(x) = a) : \varphi(\tau_1 \rightarrow \tau_2).$$

It remains to show that $\models e : \varphi(E)$. Let $y \in \text{Dom}(e)$. Write $E(y)$ as $\forall \alpha_1 \ldots \alpha_n . \tau_y$, with the α_i chosen out of reach for φ. We then have $\varphi(E(y)) = \forall \alpha_1 \ldots \alpha_n . \varphi(\tau_y)$. We need to show that $\models e(y) : \tau'$ for all instances τ' of $\varphi(E(y))$. Let τ' be such an instance. We therefore have $\tau' = \psi(\varphi(\tau_y))$

for some substitution ψ. We also have $\models e(y) : \tau_y$, by definition of $\models$ over type schemes. We can therefore apply the induction hypothesis to the value $e(y)$, to the type τ_y and to the substitution $\psi \circ \varphi$. It follows that $\models e(y) : \tau'$. This holds for all instances τ' of $\varphi(E(y))$. Hence $\models e(y) : \varphi(E(y))$. This holds for all $y \in \mathrm{Dom}(e)$. Hence $\models e : \varphi(E)$, and the expected result. □

1.4.3 Soundness proof

We are now able to precisely state and to prove the soundness of typing:

Proposition 1.6 (Strong soundness) *Let a be an expression, τ be a type, E be a typing environment, and e be an evaluation environment such that $E \vdash a : \tau$ and $\models e : E$. If there exists a result r such that $e \vdash a \Rightarrow r$, then $r \neq$ `err`; instead, r is a value v such that $\models v : \tau$.*

Proof: the proof proceeds by induction over the size of the evaluation derivation. We argue by case analysis on a, and therefore on the last rule used in the typing derivation.

- **Constants.**

$$\frac{\tau \leq \mathtt{TypCst}(cst)}{E \vdash cst : \tau}$$

The only evaluation possibility is $e \vdash cst \Rightarrow cst$. It remains to check that $\models cst : \mathtt{TypCst}(cst)$ for all constants cst. This is true for the set of constants and the type assignment `TypCst` given as examples above.

- **Variables.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

The typing guarantees that x belongs to the domain of E, which is also the domain of e by hypothesis $\models e : E$. Hence, the only evaluation possibility is $e \vdash x \Rightarrow e(x)$. By hypothesis $\models e : E$, we have $\models e(x) : E(x)$. Hence $\models e(x) : \tau$ by definition of $\models$ over type schemes.

- **Functions.**

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \mathtt{where} \ f(x) = a) : \tau_1 \to \tau_2}$$

The only evaluation possibility is $e \vdash (f \ \mathtt{where} \ f(x) = a) \Rightarrow (f, x, a, e)$. We have $\models (f, x, a, e) : \tau_1 \to \tau_2$ by definition of $\models$, taking E for the required typing environment.

- **Applications.**

$$\frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

We have three evaluation possibilities. The first one concludes $r = \mathtt{err}$ because $e \vdash a_1 \Rightarrow r_1$ where r_1 is not a closure; but this contradicts the induction hypothesis applied to a_1, which shows $\models r_1 : \tau_2 \to \tau_1$, hence r_1 is a fortiori a closure. The second evaluation leads to $r = \mathtt{err}$ because

$e \vdash a_2 \Rightarrow$ `err`; it similarly contradicts the induction hypothesis applied to a_2. Hence the last evaluation step must be:

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r}{e \vdash a_1(a_2) \Rightarrow r}$$

By induction hypothesis, we know that $\models (f, x, a_0, e_0) : \tau_2 \to \tau_1$, and $\models v_2 : \tau_2$. Hence there exists a typing environment E_0 such that $\models e_0 : E_0$ and $E_0 \vdash (f \texttt{ where } f(x) = a_0) : \tau_2 \to \tau_1$. There is only one typing rule that can derive this result; its premise must therefore hold:

$$E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1.$$

Consider the environments:

$$e_1 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \quad \text{and} \quad E_1 = E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2.$$

We have $\models e_1 : E_1$ and $E_1 \vdash a_0 : \tau_1$. Applying the induction hypothesis to the evaluation of a_0, we get $r \neq$ `err` and $\models r : \tau_1$, which is the expected result.

- **`let` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

Two evaluations are possible. The first one is $e \vdash a_1 \Rightarrow$ `err`. It contradicts the induction hypothesis applied to a_1. Hence the last step in the evaluation must be:

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r}{e \vdash \texttt{let } x = a_1 \texttt{ in } a_2 \Rightarrow r}$$

By the induction hypothesis applied to a_1, we get $\models v_1 : \tau_1$. By proposition 1.5, it follows $\models v_1 : \texttt{Gen}(\tau_1, E)$. Define

$$e_1 = e + x \mapsto v_1 \quad \text{and} \quad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E).$$

We therefore have $\models e_1 : E_1$. We apply the induction hypothesis to a_2 considered in the environments e_1 and E_1. The expected result follows: $r \neq$ `err` and $\models r : \tau_2$.

- **Pairs.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

Same reasoning as for function application, just simpler.

- **Primitive applications.**

$$\frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

Applying the induction hypothesis, we get $e \vdash a \Rightarrow v_1$ with $\models v_1 : \tau_1$. The remainder of the proof depends on the types and semantics given to the operators. It is easy to check the result in the case of the integer addition or the conditional construct, the two examples given above. □

The weak soundness property (proposition 1.4) is an immediate corollary of the strong soundness property (proposition 1.6).

1.5 Type inference

We have seen that if a closed expression a has type τ, it also has type τ' for all types τ' less general than τ (that is, $\tau' = \varphi(\tau)$ for some substitution φ). We are now going to show that the set of all possible types for a is generated this way: there exists a type for a, called the PRINCIPAL TYPE for a, that is more general than all possible types for a. The computation of this type is called TYPE INFERENCE. The existence of a type inference algorithm allows ML compilers to reconstruct the types of all objects from their uses, without the programmer providing type information in the source code.

We first recall some definitions about principal unifiers. A substitution φ is said to be a UNIFIER for the two types τ_1 and τ_2 if $\varphi(\tau_1) = \varphi(\tau_2)$. Two types are UNIFIABLE if there exists a unifier for those two types. Intuitively, two types are unifiable if they can be identified by instantiating some of their variables. A unifier for two types represents the instantiations that must be performed in order to identify them. A unifier φ for τ_1 and τ_2 is said to be MOST GENERAL if all unifiers ψ for τ_1 and τ_2 can be decomposed as $\theta \circ \varphi$ for some substitution θ. The most general unifier for two types, when it exists, represents the minimal modifications required to identify the types: all other unifiers perform at least those modifications.

Proposition 1.7 *If two types τ_1 and τ_2 are unifiables, then they admit a most general unifier, unique up to a renaming, that we write* $\mathtt{mgu}(\tau_1, \tau_2)$.

Proof: the set of types is a free term algebra. The most general unifier can be computed with Robinson's algorithm [85], or one of its variants. □

Among the most general unifiers for the two types τ_1 and τ_2, there exist some that "do not introduce fresh variables", that is, such that all variables not free in τ_1 nor in τ_2 are out of reach for these unifiers. (This is the case for the unifier built by Robinson's algorithm.) In the following, we assume that $\mathtt{mgu}(\tau_1, \tau_2)$ is chosen so as to satisfy this property.

We now give a variant of Damas and Milner's algorithm [21], that computes the principal type for an expression, if it exists, or fails otherwise. The algorithm takes as inputs an expression a, an initial typing environment E, and a set of "fresh" variables V; it returns as results a type τ (the most general type for a), a substitution φ (representing the instantiations that had to be performed over E), and a subset V' of V (the unused fresh variables).

We write $\mathtt{inst}(\sigma, V)$ for a trivial instance of the scheme σ. That is, writing $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$, we take n distinct variables $\beta_1 \ldots \beta_n$ in V and we define

$$\mathtt{inst}(\sigma, V) = ([\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau),\ V \setminus \{\beta_1 \ldots \beta_n\}).$$

Clearly, $\mathtt{inst}(\sigma, V)$ is uniquely defined up to a renaming of variables from V into variables from V.

Algorithm 1.1 $\texttt{infer}(E, a, V)$ is the triple (τ, φ, V') defined by:

If a is x and $x \in \mathrm{Dom}(E)$:
$\quad(\tau, V') = \texttt{inst}(E(x), V)$ and $\varphi = [\,]$
If a is cst:
$\quad(\tau, V') = \texttt{inst}(\texttt{TypCst}(cst), V)$ and $\varphi = [\,]$
If a is $(f\ \texttt{where}\ f(x) = a_1)$:
$\quad$let α and β be two variables from V
$\quad$let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1,\ E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha,\ V \setminus \{\alpha, \beta\})$
$\quad$let $\mu = \texttt{mgu}(\varphi_1(\beta), \tau_1)$
$\quad$then $\tau = \mu(\varphi_1(\alpha \to \beta))$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_1$
If a is $a_1(a_2)$:
$\quad$let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
$\quad$let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2, \varphi_1(E), V_1)$
$\quad$let α be a variable from V_2
$\quad$let $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \to \alpha)$
$\quad$then $\tau = \mu(\alpha)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $V' = V_2 \setminus \{\alpha\}$
If a is $\texttt{let}\ x = a_1\ \texttt{in}\ a_2$:
$\quad$let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
$\quad$let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2,\ \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$
$\quad$then $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V = V_2$
If a is (a_1, a_2):
$\quad$let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
$\quad$let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2, \varphi_1(E), V_1)$
$\quad$then $\tau = \tau_1 \times \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$
If a is $op(a_1)$:
$\quad$let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
$\quad$let $(\tau_2, V_2) = \texttt{inst}(\texttt{TypOp}(op), V_1)$
$\quad$let α be a variable from V_2
$\quad$let $\mu = \texttt{mgu}(\tau_1 \to \alpha, \tau_2)$
$\quad$then $\tau = \mu(\alpha)$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_2 \setminus \{\alpha\}$

We take that $\texttt{infer}(a, E, V)$ is not defined if at some point none of the cases match; in particular, if we try to unify two non-unifiable types.

Proposition 1.8 (Correctness of type inference) *Let a be an expression, E be a typing environment and V be a set of type variables. If $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, then we can derive $\varphi(E) \vdash a : \tau$.*

Proof: the proof is an inductive argument over the structure of a, and makes heavy use of the fact that the typing judgement is stable under substitution (proposition 1.2). We show one base step and two inductive steps; the remaining cases are similar. We use the same notations as in the algorithm.

- **Case $a = x$.** We have $(\tau, V') = \texttt{inst}(E(x), V)$ and $\varphi = [\,]$. By definition of $\texttt{inst}$, we have $\tau \leq E(x)$. We can therefore derive $E \vdash x : \tau$.

• **Case** $a = (f \texttt{ where } f(x) = a_1)$. Applying the induction hypothesis to the recursive call to `infer`, we get a derivation of

$$\varphi_1(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \tau_1.$$

By proposition 1.2, we get a derivation of:

$$\varphi(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \mu(\tau_1).$$

The substitution μ being by definition a unifier of $\varphi_1(\beta)$ and τ_1, we have $\varphi(\beta) = \psi(\tau_1)$. Hence we have shown that:

$$\varphi(E) + f \mapsto (\varphi(\alpha) \to \varphi(\beta)) + x \mapsto \varphi(\alpha) \vdash a_1 : \varphi(\beta).$$

Applying the typing rule for functions, we get:

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a_1) : \varphi(\alpha) \to \varphi(\beta).$$

That's the expected result, since $\tau = \varphi(\alpha \to \beta)$.

• **Case** $a = \texttt{let } x = a_1 \texttt{ in } a_2$. We apply the induction hypothesis to the two recursive calls to `infer`. We get derivations for:

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \varphi_2(\varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau_2.$$

If required, we rename the generalized variables in the left derivation so that they are out of reach for φ_2. We then show that

$$\texttt{Gen}(\varphi_2(\tau_1), \varphi_2(\varphi_1(E))) = \varphi_2(\texttt{Gen}(\tau_1, \varphi_1(E)))$$

as in the proof of proposition 1.2, `let` case. Taking $\varphi = \varphi_2 \circ \varphi_1$, we therefore have derived:

$$\varphi(E) \vdash a_1 : \varphi_2(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\varphi_2(\tau_1), \varphi(E)) \vdash a_2 : \tau_2.$$

We conclude, by the `let` typing rule,

$$\varphi(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2.$$

That's the expected result. □

Proposition 1.9 (Completeness of type inference) *Let a be an expression, E be a typing environment, and V be an infinite set of variables such that $V \cap \mathcal{F}(E) = \emptyset$. If there exists a type τ' and a substitution φ' such that $\varphi'(E) \vdash a : \tau'$, then $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, and there exists a substitution ψ such that*

$$\tau' = \psi(\tau) \quad \textit{and} \quad \varphi' = \psi \circ \varphi \textit{ outside } V.$$

By "$\varphi' = \psi \circ \varphi$ OUTSIDE V", we mean that $\varphi'(\alpha)$ is equal to $\psi(\varphi(\alpha))$ for all variables α that do not belong to V. The condition $\varphi' = \psi \circ \varphi$ outside V captures the fact that the two substitutions φ' and $\psi \circ \varphi$ behave the same on the initial typing problem. The variables in V are variables that do not appear in the initial problem, but can be introduced at intermediate steps during inference; they do not have to be taken into account when comparing φ' (the proposed solution to the typing problem) with φ (the inferred solution).

Proof: we first remark that, under the assumptions of the proposition, if $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, then $V' \subseteq V$, and the variables in V' are not free in τ and are out of reach for φ. Hence, $V' \cap \mathcal{F}(\varphi(E)) = \emptyset$.

The proof of proposition 1.9 proceeds by structural induction over a. We show one base step and three inductive steps; the remaining cases are similar.

• **Case** $a = x$. Since $\varphi(E) \vdash x : \tau$, we have $x \in \mathrm{Dom}(\varphi(E))$ and $\tau \leq \varphi(E)(x)$. This implies $x \in \mathrm{Dom}(E)$. Hence $\texttt{infer}(E, x)$ is defined, and returns

$$\tau = [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau_x) \quad \text{and} \quad \varphi = [\,] \quad \text{and} \quad V' = V \setminus \{\beta_1 \ldots \beta_n\}$$

for some variables $\beta_1 \ldots \beta_n \in V$. Write $E(x) = \forall \alpha_1 \ldots \alpha_n.\, \tau_x$, where the α_i are taken from V' and out of reach for φ'. We have $\varphi'(E(x)) = \forall \alpha_1 \ldots \alpha_n.\, \varphi'(\tau_x)$. Take ρ to be the substitution over the α_i such that $\tau' = \rho(\varphi'(\tau_x))$. Take

$$\psi = \rho \circ \varphi' \circ [\beta_1 \mapsto \alpha_1, \ldots, \beta_n \mapsto \alpha_n].$$

We have $\psi(\tau) = \rho(\varphi'(\tau_x)) = \tau'$. Moreover, all variables $\alpha \notin V$ are neither one of the α_i, nor one of the β_i, hence $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. This is the expected result, since $\varphi = [\,]$ here.

• **Case** $a = (f\ \texttt{where}\ f(x) = a_1)$. The initial typing derivation ends up with

$$\frac{\varphi'(E) + f \mapsto (\tau_2' \to \tau_1') + x \mapsto \tau_2' \vdash a_1 : \tau_1'}{\varphi'(E) \vdash (f\ \texttt{where}\ f(x) = a) : \tau_2' \to \tau_1'}$$

Choose α and β in V, as in the algorithm. Define the environment E_1 and the substitution φ_1' by

$$E_1 = E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha \quad \text{and} \quad \varphi_1' = \varphi' + \alpha \mapsto \tau_2' + \beta \mapsto \tau_1'.$$

We have $\varphi_1'(E_1) = \varphi'(E) + f \mapsto (\tau_2' \to \tau_1') + x \mapsto \tau_2'$. We apply the induction hypothesis to a_1, E_1, $V \setminus \{\alpha, \beta\}$, φ_1' and τ_2'. We get

$$(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E_1, V \setminus \{\alpha, \beta\}) \quad \text{and} \quad \tau_1' = \psi_1(\tau_1) \quad \text{and} \quad \varphi_1' = \psi_1 \circ \varphi_1 \text{ outside } V \setminus \{\alpha, \beta\}.$$

In particular, we have $\psi_1(\varphi_1(\beta)) = \varphi_1'(\beta) = \tau_1'$, hence ψ_1 is a unifier of $\varphi_1(\beta)$ and τ_1. The most general unifier of these two types therefore exists — let us call it μ —, and $\texttt{infer}(E, a, v)$ is well-defined. Let ψ be a substitution such that $\psi_1 = \psi \circ \mu$. We now show that this substitution ψ satisfies the claim. We have:

$$\begin{array}{rcll}
\psi(\tau) & = & \psi(\mu(\varphi_1(\alpha \to \beta))) & \text{by definition of } \tau \text{ in the algorithm} \\
& = & \psi_1(\varphi_1(\alpha \to \beta)) & \text{by definition of } \psi \\
& = & \varphi_1'(\alpha \to \beta) & \text{because } \alpha \text{ and } \beta \text{ are outside of } V \setminus \{\alpha, \beta\} \\
& = & \tau_2' \to \tau_1' & \text{by construction of } \varphi_1'
\end{array}$$

Moreover, for any variable γ outside of V,

$$\begin{array}{rcll}
\psi(\varphi(\gamma)) & = & \psi(\mu(\varphi_1(\gamma))) & \text{by definition of } \varphi \text{ in the algorithm} \\
& = & \psi_1(\varphi_1(\gamma)) & \text{by definition of } \psi_1 \\
& = & \varphi_1'(\gamma) & \text{because } \gamma \notin V \\
& = & \varphi_1(\gamma) & \text{because } \gamma \notin V \text{ implies } \gamma \neq \alpha \text{ and } \gamma \neq \beta
\end{array}$$

The desired result follows.

- **Case** $a = a_1(a_2)$. The initial derivation ends up with

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \to \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

We apply the induction hypothesis to a_1, E, V, $\tau' \to \tau''$ and φ', obtaining

$$(\tau_1, \varphi_1, V_1) = \mathtt{infer}(a_1, E, V) \quad \text{and} \quad \tau'' \to \tau' = \psi_1(\tau_1) \quad \text{and} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ outside } V.$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$. We apply the induction hypothesis to a_2, $\varphi_1(E)$, V_1, τ and ψ_1. This is legitimate, since $\mathcal{F}(\varphi_1(E)) \cap V_1 = \emptyset$ by the remark at the beginning of the proof. We get:

$$(\tau_2, \varphi_2, V_2) = \mathtt{infer}(a_2, \varphi_1(E), V_1) \quad \text{and} \quad \tau'' = \psi_2(\tau_2) \quad \text{and} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ outside } V_1.$$

We have $\mathcal{F}(\tau_1) \cap V_1 = \emptyset$, hence $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Take $\psi_3 = \psi_2 + \alpha \mapsto \tau'$. (The variable α, taken in V_2, is out of reach for ψ_2, hence ψ_3 extends ψ_2.) We have:

$$\begin{array}{lllllll}
\psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \to \tau' \\
\psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' & &
\end{array}$$

The substitution ψ_3 is therefore a unifier of $\varphi_2(\tau_1)$ and $\tau_2 \to \alpha$. Those two types therefore admit a most general unifier, μ. Hence $\mathtt{infer}(a_1(a_2), E, V)$ is well-defined. In addition, we have $\psi_3 = \psi_4 \circ \mu$ for some substitution ψ_4. We now show that $\psi = \psi_4$ meets the requirements of the proposition. With the same notations as in the algorithm, we have

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau',$$

on one hand, and on the other hand, for all $\beta \notin V$ (hence a fortiori $\beta \notin V_1$, $\beta \notin V_2$, $\beta \neq \alpha$):

$$\begin{array}{rcll}
\psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{by definition of } \varphi \\
& = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{by definition of } \psi_4 \\
& = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{because } \beta \neq \alpha \text{ and } \alpha \text{ is out of reach for } \varphi_1 \text{ and } \varphi_2 \\
& = & \psi_1(\varphi_1(\beta)) & \text{because } \varphi_1(\beta) \notin V_1 \\
& = & \varphi'(\beta) & \text{because } \beta \notin V.
\end{array}$$

This is the expected result.

- **Cas** $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$. The initial derivation ends up with

$$\frac{\varphi'(E) \vdash a_1 : \tau' \qquad \varphi'(E) + x \mapsto \mathtt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau''}{\varphi'(E) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau''}$$

We apply the induction hypothesis to a_1, E, V, τ' and φ'. We get

$$(\tau_1, \varphi_1, V_1) = \mathtt{infer}(a_1, E, V) \quad \text{and} \quad \tau' = \psi_1(\tau_1) \quad \text{and} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ outside } V.$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$. We easily check that $\psi_1(\texttt{Gen}(\tau_1, \varphi_1(E)))$ is more general than $\texttt{Gen}(\psi_1(\tau_1), \psi_1(\varphi_1(E)))$, that is, than $\texttt{Gen}(\tau', \varphi'(E))$. Since we can derive

$$\varphi'(E) + x \mapsto \texttt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau'',$$

proposition 1.3 shows that we can a fortiori derive

$$\varphi'(E) + x \mapsto \psi_1(\texttt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau'',$$

that is,

$$\psi_1(\varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau''.$$

We now apply the induction hypothesis to a_2, in the environment $\varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E))$, with the fresh variables V_1, the type τ'' and the substitution ψ_1. It follows that

$$(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2,\ \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$$

and $\tau'' = \psi_2(\tau_2)$ and $\psi_1 = \psi_2 \circ \varphi_2$ outside V_1. The algorithm takes $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$. Let us show that $\psi = \psi_2$ meets the claim. We have $\psi(\tau) = \tau''$. And if $\alpha \notin V$, a fortiori $\alpha \notin V_1$, hence:

$$\begin{array}{rcll}
\psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha)))) & \text{by definition of } \varphi \\
 & = & \psi_1(\varphi_1(\alpha)) & \text{because } \varphi_1(\alpha) \notin V_1\text{, since } \alpha \text{ is out of reach for } \varphi_1 \\
 & = & \varphi'(\alpha) & \text{since } \alpha \notin V.
\end{array}$$

Hence $\varphi' = \psi \circ \varphi$ outside V, as expected. □