The *trivialML* language: formal syntax, semantics and typing [version 2.3a]

Programmation Fonctionnelle Avancée, année 2011-2012
`http://ageinghacker.net/lipn-stuff/teaching/PFA-2011/`

Master 2 Programmation et Logiciels Sûrs — Institut Galilée, Université Paris 13

Luca Saiu `<positron@gnu.org>`
Last updated on November 27th 2012 [updated my website URL and e-mail]

trivialML is a statically-typed call-by-value purely functional language, essentially an OCaml subset. Differently from OCaml, *trivialML* only supports type checking (rather than type inference), and does not support sum types or pattern-matching. Recursion is only supported through global function definitions. Of course *trivialML* does not include more advanced features such as modules or objects, either.

This document contains a reasonably rigorous mathematical specification of *trivialML*, suitable to be translated into a working interpreter written in a functional language such as OCaml.

1 Syntax

Even if *trivialML* resembles an ML core we want to stress that *it is a different language*; that's why we write *trivialML* keywords in French, each one being the obvious translation of a corresponding OCaml keyword. We formally specify *trivialML*'s syntax by a *context-free grammar*.

We use the metavariable i for integer literals, b for boolean literals, c for values (also called "constants"), x for variables, e for expressions, t for toplevel forms, p for programs, τ for types, possibly with decorations. "*op*" represents primitive operators: $op \in \{$`+`, `-`, `*`, `/`, `=`, `<`, `et`, `ou`$\}$.

Of course expressions are central in a functional language. A *trivialML* expression can be (respectively) a literal constant, a variable, a primitive operator use, an anonymous function, a function application, a conditional or a block:

$e ::=$
- c
- $\mid x$
- $\mid e \; op \; e$
- $\mid$ `fonction` x `:` τ `->` e
- $\mid e \; e$
- $\mid$ `si` e `alors` e `sinon` e
- $\mid$ `soit` x `:` $\tau = e$ `dans` e

Notice how type declarations are mandatory for all binding forms in *trivialML*: each one always includes a "`:` τ" declaration right after the variable name.

trivialML only has integer, boolean and function types, with *no type variables*:

$\tau ::=$
- `entier`
- $\mid$ `booléen`
- $\mid \tau$ `->` τ

It's easy to anticipate at this point that the only supported literal constants are integers and booleans:

$c ::=$
- i
- $\mid b$

A *trivialML* toplevel form is simply a global definition, similar to an OCaml toplevel `let` (and different from the block `let..in` which resembles *trivialML*'s `soit..dans`):

$t ::=$
- `soit` x `:` $\tau = e$`;;`

A program is a sequence of zero or more toplevel forms followed by exactly one expression, the *main expression*. Since there is no input/output feature in *trivialML*, the idea is that we will take the value computed by the main expression as the final "result" of the whole program.

$p ::=$
- e
- $\mid t \; p$

For simplicity we will ignore the issue of parsing [ALSU07] in this document, and always write programs in concrete syntax, possibly with meta-variables; but of course a working interpreter will work on programs represented as *abstract syntax trees* rather than strings, and a parser will be needed to obtain such trees from a string representation.

Here's a sample *trivialML* program:

```
soit fact : entier -> entier =
  fonction n : entier ->
    si n = 0 alors
      1
    sinon
      n * (fact (n - 1));;
fact 10
```

Notice that there is no double semicolon at the end of the main expression.

2 Semantics

We are now proceeding to specify a *denotational semantics* [Win93] for *trivialML*: in other words, our semantics will be a mathematical function mapping a program and its global variable assignment to its "meaning", which is to say the result of evaluating the main expression.

2.1 Semantic domains

Let $\mathbb{P}$ be the set of all *trivialML* programs, $\mathbb{T}$ the set of all *trivialML* toplevel forms, $\mathbb{E}$ the set of all *trivialML* expressions, $\mathbb{X}$ the set of all variables, and $\mathbb{C}$ the set of all values.

2.1.1 Environments

An *environment* is a function $\mathbb{X} \to \mathbb{C}$ from identifiers to values. We use environments to keep track of the value of each variable when evaluating parts of the program.

Since environments are functions, functions are relations, and relations are sets of pairs, it is formally correct to use set notation when speaking about functions: we will use the symbol "$\varnothing$" for the empty environment.

If η and γ are environments, x is a variable and c is a constant, we write:

- "$\eta[x \mapsto c]$" to mean an environment identical to η everywhere except on x, where the updated environment maps x into c;
- "$\eta[\gamma]$" to mean an environment identical to η everywhere except on the domain of γ, where the updated environment behaves like γ. In other words, $\eta[\gamma]$ contains the bindings of η merged with the bindings of γ, where in case of conflict γ takes priority.

2.1.2 Values

We have already said that our values can only be integers, booleans or functions: hence we can define $\mathbb{C}$ as a *disjoint sum* of integers, booleans, and functions. We can disregard the internal structure of integers and booleans, but functions are much more interesting; we represent a function by a data structure called a *closure*. A closure is a triple containing:

- an *environment*: the local environment which was active at the time of function creation;
- a *variable*: the function formal parameter;
- an *expression*: the function body.

We write integer values as $\mathcal{I}(i)$ for $i \in \mathbb{Z}$, boolean values as $\mathcal{B}(b)$ for $b \in \{vrai,\ faux\}$, and function values as $\mathcal{F}(\rho, x, e)$ for $\rho \in \mathbb{X} \to \mathbb{C}$, $x \in \mathbb{X}$, $e \in \mathbb{E}$. Notice that in our notation "$\mathcal{I}$", "$\mathcal{B}$" and "$\mathcal{F}$" are just *labels*, not unlike value constructors in OCaml.

For example, $\mathcal{I}(42)$ is the integer value "forty-two" and $\mathcal{F}(\varnothing, n, n \texttt{ + } 1)$ is a "successor" function value.

2.2 Expression semantics

Since we also have global definitions in addition to local binders (function parameters and blocks), we need *two* distinct environments for evaluating expressions. We will use the metavariable Γ for *global environments*, and and the metavariable ρ for *local environments*.

The next definition will be our first one containing some English text which risks to be mistaken for *trivialML* syntax; from now on we will write such *meta-syntactic* operations in underlined italic and in English (for example "$\underline{\mathit{let}}\ x\ \underline{\mathit{be}}\ 2\ \underline{\mathit{in}}\ x$"), to distinguish them from similar pieces of *trivialML* syntax, written in typewriter font and in French (for example "`soit x = 2 dans x`"). The same holds for infix operators such as "+": the plus sign in "$2 \underline{+} 2$" is an actual arithmetic operation to be executed, while the plus sign in `2 + 2` is just a piece of a *trivialML* program; of course this distinction is essential when writing interpreters.

Our *expression evaluation function* $E[\![\_]\!]\_\_$ takes three parameters: a *trivialML* expression, a local environment, and a global environment; its result is a value. More formally we can write:

$$E[\![\_]\!]\_\_ : \mathbb{E} \to (\mathbb{X} \to \mathbb{C}) \to (\mathbb{X} \to \mathbb{C}) \to \mathbb{C}$$

The heart of our semantics is the following definition of $E[\![\_]\!]\_\_$:

$$\begin{array}{l}
E[\![i]\!]\rho\Gamma = \mathcal{I}(i)\\
E[\![b]\!]\rho\Gamma = \mathcal{B}(b)\\
E[\![x]\!]\rho\Gamma = \Gamma[\rho](x)\\
E[\![e_1\ op\ e_2]\!]\rho\Gamma = (E[\![e_1]\!]\rho\Gamma)\ \underline{op}\ (E[\![e_2]\!]\rho\Gamma)\\
E[\![\texttt{fonction}\ x\ \texttt{:}\ \tau\ \texttt{->}\ e]\!]\rho\Gamma = \mathcal{F}(\rho, x, e)\\
E[\![e_1\ e_2]\!]\rho\Gamma =\\
\quad \underline{\mathit{let}}\ \mathcal{F}(\rho_c, x_c, e_c)\ \underline{\mathit{be}}\ E[\![e_1]\!]\rho\Gamma\ \underline{\mathit{in}}\\
\quad \underline{\mathit{let}}\ c_2\ \underline{\mathit{be}}\ E[\![e_2]\!]\rho\Gamma\ \underline{\mathit{in}}\\
\quad E[\![e_c]\!](\rho_c[x_c \mapsto c_2])\Gamma\\
E[\![\texttt{si}\ e_1\ \texttt{alors}\ e_2\ \texttt{sinon}\ e_3]\!]\rho\Gamma =\\
\quad \underline{\mathit{let}}\ c_1\ \underline{\mathit{be}}\ E[\![e_1]\!]\rho\Gamma\ \underline{\mathit{in}}\\
\quad \underline{\mathit{if}}\ c_1 = \mathcal{B}(vrai)\ \underline{\mathit{then}}\ E[\![e_2]\!]\rho\Gamma\ \underline{\mathit{else}}\ E[\![e_3]\!]\rho\Gamma\\
E[\![\texttt{soit}\ x\ \texttt{:}\ \tau\ \texttt{=}\ e_1\ \texttt{dans}\ e_2]\!]\rho\Gamma =\\
\quad \underline{\mathit{let}}\ c_1\ \underline{\mathit{be}}\ E[\![e_1]\!]\rho\Gamma\ \underline{\mathit{in}}\\
\quad E[\![e_2]\!](\rho[x \mapsto c_1])\Gamma
\end{array}$$

We give a quick commentary of the definition above, in order.

The semantics of a *constant* is the same constant. The semantics of a *variable* is its value in the current environment, either local or (if the variable is unbound in the local environment) global; notice how the updated environment is applied like a function. The semantics of a *primitive operation* is the result of executing the corresponding primitive operation on the semantics of its arguments, evaluated in the same environments (here we use $E[\![\_]\!]\_\_$ recursively for the first time). The semantics of a *function* is simply a closure containing the local environment, the formal parameter and the body. The semantics of a function *application* is obtained by computing the semantic of the operator, which should return a closure, and the semantics of the operand, yielding the actual parameter; we obtain the final result by evaluating the closure body in the closure environment, extended by binding the formal parameter to the actual parameter. The semantics of a *conditional* is obtained by computing the semantics of the condition: if it's true then the semantics of the whole conditional is the same as the semantics of the "then" branch, otherwise it's the same as the semantics of the "else" branch (same environments). The semantics of a *block* is given by evaluating the bound expression, and then evaluating the body in an environment extended by binding the bound variable to the value obtained before.

Notice that in the semantics above we never used types and we assumed all types were correct (for example, we assumed the operator of a function application to yield a closure and the condition of a conditional to yield a boolean); this is acceptable because *trivialML* is a *statically-typed* programming language, which means that we do type checking in a separate phase, *before* execution.

2.2.1 Alternative block semantics

The following alternative definition for the block is operationally equivalent to the one above, and shows how a block can be emulated by applying an anonymous function:

$E[\![\texttt{soit } x \texttt{ : } \tau = e_1 \texttt{ dans } e_2]\!]\rho\Gamma =$
$\quad E[\![\texttt{(fonction } x \texttt{ : } \tau \texttt{ -> } e_2\texttt{)}\ e_1]\!]\rho\Gamma$

For example it's easy to convince oneself that the expression "`soit x : entier = 20 dans x + 1`" will behave just like "`(fonction x : entier -> x + 1) 20`".

2.3 Toplevel semantics

A toplevel forms yields no "result" in itself: we just evaluate a given toplevel form in a given global environment, obtaining a new global environment: so if we call $T[\![\_]\!]\_$ the *toplevel form evaluation function* we can write:

$$T[\![\_]\!]\_ : \mathbb{T} \to (\mathbb{X} \to \mathbb{C}) \to (\mathbb{X} \to \mathbb{C})$$

Here is the definition of $T[\![\_]\!]\_$:

$T[\![\texttt{soit } x \texttt{ : } \tau \texttt{ = } e\texttt{;;}]\!]\Gamma =$
$\quad \underline{let}\ c\ \underline{be}\ E[\![e]\!]\varnothing\Gamma\ \underline{in}$
$\quad \Gamma[x \mapsto c]$

Here the idea is simply obtaining an updated global environment where the bound name is associated to the result of evaluating the expression.

2.4 Program semantics

We evaluate a given program in a given global environment, obtaining a value (which is the result of evaluating the main expression) as a result. We call $P[\![\_]\!]\_$ the *program evaluation function*. Since the program evaluation function takes a program and a global environment and returns a value, we can write:

$$P[\![\_]\!]\_ : \mathbb{P} \to (\mathbb{X} \to \mathbb{C}) \to \mathbb{C}$$

Here is the definition of $P[\![\_]\!]\_$:

$P[\![e]\!]\Gamma = E[\![e]\!]\varnothing\Gamma$
$P[\![t.p]\!]\Gamma =$
$\quad \underline{let}\ \Gamma'\ \underline{be}\ T[\![t]\!]\Gamma\ \underline{in}$
$\quad P[\![p]\!]\Gamma'$

The definition above clearly follows the inductive structure of a program:

- *base case*: the result of evaluating a program only made of a main expression without any toplevel forms is the same as evaluating the main expression in the same global environment, with an empty local environment;
- *recursive case*: for evaluating a program containing at least one initial toplevel form, we first evaluate the initial toplevel form in the given global environment, obtaining a new global environment; in this new global environment we evaluate the rest of the program.

3 Typing

We are now about to specify how to statically type *trivialML* expressions [Pie02]. *Logical rules* such as the ones below can be used to type languages stronger and more realistic than *trivialML* such as more powerful ML dialects; moreover such rule-based formalisms even permit to *infer* types (automatically computing an expression type, as the OCaml system does), rather than just *checking* types.

We work with *type environments* — also called *type assignments* — which is to say mappings from variables to *types* (instead of values). We call "$\mathbb{Y}$" the set of all types; hence type environments will be $\mathbb{X} \to \mathbb{Y}$ functions. We adopt the same notational conventions for type environments as for environments.

We restrict our attention to expressions here: the toplevel part is trivial.

3.1 Typing rules for *trivialML* expressions

We ignore the distinction between local and global type environments in typing rules; such a distinction is uninteresting if we only deal with expressions, and merging global and local environments into a single object lets us lighten our notation.

A *judgement* "$\Sigma \vdash e : \tau$" consists in a type environment, an expression and a type (with meta-variables); intuitively it means that under the type environment Σ the expression e has type τ.

A rule has zero or more judgments as *premises*, written above an horizontal line, and exactly one judgment as *consequence*, written below the line. We write the *rule name* between brackets on the left.

A rule should be read top-to-bottom: intuitively it means that if all the premises are true, then the consequence is also true.

$$[c_{\mathcal{I}}]\ \frac{}{\Sigma \vdash i : \texttt{entier}}$$

$$[c_{\mathcal{B}}]\ \frac{}{\Sigma \vdash b : \texttt{booléen}}$$

$$[x]\ \frac{}{\Sigma[x : \tau] \vdash x : \tau}$$

$$[op_{\mathcal{II}}]\ \frac{\Sigma \vdash e_1 : \texttt{entier} \qquad \Sigma \vdash e_2 : \texttt{entier}}{\Sigma \vdash e_1\ op\ e_2 : \texttt{entier}}\ op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$$

$$[op_{\mathcal{IB}}]\ \frac{\Sigma \vdash e_1 : \texttt{entier} \qquad \Sigma \vdash e_2 : \texttt{entier}}{\Sigma \vdash e_1\ op\ e_2 : \texttt{booléen}}\ op \in \{\texttt{=}, \texttt{<}\}$$

$$[op_{\mathcal{BB}}]\ \frac{\Sigma \vdash e_1 : \texttt{booléen} \qquad \Sigma \vdash e_2 : \texttt{booléen}}{\Sigma \vdash e_1\ op\ e_2 : \texttt{booléen}}\ op \in \{\texttt{et}, \texttt{ou}\}$$

$$[\texttt{fonction}]\ \frac{\Sigma[x : \tau_1] \vdash e : \tau_2}{\Sigma \vdash (\texttt{fonction } x : \tau_1 \texttt{ -> } e) : \tau_1 \texttt{ -> } \tau_2}$$

$$[@]\ \frac{\Sigma \vdash e_1 : \tau_1 \texttt{ -> } \tau_2 \qquad \Sigma \vdash e_2 : \tau_1}{\Sigma \vdash (e_1\ e_2) : \tau_2}$$

$$[\texttt{si}]\ \frac{\Sigma \vdash e_1 : \texttt{booléen} \qquad \Sigma \vdash e_2 : \tau \qquad \Sigma \vdash e_3 : \tau}{\Sigma \vdash \texttt{si } e_1 \texttt{ alors } e_2 \texttt{ sinon } e_3 : \tau}$$

$$[\texttt{soit}]\ \frac{\Sigma \vdash e_1 : \tau_1 \qquad \Sigma[x : \tau_1] \vdash e_2 : \tau_2}{\Sigma \vdash \texttt{soit } x \texttt{ : } \tau_1 = e_1 \texttt{ dans } e_2 : \tau_2}$$

As shown in Figure 1 rules can be combined into a *proof tree* so that the consequence of one becomes a premise of another, instantiating meta-variables in a consistent way; in a proof the tree leaves are axioms (rules with zero premises) and the tree root is the judgement we are proving.

$$[\texttt{soit}]\ \dfrac{[op_{\mathcal{IB}}]\ \dfrac{[c_{\mathcal{I}}]\ \dfrac{}{\varnothing \vdash \texttt{1 : entier}} \qquad [c_{\mathcal{I}}]\ \dfrac{}{\varnothing \vdash \texttt{2 : entier}}}{\varnothing \vdash \texttt{1 < 2 : booléen}} \qquad [x]\ \dfrac{}{\{\texttt{x : booléen}\} \vdash \texttt{x : booléen}}}{\varnothing \vdash \texttt{soit x : booléen = 1 < 2 dans x : booléen}}$$

$$[\texttt{soit}]\ \dfrac{[c_{\mathcal{I}}]\ \dfrac{}{\varnothing \vdash \texttt{1 : entier}} \qquad [op_{\mathcal{IB}}]\ \dfrac{[x]\ \dfrac{}{\{\texttt{a : entier}\} \vdash \texttt{a : entier}} \qquad [c_{\mathcal{I}}]\ \dfrac{}{\{\texttt{a : entier}\} \vdash \texttt{3 : entier}}}{\{\texttt{a : entier}\} \vdash \texttt{a < 3 : booléen}}}{\varnothing \vdash \texttt{soit a : entier = 1 in a < 3 : booléen}}$$

Figure 1: Two proof trees, obtained by composing proof rules while instantiating meta-variables in a consistent way; we obtain the conclusion (the tree root, on the bottom) starting from axioms (tree leafs, on the top). The first tree is the proof that the expression `soit x : booléen = 1 < 2 dans x` has type `booléen` in an empty type environment; the second tree is the proof that the expression `soit x : entier = 1 in a < 3` has type `booléen`, again in an empty type environment.

All of this is very beautiful, and an interpreter for a complex system such as OCaml could directly use rules like ours for type inference [DM82, Mil78]; but adding such support to an interpreter requires some advanced programming techniques such as *unification* [MM82, Pie02], which we don't cover in this course; that's why in *trivialML* we only do *type checking* instead of type inference, making our task much simpler.

3.2 Typing by abstract interpretation

Instead of directly using type rules, we will write *a function returning a type* from an expression (which includes type declaration for all binders); our definition of the type function will closely resemble our definition of the expression evaluation function, in a technique called *abstract interpretation* [CC77]. Of course our definition of the typing function will be inspired by our rules.

Here we take ρ and Γ to stand for *type environments*, respectively local and global; our typing function $Y[\![\_]\!]_{\_\_}$, closely follows the structure of $E[\![\_]\!]_{\_\_}$.

$$Y[\![\_]\!]_{\_\_} : \mathbb{E} \to (\mathbb{X} \to \mathbb{Y}) \to (\mathbb{X} \to \mathbb{Y}) \to \mathbb{Y}$$

Here follows the definition of $Y[\![\_]\!]_{\_\_}$:

```
$Y[\![i]\!]\rho\Gamma = \texttt{entier}$
$Y[\![b]\!]\rho\Gamma = \texttt{booléen}$
$Y[\![x]\!]\rho\Gamma = \Gamma[\rho](x)$
$Y[\![e_1\ op\ e_2]\!]\rho\Gamma = (Y[\![e_1]\!]\rho\Gamma)\ \underline{op_Y}\ (Y[\![e_2]\!]\rho\Gamma)$
$Y[\![\texttt{fonction}\ x : \tau \texttt{ -> } e]\!]\rho\Gamma = \tau \texttt{ -> } Y[\![e]\!](\rho[x : \tau])\Gamma$
$Y[\![e_1\ e_2]\!]\rho\Gamma =$
  $\underline{let}\ \tau_1 \texttt{ -> } \tau_2\ \underline{be}\ Y[\![e_1]\!]\rho\Gamma\ \underline{in}$
  $\underline{let}\ \tau_3\ \underline{be}\ Y[\![e_2]\!]\rho\Gamma\ \underline{in}$
  $\tau_2\ \underline{when}\ \tau_1 = \tau_3$
$Y[\![\texttt{si}\ e_1\ \texttt{alors}\ e_2\ \texttt{sinon}\ e_3]\!]\rho\Gamma =$
  $\underline{let}\ \tau_1\ \underline{be}\ Y[\![e_1]\!]\rho\Gamma\ \underline{in}$
  $\underline{let}\ \tau_2\ \underline{be}\ Y[\![e_2]\!]\rho\Gamma\ \underline{in}$
  $\underline{let}\ \tau_3\ \underline{be}\ Y[\![e_3]\!]\rho\Gamma\ \underline{in}$
  $\tau_2\ \underline{when}\ \tau_1 = \texttt{booléen}\ \underline{and}\ \tau_2 = \tau_3$
$Y[\![\texttt{soit}\ x : \tau = e_1\ \texttt{dans}\ e_2]\!]\rho\Gamma =$
  $\underline{let}\ \tau_1\ \underline{be}\ Y[\![e_1]\!]\rho\Gamma\ \underline{in}$
  $\underline{let}\ \tau_2\ \underline{be}\ Y[\![e_2]\!](\rho[x : \tau])\Gamma\ \underline{in}$
  $\tau_2\ \underline{when}\ \tau_1 = \tau$
```

Pay attention to the "when" clauses in the definition above: of course $Y[\![\_]\!]_{\_\_}$ is a partial function: it is undefined on ill-typed expressions.

3.2.1 Alternative block typing

Alternative typing definition for `soit..dans`:

```
$Y[\![\texttt{soit}\ x : \tau = e_1\ \texttt{dans}\ e_2]\!]\rho\Gamma =$
  $Y[\![\texttt{(fonction}\ x : \tau \texttt{ -> } e_2\texttt{)}\ e_1]\!]\rho\Gamma$
```

References

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools.* Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7528`.

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

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4:258–282, April 1982.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.