TOWARDS A ML EXTENSION WITH REFINEMENT : A SEMANTIC ISSUE

SIGNOLES J

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

03/2006

Rapport de Recherche N° 1440

Towards a ML Extension with Refinement: a Semantic Issue

Julien Signoles

PCRI — LRI (CNRS UMR 8623), LIX, INRIA Futurs
Université Paris-Sud, 91405 Orsay Cedex, France,
`Julien.Signoles@lri.fr`

Abstract. Refinement is a method to derive correct programs from specifications. A rich type language is another way to ensure program correctness. In this paper, we propose a wide-spectrum language mixing both approaches for the `ML` language. Mainly, base types are simply included into expressions, introducing underdeterminism and dependent types. We focus on the semantic aspects of such a language. We study three different semantics: a denotational, a deterministic operational and a nondeterministic operational semantics. We prove their equivalence. We show that this language is a conservative extension of `ML`.

1 Introduction

Refinement Programming by refinement consists of getting an executable program from an original abstract specification by an unbounded sequence of correctness preservation refinements. This programming paradigm, called stepwise refinement, comes from the writings of Dijkstra [10] and Wirth [22]. One of the main ideas of refinement is: as the refinement steps can be as small as wanted, correctness preserving is easy to establish. Another characteristic of refinement comes from the fact that it is piecewise: a large specification may be refined one piece at a time, each piece in an independent manner. Refinement treats programs as particular specifications: they are executable specifications. Thus the specification language should include the programming language in a wide-spectrum refinement-oriented language.

Historically refinement calculus was introduced for imperative programs. It based on the Dijkstra's weakest precondition calculus (`wpc`) [11]. At this day, the most famous refinement calculus is Back's refinement calculus [3, 4, 18]. The `B` method [1] is a notable example of commercial success. Refinement calculus for functional programs is known as "expression refinement" and was first introduced by Bird [5] and Meertens [17]. It generally uses nondeterministic expressions to introduce specification [21, 6, 19]. Another approach consists of refining types instead of expressions [13, 12]. It is based on the types-as-specifications paradigm.

Types as specifications Type checking is another way of verifying the correctness of a program with respect to a specification: types are particular specifications and the richer the type language is, the more sophisticated the specifications can be. But the more difficult the type checking is: one has to choose between powerfulness and decidability. Several approaches are possible. One can have a semi-decidable powerful type system such as this of `Cayenne` [2] or a decidable less-powerful type system such as this of `Dependent ML` [23]. An intermediate approach consists of generating proof obligations when type checking cannot be done automatically. So powerful specifications are possible and the type-checking algorithm always terminates. But, as counterpart, some proofs of correctness are left for the programmer who has to perform them by using other tools.

Purpose Our purpose is to extend the expressions of a `ML` language in order to express powerful specifications by mixing both approaches. In this paper, we focus on a fragment of `ML` restricted to a simply-typed lambda-calculus with recursive functions. We believe that the ideas described here can be further extended to a full `ML` language. As types are specifications and as such a language should be wide-spectrum, types and expressions of this language should be mixed and not distinguished. Our extension mainly consists of including `ML` base types (as `int` or `bool`) into `ML` expressions. In this way, we introduce Denney's underdeterminism [8] and dependent types. Underdeterminism is not nondeterminism: the latter is a specificational characteristic whereas the former is computational. Underdeterministic terms are only partially determined and non computable. Typing is not our main concern in this paper: here we focus on the semantic aspects of such a language. We introduce three different semantics: a denotational, a deterministic operational and a nondeterministic operational semantics. Then we prove their equivalence and show that our language is a conservative extension of `ML`.

Related work Denney's $\lambda_{\sqsubseteq}$ calculus [9] is a close work to ours. Denney uses a notion of *stubs* introducing underdeterminism by this way. These stubs correspond to the base types we introduce in expressions. So, as ours, his calculus mixes the refinement and the types-as-specifications approaches. Denney uses a set-theoretic semantics similar to ours but it has no operational semantics. Morever his calculus has no primitive notion of recursive functions, predicates are syntactically separated from the core language and they are not first-class values.

Outline Section 2 informally presents the syntax and the semantics of our language. Section 3 precisely describes the syntax. In Section 4, we present three different variants of the semantics. We prove that they are equivalent and that the language is a conservative extension of `ML`. Finally, future work is discussed in Section 5, in particular typing issues and extensions to the language.

2 Informal presentation

Syntax in short Usual `ML` programs contain expressions and types, syntactically separated. Our extension mixes them by including base types into expressions. Figure 1 presents some correct expressions. Expression 1, which contains the base type `int`, represents the set of even integers. Expression 2 generalizes the

$$2 * \texttt{int} \tag{1}$$
$$(4 * \texttt{int} \ : \ 2 * \texttt{int}) \tag{2}$$
$$\textbf{rec}\ f(x \ : \ \texttt{int}) \ = \ \textbf{if}\ x \leq 0\ \textbf{then}\ \texttt{int}\ \textbf{else}\ x * (f\ (x-1)) \tag{3}$$
$$(2 * \texttt{int}) \rightarrow (2 * \texttt{int} + 1) \tag{4}$$

Fig. 1. Some correct expressions

ML type constraint (*expr* : *type*): as we do not distinguish expressions and types, both part of this constraint are (extended) expressions. Expression 3 is a recursive function generalizing the usual factorial function. We give and explain its semantics later. Expression 4 extends the usual ML arrow types and represents the specification of a function taking an even integer and returning an odd integer. In our language, it is syntactic sugar for a lambda-expression in which the parameter does not appear in its body.

Semantics in short ML expressions are commonly interpreted by values. In our framework, expressions are interpreted by sets of values. If a ML expression e is interpreted by a value v in ML, e is interpreted by the singleton $\{v\}$ in our extension. Our language is thus a conservative extension of ML. Totally-determined expressions are programs interpreted by singletons whereas partially-determined expressions, called underdeterministic expressions, are non-computable specifications. The interpretation of an expression e containing a type ι is intuitively the collection of all the ML interpretations of e where ι is substituted (or *refined*) for some ML value of type ι. An expression e_1 is a *refinement* of (or *refines*) an expression e_2 if and only if the interpretation of e_1 is included in the interpretation of e_2 *i.e.* e_1 is more determined than e_2.

Some examples Following the above intuitive definition of an interpretation, the expression $2 * \texttt{int}$ (*resp.* $2 * \texttt{int} + 1$) denotes the set of even (*resp.* odd) integers: if one substitutes int for an integer, one gets an even (*resp.* odd) integer. Each occurrence of a type may be differently refined: for example, the interpretation of

$$e_1 \triangleq \texttt{int} * \texttt{int}$$

is *not* the set of square integers but the set of all products $p * q$ for any integers p and q, that is $\mathbb{Z}$. A valid expression denoting the set of square integers is

$$e_2 \triangleq ((\lambda x \ : \ \texttt{int}.\ x * x)\ \texttt{int}).$$

So the semantics is *not* preserved by β-reduction: interpretations of e_1 and e_2 (the redex of e_1) are different. Another interesting example is the factorial function. The interpretation of this function is, of course, the singleton containing the mathematical factorial function. This function is a refinement of its generalized version shown in Figure 1 and denoting the set of functions $x \mapsto n \times x!;\ n \ \in \ \mathbb{Z}$. More details about this are given in the paragraph "example" of Section 4.1.

Additional constructs Adding base types to expressions is however not powerful enough to express specification such as "to be a function from ℕ to ℕ" or containing first-order logical quantifications. To express such specifications we introduce two additional constructs. The first one is $\emptyset_\tau$ denoting the empty set. The annotation τ is only required for typing reasons. For example the expression

$$((\lambda x\ :\ \texttt{int}.\ \textbf{if}\ x \geq 0\ \textbf{then}\ x\ \textbf{else}\ \emptyset_{\texttt{int}})\ \texttt{int})$$

denotes ℕ. The second one is $(e_1 \mathbin{@} e_2)$, called *demonic* application and dual of $(e_1\ e_2)$, the *angelic* application which corresponds to the usual application. Informally the interpretation of the angelic application collects all possible $f(x)$ with f in the interpretation of e_1 and x in that of e_2 whereas the demonic application only collects the $f(x)$ giving the same result for all x fixing f. To get the universal and existential quantifiers is the main role of this construct (see Section 4.1). We prefer introducing this construct to introducing these quantifiers because expressions denoting first-order propositions are thus not introduced as a hack in our language.

3 Syntax

The abstract syntax of the considered language is defined by the grammar rule e presented in Figure 2. The set of the expressions e is noted $\mathcal{E}$. $\textbf{rec}\ f(x\ :\ e_1)\ =\ e_2$ may be shorted to $\lambda x\ :\ e_1.\ e_2$ if f does not appear in e_2 and $\lambda x\ :\ e_1.\ e_2$ may be shorted to $e_1 \rightarrow e_2$ if x does not appear in e_2. The construct $(e_1\ :\ e_2)$ is not useful for the semantics but, as it is essential in order to provide typing constraints, we introduce it now. This language extends a primitive functional language, called

$e ::= x$	identifier
$\mid o$	operator
$\mid \iota$	base type
$\mid \textbf{rec}\ f(x\ :\ e)\ =\ e$	recursive function
$\mid (e\ e)$	angelic application
$\mid (e \mathbin{@} e)$	demonic application
$\mid (e\ :\ e)$	refinement
$\mid \emptyset_\tau$	empty
$\tau ::= \iota \mid \tau \rightarrow \tau$	
$\epsilon ::= x \mid o \mid (\epsilon\ \epsilon) \mid (\epsilon\ :\ \tau) \mid \textbf{rec}\ f(x\ :\ \tau)\ =\ \epsilon$	

Fig. 2. Abstract syntax

$\mathcal{ML}$, defined by the grammar rule ϵ. $\mathcal{ML}$ corresponds to a simply-typed lambda-calculus with recursive functions. We believe that this language can easily be extended to a full `ML` language but this extension would unnecessarily complexify this paper. The typing and semantics of $\mathcal{ML}$ are standard and are not given here. We call $\mathcal{E}_\tau$ (resp. $\mathcal{E}_\epsilon$) the set defined by the grammar rules τ (resp. ϵ). Note that $\mathcal{E}_\tau$ and $\mathcal{E}_\epsilon$ are strict subsets of $\mathcal{E}$. $\mathcal{E}_\tau$ corresponds to the $\mathcal{ML}$-types and $\mathcal{E}_\epsilon$ corresponds to the $\mathcal{ML}$-expressions (see Proposition 6).

$\mathcal{ML}$ is parameterized by some interfaces shown in Figure 3. Constants are treated as 0-ary operators. Σ_τ associates an arrow type $\tau_1 \to \ldots \to \tau_n$ to each $(n-1)$-ary operator $(n > 0)$. Σ_ϕ associates a set to each base type and an element in the set $\Sigma_\phi(\tau_1) \to \ldots \to \Sigma_\phi(\tau_n)$ to each $(n-1)$-ary operator o such that $\Sigma_\tau(o) = \tau_1 \to \ldots \to \tau_n$ $(n > 0)$.

$\mathcal{I}$	Infinite set of identifiers x (and f)
$\mathcal{O}$	Set of operators o
$\mathcal{T}$	Set of base types ι
Σ_τ	constant and operator types
Σ_ϕ	constant, operator and base type interpretations

Fig. 3. Language parameters

4 Semantics

Before defining the semantics of the language, we have to consider a typing judgment $\Gamma \vdash e \Downarrow \tau$ inferring that e has the $\mathcal{ML}$-type τ in the typing environment Γ. A typing environment is a partial application with a finite domain from $\mathcal{I}$ to $\mathcal{E}_\tau$. The typing judgment is given in Figure 4 and defines an algorithm:

Proposition 1. *The expression inferred by the typing judgement, if it exists, is unique:* [⋆]

$$\forall \Gamma, e, \tau_1, \tau_2, \text{ if } \Gamma \vdash e \Downarrow \tau_1 \text{ and } \Gamma \vdash e \Downarrow \tau_2 \text{ then } \tau_1 \equiv \tau_2.$$

When it exists, we use $T_\Gamma(e)$ to denote this unique expression and we call it "the $\mathcal{ML}$-type of e". This name is justified by the following proposition:

Proposition 2. *The expression inferred by the typing judgement, if it exists, is a $\mathcal{ML}$-type:*

$$\forall \Gamma, e, \tau, \text{ if } \Gamma \vdash e \Downarrow \tau \text{ then } \tau \in \mathcal{E}_\tau.$$

Both these propositions are easy to prove by induction on the structure of the expression e.

4.1 Denotational semantics

Mathematical preliminaries Here we introduce some standard notions and results related to functions and domain theory. We use $A \to B$ (resp. $A \rightharpoonup B$) to denote the set of total (resp. partial) functions f from A to B and $\mathrm{dom}(f)$ the domain

[⋆] We use $\equiv$ to denote the syntactic equivalence relation.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Downarrow \tau} \qquad \frac{}{\Gamma \vdash \iota \Downarrow \iota} \qquad \frac{}{\Gamma \vdash o \Downarrow \Sigma_\tau(o)} \qquad \frac{\Gamma \vdash e_1 \Downarrow \tau \quad \Gamma \vdash e_2 \Downarrow \tau}{\Gamma \vdash (e_1 \,:\, e_2) \Downarrow \tau} \qquad \frac{}{\Gamma \vdash \emptyset_\tau \Downarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Downarrow \tau_1 \quad \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e_2 \Downarrow \tau_2}{\Gamma \vdash \mathbf{rec}\ f(x \,:\, e_1) \,=\, e_2 \Downarrow \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_2 \Downarrow \tau_2 \quad \Gamma \vdash e_1 \Downarrow \tau_2 \to \tau_1}{\Gamma \vdash (e_1\ e_2) \Downarrow \tau_1} \qquad \frac{\Gamma \vdash e_2 \Downarrow \tau_2 \quad \Gamma \vdash e_1 \Downarrow \tau_2 \to \tau_1}{\Gamma \vdash (e_1 \,@\, e_2) \Downarrow \tau_1}$$

Fig. 4. $\mathcal{ML}$-type inference

of f (*i.e.* the subset of A where f is defined). We use $\{f_i\}_{i \in I}^{\leq}$ to denote a chain on some order $\leq$. We use $E_\bot$ to denote the "flat domain" of a set E, *i.e.* the domain $(E \uplus \{\bot\}, \preceq)$ where $\uplus$ is the disjoint union over sets and $\preceq$ is a partial order defined as follows:

$$\forall (x, y) \in (E_\bot)^2,\ x \preceq y \iff x = \bot \text{ or } x = y.$$

If f is a partial function from A to B, $f_\bot$ is the partial function from $A_\bot$ to $B_\bot$ defined by:

$$x \mapsto \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(A) \\ \bot & \text{if } x = \bot \end{cases}$$

We extend $\preceq$ to partial functions from $A_\bot$ to $B_\bot$ as follows:

$$\forall (f, g) \in (A_\bot \rightharpoonup B_\bot)^2,\ f \preceq g \iff \forall x \in (\mathrm{dom}(f) \cap \mathrm{dom}(g)), f(x) \preceq g(x).$$

Note that we have $f \preceq g$ if and only if

$$\forall x \in (\mathrm{dom}(f) \cap \mathrm{dom}(g)),\ f(x) \neq \bot \implies f(x) = g(x).$$

Let X and Y be two sets. We use $\mathcal{F}_X^Y$ to denote the set of partial functions from $X_\bot$ to $Y_\bot$ with the same domain:

$$\{F \in \mathcal{P}(X_\bot \rightharpoonup Y_\bot) \mid \forall (f, g) \in F^2,\ \mathrm{dom}(f) = \mathrm{dom}(g)\}.$$

We use $\sqsubseteq$ to denote the binary relation over $\mathcal{F}_X^Y$ defined by:

$$\begin{gathered} F \sqsubseteq G \\ \iff \\ \mathrm{dom}(F) \supseteq \mathrm{dom}(G)^{\star\star} \text{ and } \forall \{f_i\}_{i \in I}^{\preceq} \subseteq F,\ \exists \{g_i\}_{i \in I}^{\preceq} \subseteq G,\ \forall i \in I,\ f_i \preceq g_i \end{gathered}$$

Lemma 1 **($(\mathcal{F}_X^Y, \sqsubseteq)$ is a cpo).**
The relation $\sqsubseteq$ is reflexive, transitive and antisymmetric and any chain $\{F_i\}_{i \in I}^{\sqsubseteq} \subseteq \mathcal{F}_X^Y$ has a least upper bound, noted $\bigsqcup_{n \geq 0} F_n$, in $\mathcal{F}_X^Y$.

** We naturally extend the dom operator to a set of functions with a same domain.

Proposition 3 **($(\mathcal{F}_X^Y, \sqsubseteq)$ is a domain).**
The cpo $(\mathcal{F}_X^Y, \sqsubseteq)$ has a least element. This least element is $\emptyset$.

By Tarski's theorem, any monotonic continuous function ϕ over $\mathcal{F}_X^Y$ has a least (pre)fixpoint, noted $\mathtt{fix}\,\phi$, such that:

$$\mathtt{fix}\,\phi = \bigsqcup_{n \geq 0} \phi^n(\emptyset)$$

Interpretation We interpret each $\mathcal{ML}$-type as a set and each expression as a subset of its $\mathcal{ML}$-type. The interpretation of a $\mathcal{ML}$-type, noted $\overline{\tau}$ is defined by induction on its structure:

$$\overline{\iota} \triangleq (\Sigma_\phi(\iota))_\perp$$
$$\overline{\tau_1 \to \tau_2} \triangleq (\overline{\tau_1} \rightharpoonup \overline{\tau_2})_\perp$$

The underlying idea is that $\perp$ is used to interpret a non-terminating expression and a partial function is used to interpret a function of our language the body of which contains the $\emptyset_{\_}$ symbol (interpreted as the empty set). So we distinguish the expressions $\lambda x : \mathtt{int}.\ \emptyset_{\mathtt{int}}$ and **rec** $f(x : \mathtt{int}) = (f\ x)$: the first expression is interpreted as the singleton containing the function never defined (its domain is empty) whereas the second expression is interpreted as the singleton containing the constant function always defined and returning $\perp$.

We can now define the interpretation $[\![e]\!]_\Gamma^\Lambda$, in the typing environment Γ and the interpretation environment Λ, of a well-typed expression e as a subset of $\overline{T_\Gamma(e)}$. Λ is a partial application which associates, to each identifier x in $\mathcal{I}$, a value in $\overline{T_\Gamma(x)}$. Note that x is not associated to a set of values. The formal definition of $[\![\_]\!]$ is given in Figure 5. Some cases are easy: the interpretation of a variable is a singleton containing its associated value in the environment, the interpretations of an operator and a base type follow Σ_ϕ, the refinement construct ignores its right part (which is less precise than the left part) and the interpretation of $\emptyset_\tau$ is the empty set. An angelic application $(e_1\ e_2)$ joins all the possible $f(x)$ resulting of the application of a function f in the set denoting e_1 on an element x in the set denoting e_2 whereas a demonic application $(e_1 @ e_2)$ only joins the $f(x)$ giving the same result for each x in the set denoting e_2. The denotation of a recursive function of our language is given by the fixpoint of a certain application ψ as usual. We now explain the intuitive meaning of ψ and its fixpoint. If a function is not recursive (*i.e.* has the form $\lambda x : e_1.\ e_2$), the fixpoint of ψ is

$$\mathtt{fix}\,\psi = \{h \mid \forall y \in [\![e_1]\!]_\Gamma^\Lambda,\ h(y) \in [\![e_2]\!]_{\Gamma, x : T_\Gamma(e_1)}^{\Lambda, x \mapsto y}\}.$$

This fixpoint is a set of functions and, for each of them, the application to an element in $[\![e_1]\!]$ belongs to $[\![e_2]\!]$ (in their respective environments). If a function is recursive, the subset of the domain which maps to $\perp$ for each function decreases step by step. When the fixpoint is reached, the remaining $\perp$ correspond exactly to the non-terminating function applications.

$$[\![\_]\!]^{\Lambda}_{\Gamma} : e : \mathcal{E} \to \mathcal{P}(\overline{T_{\Gamma}(e)})$$

$$\begin{aligned}
[\![x]\!]^{\Lambda}_{\Gamma} &= \{\Lambda(x)\} \\
[\![o]\!]^{\Lambda}_{\Gamma} &= \{(\Sigma_{\phi}(o))_{\perp}\} \\
[\![\iota]\!]^{\Lambda}_{\Gamma} &= \Sigma_{\phi}(\iota) \\
[\![\mathbf{rec}\ f(x\ :\ e_1)\ =\ e_2]\!]^{\Lambda}_{\Gamma} &= \mathtt{fix}\,\psi \\
[\![(e_1\ e_2)]\!]^{\Lambda}_{\Gamma} &= \bigcup_{f \in [\![e_1]\!]^{\Lambda}_{\Gamma}} \bigcup_{x \in [\![e_2]\!]^{\Lambda}_{\Gamma}} \{f(x)\} \\
[\![(e_1 \,@\, e_2)]\!]^{\Lambda}_{\Gamma} &= \bigcup_{f \in [\![e_1]\!]^{\Lambda}_{\Gamma}} \bigcap_{x \in [\![e_2]\!]^{\Lambda}_{\Gamma}} \{f(x)\} \\
[\![(e_1\ :\ e_2)]\!]^{\Lambda}_{\Gamma} &= [\![e_1]\!]^{\Lambda}_{\Gamma} \\
[\![\emptyset_{\tau}]\!]^{\Lambda}_{\Gamma} &= \emptyset
\end{aligned}$$

with, if we note $T_1 = T_{\Gamma}(e_1)$, $T_2 = T_{\Gamma, x:T_1, f:T_1 \to T_2}(e_2)$ and $F = \mathcal{F}^{\overline{T_2}}_{[\![e_1]\!]^{\Lambda}_{\Gamma}}$,

$$\begin{aligned}
\psi &: \mathcal{P}(F) \to \mathcal{P}(F) \\
G &\mapsto \left[\begin{array}{l} \text{if } G = \emptyset \quad \{x \mapsto \perp\} \\ \text{otherwise:} \\ \displaystyle\bigcup_{g \in G} \left\{ h \in F \;\middle|\; \forall y \in [\![e_1]\!]^{\Lambda}_{\Gamma}, \left[\begin{array}{ll} \text{if } g(y) = \perp & h(y) \in [\![e_2]\!]^{\Lambda, x \mapsto y, f \mapsto g}_{\Gamma, x:T_1, f:T_1 \to T_2} \\ \text{otherwise} & h(y) = g(y) \end{array}\right. \right\} \end{array}\right.
\end{aligned}$$

Fig. 5. Denotational semantics

Lemma 2 ($\mathtt{fix}\,\psi$ is well defined).
ψ is a monotonic continuous operator from $\mathcal{P}(F)$ to $\mathcal{P}(F)$.

Example We consider the generalized factorial given in Figure 1:

$$\mathbf{rec}\ f(x\ :\ \mathtt{int})\ =\ \mathbf{if}\ x \leq 0\ \mathbf{then}\ \mathtt{int}\ \mathbf{else}\ x * (f\ (x-1)).$$

By successive iterations, we obtain the following fixpoint:

$$\psi_{\omega} = \left\{ f \;\middle|\; \forall x \in \mathbb{Z}, \left[\begin{array}{ll} f(x) \in \mathbb{Z} & \text{if } x \leq 0 \\ f(x) = x \times f(x-1) & \text{otherwise} \end{array}\right. \right\}.$$

So, as claimed in section 2, ψ_{ω} is the set of functions $x \mapsto n \times x!;\ n \in \mathbb{Z}$ with any values for negative arguments. If we slightly modify the program by replacing $\leq$ by $=$ in its body, we obtain the following ψ_{ω}:

$$\left\{ f \;\middle|\; \forall x \in \mathbb{Z}, \left[\begin{array}{ll} f(x) = \perp & \text{if } x < 0 \\ f(x) \in \mathbb{Z} & \text{if } x = 0 \\ f(x) = x \times f(x-1) & \text{if } x > 0 \end{array}\right. \right\}$$

illustrating that these functions loop on negative arguments. But a better way to define such a function is to restrict the domain to $\mathbb{N}$:

$$\begin{gathered}
\mathbf{rec}\ f(x : ((\lambda y\ :\ \mathtt{int}.\ \mathbf{if}\ y \geq 0\ \mathbf{then}\ y\ \mathbf{else}\ \emptyset_{\mathtt{int}})\ \mathtt{int})) = \\
\mathbf{if}\ x = 0\ \mathbf{then}\ \mathtt{int}\ \mathbf{else}\ x * (f\ (x-1))
\end{gathered}$$

and we obtain the fixpoint ψ_ω

$$\left\{ f \;\middle|\; \forall x \in \mathbb{N}, \begin{bmatrix} f(x) \in \mathbb{Z} & \text{if } x = 0 \\ f(x) = x \times f(x-1) & \text{if } x > 0 \end{bmatrix} \right\}.$$

Syntactic sugar Figure 6 introduces additional useful constructs as syntactic sugar on top of our core language. For example we can rewrite the last form of

$\textbf{let } x = e_1 \textbf{ in } e_2$	$\triangleq ((\lambda x : e_1.\, e_2)\ e_1)$	let-binding
$\textbf{tel } x = e_1 \textbf{ in } e_2$	$\triangleq ((\lambda x : e_1.\, e_2) \mathbin{@} e_1)$	tel-binding
$\{x : e_1 \mid e_2\}$	$\triangleq \textbf{let } x = e_1 \textbf{ in if } e_2 \textbf{ then } x \textbf{ else } \emptyset_{T(x)}$	subtype
$(a \cup b)_\tau$	$\triangleq \{x : \tau \mid x = a \texttt{ or } x = b\}$	union
$(a \cap b)_\tau$	$\triangleq \{x : \tau \mid x = a \texttt{ and } x = b\}$	intersection
a^C_τ	$\triangleq \{x : \tau \mid x \neq a\}$	complement
$\exists x : e_1,\, e_2$	$\triangleq \textbf{let } x = e_1 \textbf{ in if } e_2 \textbf{ then } \texttt{true} \textbf{ else tel } x = e_1 \textbf{ in } e_2$	exists
$\forall x : e_1,\, e_2$	$\triangleq \textbf{let } x = e_1 \textbf{ in if } e_2 \textbf{ then tel } x = e_1 \textbf{ in } e_2 \textbf{ else } \texttt{false}$	forall

Fig. 6. Additional constructs as syntactic sugar

the factorial given in the example in a better way:

$$\textbf{rec } f(x : \texttt{int} \mid x \geq 0) \ = \ \textbf{if } x = 0 \textbf{ then } \texttt{int} \textbf{ else } x * (f\ (x-1)).$$

$x : \texttt{int} \mid x \geq 0$ is just a shortcut for $x : \{x : \texttt{int} \mid x \geq 0\}$. Proposition 4 explains why we claim that the "exists" and the "forall" constructs correspond to the usual quantifiers:

Proposition 4. *Let Γ and Λ be respectively a typing and an interpretation environment.*

"exists" characterization

1. *If* $\exists y \in [\![e_1]\!]^\Lambda_\Gamma$, $\forall p \in [\![e_2]\!]^{\Lambda, x \mapsto y}_{\Gamma, x : T_\Gamma(e_1)} \neq \emptyset$, $p(y) = \texttt{true}$, *then* $[\![\exists x : e_1,\, e_2]\!]^\Lambda_\Gamma = \{\texttt{true}\}$.
2. *If* $\forall y \in [\![e_1]\!]^\Lambda_\Gamma$, $\forall p \in [\![e_2]\!]^{\Lambda, x \mapsto y}_{\Gamma, x : T_\Gamma(e_1)} \neq \emptyset$, $p(y) = \texttt{false}$, *then* $[\![\exists x : e_1,\, e_2]\!]^\Lambda_\Gamma = \{\texttt{false}\}$.

"forall" characterization

1. *If* $\exists y \in [\![e_1]\!]^\Lambda_\Gamma$, $\forall p \in [\![e_2]\!]^{\Lambda, x \mapsto y}_{\Gamma, x : T_\Gamma(e_1)} \neq \emptyset$, $p(y) = \texttt{false}$, *then* $[\![\forall x : e_1,\, e_2]\!]^\Lambda_\Gamma = \{\texttt{false}\}$.
2. *If* $\forall y \in [\![e_1]\!]^\Lambda_\Gamma$, $\forall p \in [\![e_2]\!]^{\Lambda, x \mapsto y}_{\Gamma, x : T_\Gamma(e_1)} \neq \emptyset$, $p(y) = \texttt{true}$, *then* $[\![\forall x : e_1,\, e_2]\!]^\Lambda_\Gamma = \{\texttt{true}\}$.

4.2 Operational semantics

Operational semantics of `ML` programs commonly compute the (unique) value of a well-typed expression e in an interpretation environment $\mathit{\Delta}$. This value is either a closure when e is a function or the interpretation of a constant in its domain of interpretation otherwise. More formally, and with our notation, the set $\mathcal{V}$ of the values v is defined by

$$v ::= \mathit{\Sigma}_\phi(c) \mid (f,\, x,\, e,\, \mathit{\Delta})$$

where $(f,\, x,\, e,\, \mathit{\Delta})$ is a closure. An interpretation environment $\mathit{\Delta}$ is a partial application from $\mathcal{I}$ to $\mathcal{V}$.

Similarly to the denotational semantics, our operational semantics does not compute a unique value but a unique *set* of values. This operational semantics can be expressed in two different ways: either deterministic or not. The deterministic operational semantics directly computes the set of values corresponding to an expression. The nondeterministic operational semantics only computes one value. Each "execution" of this semantics for a given expression may compute a different value. The evaluation of an expression e is the set of values resulting of all the possible executions for e. By this way undeterministic expressions cannot really be computed with these operational semantics: these expressions are not computable. The nondeterministic operational semantics is useful in order to connect our language with the usual (operational) semantics of `ML` programs whereas the deterministic operational semantics is useful to join the nondeterministic one and the denotational semantics.

Deterministic operational semantics The evaluation judgment $\mathit{\Delta} \vdash e \blacktriangleright V$ of a well-typed expression e into a set V of values in an interpretation environment $\mathit{\Delta}$ is given in Figure 7. As e is well-typed, each sub-expression s of e has a unique $\mathcal{ML}$ type, noted $T(s)$. The rules for constants, base types, $\emptyset_\tau$ and refinement do not present any difficulty. We associate a singleton containing an unique closure to each function. The environment is used to associate a value to a variable. The angelic and demonic applications operate as they operate in the denotational semantics. $\{(f^i,\, x^i,\, e^i,\, \mathit{\Delta}^i_f)\}_{i \in I \subseteq \mathbb{N}}$ represents an indexed set of closures. The rules for an operator application $o(e_1, \ldots, e_n)$ are not given in Figure 7. They follow the denotational semantics and mix the predefined semantics $\mathit{\Sigma}_\phi(o)$ of o and the angelic application rule.

Nondeterministic operational semantics The evaluation judgment $\mathit{\Delta} \vdash e \vartriangleright v$ of a well-typed expression e into a value v in an interpretation environment $\mathit{\Delta}$ is given in Figure 8. The rules for variables, constants, refinements, functions and angelic applications are the same as those of any `ML` language. The rule for base type introduces nondeterminism: a base type ι is interpreted by choosing some value in this type. The rule for the demonic application $(e_1 \mathbin{@} e_2)$ reduces nondeterminism because the chosen value has to be computable for each possible value of e_2. There is no rule for $\emptyset_\_$ since you cannot choose a value in the empty set. As we did for the deterministic operational semantics, we omit the rules for operator applications.

$$\frac{x \mapsto v \in \Delta}{\Delta \vdash x \blacktriangleright \{v\}} \qquad \frac{}{\Delta \vdash c \blacktriangleright \{\Sigma_\phi(c)\}} \qquad \frac{}{\Delta \vdash \iota \blacktriangleright \Sigma_\phi(\iota)} \qquad \frac{}{\Delta \vdash \emptyset_\tau \blacktriangleright \emptyset}$$

$$\frac{}{\Delta \vdash \mathbf{rec}\ f(x : e_1) = e_2 \blacktriangleright \{(f, x, e_2, \Delta)\}} \qquad \frac{\Delta \vdash e_1 \blacktriangleright V_1}{\Delta \vdash (e_1 : e_2) \blacktriangleright V_1}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \blacktriangleright \{(f^i, x^i, e^i, \Delta_f^i)\}_{i \in I \subseteq \mathbb{N}} \quad \Delta \vdash e_2 \blacktriangleright \{v_2^j\}_{j \in J \subseteq \mathbb{N}} \\ \Delta_f^i, x^i \mapsto v_2^j, f^i \mapsto (f^i, x^i, e^i, \Delta_f^i) \vdash e^i \blacktriangleright V^{ij}\end{array}}{\Delta \vdash (e_1\ e_2) \blacktriangleright \bigcup_{i \in I} \bigcup_{j \in J} V^{ij}}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \blacktriangleright \{(f^i, x^i, e^i, \Delta_f^i)\}_{i \in I \subseteq \mathbb{N}} \quad \Delta \vdash e_2 \blacktriangleright \{v_2^j\}_{j \in J \subseteq \mathbb{N}} \\ \Delta_f^i, x^i \mapsto v_2^j, f^i \mapsto (f^i, x^i, e^i, \Delta_f^i) \vdash e^i \blacktriangleright V^{ij}\end{array}}{\Delta \vdash (e_1 \mathbin{@} e_2) \blacktriangleright \bigcup_{i \in I} \bigcap_{j \in J} V^{ij}}$$

Fig. 7. Deterministic operational semantics

$$\frac{x \mapsto v \in \Delta}{\Delta \vdash x \vartriangleright v} \qquad \frac{}{\Delta \vdash c \vartriangleright \Sigma_\phi(c)} \qquad \frac{\Sigma_\tau(c) \equiv \iota}{\Delta \vdash \iota \vartriangleright \Sigma_\phi(c)} \qquad \frac{\Delta \vdash e_1 \vartriangleright v_1}{\Delta \vdash (e_1 : e_2) \vartriangleright v_1}$$

$$\frac{}{\Delta \vdash \mathbf{rec}\ f(x : e_1) = e_2 \vartriangleright (f, x, e_2, \Delta)}$$

$$\frac{\Delta \vdash e_1 \vartriangleright (f, x, e, \Delta_f) \quad \Delta \vdash e_2 \vartriangleright v_2 \quad \Delta_f, x \mapsto v_2, f \mapsto (f, x, e, \Delta_f) \vdash e \vartriangleright v}{\Delta \vdash (e_1\ e_2) \vartriangleright v}$$

$$\frac{\Delta \vdash e_1 \vartriangleright (f, x, e, \Delta_f) \quad \forall v_2 \text{ such that } \Delta \vdash e_2 \vartriangleright v_2, \Delta_f, x \mapsto v_2, f \mapsto (f, x, e, \Delta_f) \vdash e \vartriangleright v}{\Delta \vdash (e_1 \mathbin{@} e_2) \vartriangleright v}$$

Fig. 8. Nondeterministic operational semantics

A conservative extension of $\mathcal{ML}$ If the language is restricted to its $\mathcal{ML}$ fragment $\mathcal{E}_\epsilon$, we have the usual semantics of $\mathcal{ML}$ programs. First, as the remaining rules of Figure 8 do not introduce nondeterminism, the interpretation of each $\mathcal{ML}$ expression is a singleton:

Proposition 5. *Let $\epsilon \in \mathcal{E}_\epsilon$, Δ be an interpretation environment and $(v_1, v_2) \in \mathcal{V}^2$.*

$$\textit{If } \Delta \vdash \epsilon \vartriangleright v_1 \textit{ and } \Delta \vdash \epsilon \vartriangleright v_2 \textit{ then } v_1 \equiv v_2.$$

Then, as a corollary of this proposition, both angelic and demonic applications are equivalent:

Proposition 6. *Let $(\epsilon_1, \epsilon_2) \in \mathcal{E}_\epsilon^2$, Δ be an interpretation environment and $(v_1, v_2) \in \mathcal{V}^2$.*

$$\textit{If } \Delta \vdash (\epsilon_1\ \epsilon_2) \vartriangleright v_1 \textit{ and } \Delta \vdash (\epsilon_1 \mathbin{@} \epsilon_2) \vartriangleright v_2 \textit{ then } v_1 \equiv v_2.$$

So it is possible to add conservatively the demonic application to the grammar rule ϵ defining $\mathcal{E}_\epsilon$. The only remaining rules are exactly those of $\mathcal{ML}$, then the given restricted semantics is the same that the one of $\mathcal{ML}$. In particular, it is preserved by β-reduction:

Proposition 7. *Let $(\epsilon_1, \epsilon_2) \in \mathcal{E}_\epsilon^2$, $\tau \in \mathcal{E}_\tau$, $x \in \mathcal{I}$, Δ be an interpretation environment and $(v_1, v_2) \in \mathcal{V}^2$.*

$$\textit{If } \Delta \vdash ((\lambda x : \tau.\, e_1)\ e_2) \rhd v_1 \textit{ and } \Delta \vdash e_1[e_2/x] \rhd v_2 \textit{ then } v_1 \equiv v_2.$$

4.3 Equivalence of the semantics

In this section, we prove that the three semantics are equivalent. We need a notion of compatibility between an interpretation environment Δ of the operational semantics and an interpretation environment Λ of the denotational semantics. Such a notion is quite intuitive but rather technical to formalize and is not detailed here. We extend this notion to a typing environment Γ. First, the deterministic and the nondeterministic operational semantics are equivalent on the terminating programs:

Theorem 1. *Let Δ be an operational interpretation environment and $e \in \mathcal{E}$. Let Λ (resp. Γ) be a denotational interpretation (resp. typing) environment compatible with Δ. Suppose that $\bot \notin [\![e]\!]_\Gamma^\Lambda$. Then*

$$\forall V,\ \Delta \vdash e \blacktriangleright V \iff (\forall v,\ \Delta \vdash e \rhd v \iff v \in V).$$

Second, the deterministic operational and the denotational semantics are equivalent on the terminating programs:

Theorem 2. *Let Δ be an operational interpretation environment, Λ be a denotational interpretation environment and Γ a typing environment, all of them pairwise compatible. Let $e \in \mathcal{E}$ such that $\bot \notin [\![e]\!]_\Gamma^\Lambda$. Then*

$$\begin{array}{l} \textit{if } T_\Gamma(e) \equiv \iota \textit{ then } \Delta \vdash e \blacktriangleright [\![e]\!]_\Gamma^\Lambda \\ \textit{otherwise } (\text{i.e } \textit{if } T_\Gamma(e) \equiv \tau_1 \to \tau_2) \\ \quad \Delta \vdash e \blacktriangleright \{(f^i, x^i, e^i, \Delta_f^i)\}_{i \in I} \iff [\![e]\!]_\Gamma^\Lambda = \bigcup_{i \in I} \mathtt{fix}\, \psi_i.^{\dagger} \end{array}$$

We prove both theorems by induction on the structure of the expression e. If a program p loops on some entries, the operational semantics and the denotational semantics may differ because the first one does not produce an output (the derivation tree is infinite) whereas the denotation of p is a set containing $\bot$. Using both theorems, we immediately deduce that the nondeterministic operational semantics and the denotational semantics are equivalent on the terminating programs.

† ψ_i corresponds to the operator ψ defined in Figure 5 where e_3, Λ and Γ are respectively substituted by e^i, Λ_f^i and Γ_f^i.

5 Future work

There are many directions for future work. Our most immediate concern is typing in order to verify the type annotations in a program of our language. The next item of interest for us is the extension of our language in order to include `ML` features. We are also interested in developing a prototype of this language.

5.1 Typing

Type annotations of our language must be verified to ensure its correctness: for example, when a function $\lambda x : e_2.\ e$ is applied to an argument e_1, we have to verify that e_2 is an acceptable type for e_1, *i.e.* that the interpretation of e_1 is included in the interpretation of e_2. Unfortunately, such a verification is undecidable in presence of dependent types as we have in our language.

There are several approaches to solve this problem. First, we could have an undecidable type system such as the one of `Cayenne` [2]. The presence of $\emptyset_{\_}$ and $(\_ @ \_)$ potentially increases the number of programs for which the type verifier does not return an answer. So this solution has to be considered carefully. Second, we could only accept a restricted form of dependent types for which there is a decidable type system similarly to `Dependent ML` [23]. But we prefer not to restrict the expressive power of our language. Moreover such a restriction would probably complicate the syntax of our language. We prefer an intermediate approach consisting of generating proof obligations when we cannot automatically verify a type annotation. Then the user has to prove these obligations using external tools. This approach is already followed for example by the `B` method [1]. We describe below how we can proceed.

The rules which compose a type system verifying type annotations may be separated in two groups: the rules generating the proof obligations and the others. The simplest type system we can imagine has only one rule. This rule generates a proof obligation and looks like

$$\frac{[\![e_1]\!]^{\Lambda}_{\Gamma} \subseteq [\![e_2]\!]^{\Lambda}_{\Gamma}}{e_1 \text{ type-checks } e_2 \text{ in } \Gamma \text{ and } \Lambda}$$

Of course, such a type system is not satisfying because all the proofs are discharged to the user who has to understand the theoretical denotation of a program: it is untractable in practice. The easier to prove the proof obligations are, the better the type system is. A good approach seems to mix a verification judgment and an inference judgment as it is done for intersection types by Davies and Pfenning [7]. For example, in order to verify that an angelic application $(e_1\ e_2)$ verifies an expression e, we have to infer that e_1 is a function, then we have to verify that e_2 matches the type of the parameter of e_1 and, finally, that the result of the application matches e. We believe that such mixed judgments would generate few and concise proof obligations. However these proof obligations would be hard to understand for the user. Another approach consists of converting a typing constraint of our language into a first-order logical proposition. It is probably a good way to have "human-understandable" proof obligations. Curry-Howard isomorphism [14] gives us hope of establishing such a proposition. But this approach may probably generate big untractable proof obligations from not-so-big expressions. It is possible to combine both these approaches. One verifies each type annotation using a syntax-directed type system such as in the first approach. When one has to generate a proof obligation, one converts expressions into first-order logical propositions using the second approach. We would obtain a type system generating few, concise and tractable human-provable proof obligations.

5.2 Extensions

The language presented in this paper is based on the core of a `ML` language. But sum types, pattern matching, polymorphism and imperative features are essential in practice. Thus our language has to include them in order to apply to some realistic `ML` programs. Morever polymorphism could help removing annotations on $\emptyset_{\_}$ constructs, sum types could have some connections with the theory of inductive types like those of the Calculus of Inductive Constructions [20]. It should be interesting to compare our language extended with imperative features with some imperative-based refinement languages. However mixing all these extensions is really challenging: to our knowledge, there is no practical tool dedied to proof of programs which combines these functional and imperative features.

A module system *à la* `ML` is a typed functional language built on top of any other language [16] and useful in order to compose pieces of program. It seems to be not so difficult to extend our language with a module system: exactly as we introduce base types in our expressions, it is possible to introduce module types into the module expressions in order to refine modules and not only expressions. Morever we can add a notion of *axiom à la* `Extended ML` [15] into the module system in order to easily specify constraints between different definitions.

6 Conclusion

We have presented a wide-spectrum language mixing refinement and types-as-specifications approaches. Mainly, base types are simply included into expressions: underdeterministic expressions and dependent types are introduced in this way. Denotational, deterministic operational and nondeterministic operational semantics have been introduced. We have proved that they are equivalent. We have shown that our language is a conservative extension of `ML`. Future work, mainly typing and extensions to the language, are required to get a realistic program verification methodology.

References

1. Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.
2. Lennart Augustsson. Cayenne – a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

3. Ralph-Johan J. Back. *On the correctness of refinement in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.
4. Ralph-Johan J. Back, Abo Akademi, J. Von Wright, F. B. Schneider, and D. Gries. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., 1998.
5. Richard S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42. Springer-Verlag New York, Inc., 1987.
6. Alexander Bunkenburg. *Expression Refinement.* PhD thesis, Computing Science Department, University of Glasgow, 1997.
7. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208. ACM Press, 2000.
8. Ewen W.K.C. Denney. Simply-typed underdeterminism, 1997. In EU KIT/IOS International Workshop on Formal Models of Programming and their Applications, Institute of Software, Beijing.
9. Ewen W.K.C. Denney. *A Theory of Program Refinement.* PhD thesis, University of Edimburg, 1998.
10. Edsger W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured programming.* Academic Press, 1971.
11. Edsger W. Dijkstra. *A discipline of programming.* Series in Automatic Computation. 1976.
12. Tim Freeman. *Refinement Types for ML.* PhD thesis, Carnegie Mellon University, 1994.
13. Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1991.
14. William A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley Jonathan P. Seldin, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.
15. Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. The definition of Extended ML. LFCS Report ECS-LFCS-94-300, University of Edinburgh, January 1994.
16. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
17. Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, 1986.
18. Carroll Morgan. *Programming from specifications (2nd ed.).* Prentice Hall International (UK) Ltd., 1994.
19. Joseph M. Morris. Non-deterministic expressions and predicate transformers. *Inf. Process. Lett.*, 61(5):241–246, 1997.
20. Christine Paulin-Mohring. Inductive Definitions in the System Coq — Rules and Properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag, 1993.
21. Nigel Thomas Edgar Ward. *A Refinement Calculus for Nondeterministic Expressions.* PhD thesis, Dept of Computer Science, University of Queensland, 1994.
22. Niklaus Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4):221–227, april 1971.
23. Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, september 1998.