Towards a denotational semantics for a reflective Scheme - An implementation of the towerless model

Catherine Recanati, Alain Deutsch

Towards a denotational semantics for a reflective Scheme

- An implementation of the towerless model -

C. Recanati, Non Standard Logics, ...mcvax!inria!lri!nsl!cathy
A. Deutsch, LITP, Universite de Paris VI, ...mcvax!inria!litp!ald

This paper presents the Flat Reflexive Scheme Interpreter (FRSI) by means of which process migration is performed in our prototypic DSM system (see [16]). The implementation of this interpreter is based on the semantical description of the underlying language. An important conclusion of this paper is that it is not possible to give an usual structural denotational description of reflective languages. Nevertheless, we have given what we call a *relative* denotational semantics. In the first part of this paper, we explain what we mean by this and in the second part, we give a relative denotational description of our language.

1. Introduction

As it has been pointed out in previous Chameleon reports, [13, 14, 15], procedural reflection provides a fruitful research framework for the building of dynamic software migration systems. We have extended the description given in these reports to a real size language - Scheme - with a real implementation in SKIM, a dialect of Scheme (see [4, 5]).
Originally, the notion of reflection was introduced by Smith. It was then viewed as *"the ability of an agent to reason not only introspectively about its self and internal thought processes, but also externally, about its behaviour and situation in the world"* (in [8]). In terms of programs, it should be a mechanism which allows a program to gain access both to its text and to the context in which it is running. Such a device should provide a means by which an active program can extract from its environments a context-free description of its state.
In the model developed by Smith and des Rivières [9] the architecture resembles an infinite tower of continuation-passing metacircular interpreters. A program running at one level can provide code to be run at the next higher level. Although it may be possible to give a denotational description of this potentially infinite tower of interpreters, as has been claimed by Friedman and Wand in [12], the advantages of such a model of reflection have not yet been investigated and will be the subject of future work (see [16, 17]).
Friedman and Wand have proposed another model of reflection in [10]. They propose to build a reflective Scheme interpreter by adding two more primitives called `reify` and `reflect`. The intuitive meaning of these two functions is the following: `reify` will freeze the state of the program and `reflect` will reanimate it. Unfortunately, the formal description of these two primitives is not easy.

2. Towards a denotational description

In this section, we summarize the difficulties encountered and present a *relative* denotational description of a reflective Scheme.

2.1. Semantic problems

The `reflect` primitive is very similar to the `eval` procedure of LISP. The introduction of `eval` is problematic in the denotational approach because it leads to the following equation:

$$\mathcal{E}[\![(\texttt{eval}\ exp)]\!]\rho\ =\ \mathcal{E}[\![\mathcal{E}[\![exp]\!]\ \rho\ ]\!]\ \rho$$

which is not structural.
With `reflect`, a similar failing occurs. As Yelland ([15]) has pointed out, in Friedman and Wand, the semantical equation for `reflect` leads to the following term:

$$[\![e_0]\!]\rho\ \{(\lambda e\ .\ ...[\![e]\!]\ \texttt{reflectP(p)}\ \texttt{reflectK(k)})\}$$

which is again non structural because when $[\![e_0]\!]\rho$ will be passed to the continuation, it will lead to

$$...[\![\ [\![e_0]\!]\rho]\!]\ \texttt{reflectP(p)}\ \texttt{reflectK(k)}$$

Now, the use of structural induction is a corner stone of the denotational approach. Without it, it is not guaranteed that every program will be assigned a meaning.
To solve this problem Yelland proposes what he calls a metacircular denotational semantics. His approach is more operational than denotational because it is still non structural. To preserve structural induction, a denotational description defines the meaning of a syntactic component only in terms of the meaning of its immediate subcomponents. Now, in the definition of ($\texttt{reflect}\ e_0\ e_1\ e_2$) the following expression occurs:

$$\mathrm{I}\ e_0\ \rho\ \{\ (\lambda e\ .\ \mathrm{I}\ e_1\ \rho\ (\lambda p\ ...\ \mathrm{I}\ e\ \texttt{reflectP(p)}\ \texttt{reflectK(k)}))\}$$

Therefore, one more time, there is a call to the metacircular interpreter I taking as argument an expression e which is not an immediate syntactic subcomponent of the original expression. (Here, the only immediate syntactic subcomponents of the expression are e_0 e_1 and e_2).
Stoy in [7] has convincingly emphasized the differences between the operational and the denotational approach. From an implementation point of view, the operational approach can be very convenient. But the operational approach will define the value of the program in terms of what an abstract machine does with the complete program. The denotational approach will define the value of a program in terms of the values of its subcomponents. According to Stoy, "*the crux of the distinction is that by considering the function associated with a program by an operational definition of semantics, we may be begging the question of whether such a function exists and is well-defined*" ([7]).
But although the denotational approach is safer, reflection is intrinsically operational. So a complete denotational description is not possible and we have to look for another solution.

2.2. A relative structural denotational description

It is not possible to give a completely structural denotational description of a reflective language. The only thing we can do is to give a description whose structural character is relative to `reflect`. An analogous notion exists in computation theory. Computability is usually defined by a set of primitives and by a structural induction schema which makes the construction of other computable functions possible. Usually, the set of primitive functions effectively contains computable functions. But another notion of computability can be defined, by adding to the set of primitives other

functions, like the truth function T of the predicate calculus, which are not necessary computable. The new class of functions which can then be reached by structural induction is said to be *relatively* computable *in* T.

We have done something similar in the denotational model. Our idea is that, even if the whole language cannot be given a structural denotational description, it is interesting to give a description which will be structural as far as possible and make explicit the precise points at which the structural approach fails.

For instance, in our language, only the `reflect` primitive cannot be given a structural description. So we have given what we call a *relative* structural description *in* `reflect`. In this description, the meaning of all the expressions but the ones containing `reflect` is given a full denotational definition. One of the advantages of this approach is that if a program doesn't contain any `reflect` expression its correctness may be proved by structural induction.

Another interesting property of this description is that it shows which part of a program may be actually compiled. In the implementation, the purely denotational part of the semantics will translate into a compiling program and the operational part will correspond to the call of an interpreter.

3. The language

The language itself is a subset of Scheme ([1, 3, 4, 5]) extended to support reflection and reification. We present an abstract semantic of the language using notations of [7], based on the work of M. Wand in [1]; we have extended this description with a top-level definition, continuations definitions, and reify and reflect expressions. The formal specification is in section 3.1 with comments in section 3.2.

3.1. Formal specification

3.1.1. Notations

The notation is summarized below:

$\langle \dots \rangle$	sequence formation
$s \dagger k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s \S t$	concatenation of sequences s and t
$s \downarrow k$	drop the first k members of sequence s
$t \rightarrow a, b$	McCarthy conditional "if t then a else b"
$\rho[x/i]$	substitution "ρ with x for i"
$\mathtt{E} = A + B$	sum of A and B (disjoint union)
$e \mid A$	projection onto A
a in $\mathtt{E}$	canonical injection

We also use $\downarrow i$ to denote the i-th projection of a cartesian product.

3.1.2. Syntax

The following is the abstract BNF syntax of the language:

$P \in Prg$	programs
$T \in Top$	top-level expressions
$D \in Def$	definitions
$E \in Exp$	expressions
$K \in Con$	constants, including quotations
$I \in Ide$	variables
$\Gamma \in Com = Exp$	commands

$$Prg \longrightarrow T \mid T\ P$$

$$Top \longrightarrow E \mid D$$

$$\begin{aligned} Def \longrightarrow\ & \texttt{(define}\ I\ E\texttt{)} \\ \mid\ & \texttt{(define (}I_0\ I^*\texttt{)}\ \Gamma^*\ E_0\texttt{)} \end{aligned}$$

$$\begin{aligned} Exp \longrightarrow\ & K \mid I \mid \texttt{(}E_0\ E^*\texttt{)} \\ \mid\ & \texttt{(lambda (}I^*\texttt{)}\ \Gamma^*\ E_0\texttt{)} \\ \mid\ & \texttt{(if}\ E_0\ E_1\ E_2\texttt{)} \\ \mid\ & \texttt{(set!}\ I\ E\texttt{)} \\ \mid\ & \texttt{(reify}\ E_0\ K_0\texttt{)} \\ \mid\ & \texttt{(reflect}\ E_0\ E_1\ E_2\ E_3\texttt{)} \end{aligned}$$

3.1.3. Domains

$\alpha \in \mathtt{L}$		locations
$\nu \in \mathtt{N}$		natural numbers
$\tau \in \mathtt{T}$	$= \mathtt{T_k} + \mathtt{T_{var}} + \mathtt{T_\lambda} + \mathtt{T_{if}} + \mathtt{T_{set}} + \mathtt{T_{call}} + \mathtt{T_{seq}} + \mathtt{T_{rei}} + \mathtt{T_{ref}}$	textual forms
$\mathtt{T_k}$	$= \mathtt{E}$	constants
$\mathtt{T_{var}}$	$= \mathtt{Q}$	variables
$\mathtt{T_\lambda}$	$= \mathtt{Q}^* \times \mathtt{T}^* \times \mathtt{T}$	lambda forms
$\mathtt{T_{if}}$	$= \mathtt{T} \times \mathtt{T} \times \mathtt{T}$	conditional forms
$\mathtt{T_{set}}$	$= \mathtt{Q} \times \mathtt{T}$	assignments
$\mathtt{T_{call}}$	$= \mathtt{T}^*$	combinations
$\mathtt{T_{seq}}$	$= \mathtt{T}^* \times \mathtt{T}$	sequences
$\mathtt{T_{rei}}$	$= \mathtt{T}^* \times \mathtt{E}$	reify forms
$\mathtt{T_{ref}}$	$= \mathtt{T} \times \mathtt{T} \times \mathtt{T} \times \mathtt{T}$	reflect forms
B	$= \{\mathit{false}, \mathit{true}\}$	booleans
Q		symbols
H		characters
R		numbers
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L}$	pairs
$\mathtt{E_v}$	$= \mathtt{L}^*$	vectors
$\mathtt{E_s}$	$= \mathtt{L}^*$	strings
M	$= \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}, \mathit{nihil}\}$	
		miscellaneous
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	procedure values
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{B})$	stores
$\rho \in \mathtt{U}$	$= (\mathrm{Ide} \to \mathtt{Q}) \times (\mathtt{Q} \to \mathtt{L})$	environments
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	command continuations
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	expression continuations
$\zeta \in \mathtt{Z}$	$= \mathtt{U} \to \mathtt{C}$	definition continuations
A		answers
X		errors

3.1.4. Semantic functions

$\mathcal{P} : \mathrm{Prg} \to \mathtt{U} \to \mathtt{K} \to \mathtt{Z} \to \mathtt{C}$
$\mathcal{T} : \mathrm{Top} \to \mathtt{U} \to \mathtt{K} \to \mathtt{Z} \to \mathtt{C}$

$\mathcal{D} : \mathrm{Def} \to \mathtt{U} \to \mathtt{Z} \to \mathtt{C}$

$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$
$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}$
$\mathcal{C} : \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$
$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$

$$\mathcal{P}[\![\mathrm{T}]\!] = \lambda\rho\kappa\zeta \,.\, \mathcal{T}[\![\mathrm{T}]\!]\rho\kappa\zeta$$

$$\begin{aligned}\mathcal{P}[\![\mathrm{T\ P}]\!] = \lambda\rho\kappa\zeta \,.\, \mathcal{T}[\![\mathrm{T}]\!]\rho \\ (\lambda\epsilon^* \,.\, (\mathit{seq}\ (\mathit{print}\ \epsilon^*)\ \mathcal{P}[\![\mathrm{P}]\!]\rho\kappa\zeta\ )\ ) \\ (\lambda\rho' \,.\, \mathcal{P}[\![\mathrm{P}]\!]\rho'\kappa\zeta)\end{aligned}$$

$$\mathcal{T}[\![\mathrm{E}]\!] = \lambda\rho\kappa\zeta\,.\,\mathcal{E}[\![\mathrm{E}]\!]\rho\kappa$$

$$\mathcal{T}[\![\mathrm{D}]\!] = \lambda\rho\kappa\zeta\,.\,\mathcal{D}[\![\mathrm{D}]\!]\rho\zeta$$

$$\begin{array}{l}\mathcal{D}[\![(\texttt{define I E})]\!] = \\ \quad \lambda\rho\zeta\,.\,(\,\lambda\sigma\,.\,((\lambda\rho'\,.\,\mathcal{E}[\![(\texttt{set! I E})]\!]\,\rho'\,(\lambda\epsilon^*\,.\,\zeta(\rho'))) \\ \qquad\qquad (\mathit{extends}\,\rho\,\langle \mathrm{I}\rangle\,\langle \mathit{new}\,\sigma\rangle))\,)\;\sigma\end{array}$$

$$\begin{array}{l}\mathcal{D}[\![\,(\texttt{define }(\mathrm{I}_0\ \mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0)]\!] = \\ \quad \lambda\rho\zeta\,.\,(\,\lambda\sigma\,.\,((\lambda\rho'\,.\,\mathcal{E}[\![(\,\texttt{set! I}_0\ (\texttt{lambda }(\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0))]\!]\rho'\,(\lambda\epsilon^*\,.\,\zeta(\rho'))) \\ \qquad\qquad (\mathit{extends}\,\rho\,\langle \mathrm{I}_0\rangle\,\langle \mathit{new}\,\sigma\rangle))\,)\;\sigma\end{array}$$

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{array}{l}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\ \qquad\qquad (\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \rightarrow \\ \qquad\qquad\qquad\qquad \mathit{wrong}\,\texttt{"undefined variable"}, \\ \qquad\qquad\qquad\quad \mathit{send}\,\epsilon\,\kappa))\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\ \quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \S \mathrm{E}^*) \\ \qquad\qquad \rho \\ \qquad\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\ \qquad\qquad\qquad (\mathit{unpermute}\,\epsilon^*)))\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![(\texttt{lambda }(\mathrm{I}^*)\ \Gamma^*\ \mathrm{E}_0)]\!] = \\ \quad \lambda\rho\kappa\,.\,\lambda\sigma\,. \\ \qquad \mathit{new}\,\sigma \in \mathtt{L} \rightarrow \\ \qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma \mid \mathtt{L}, \\ \qquad\qquad\quad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \rightarrow \\ \qquad\qquad\qquad\quad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\kappa')) \\ \qquad\qquad\qquad\qquad\qquad\quad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*)) \\ \qquad\qquad\qquad\qquad\quad \epsilon^*, \\ \qquad\qquad\qquad\quad \mathit{wrong}\,\texttt{"wrong number of arguments"}\rangle \\ \qquad\qquad\quad \text{in } \mathtt{E}) \\ \qquad\qquad \kappa \\ \qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L})\,\mathit{unspecified}\,\sigma), \\ \qquad\quad \mathit{wrong}\,\texttt{"out of memory"}\,\sigma\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![(\texttt{if }\mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = \\ \quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\ \qquad\qquad\qquad\qquad\qquad\quad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\kappa))\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![(\texttt{if }\mathrm{E}_0\ \mathrm{E}_1)]\!] = \\ \quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\kappa, \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{send}\,\mathit{unspecified}\,\kappa))\end{array}$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified*.

$$\mathcal{E}[\![\texttt{(set! I E)}]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(single(\lambda\epsilon\,.\,assign\,(lookup\,\rho\,\mathrm{I})$$
$$\epsilon$$
$$(send\,unspecified\,\kappa)))$$

$$\mathcal{E}[\![\texttt{(begin } \Gamma^* \ \mathrm{E}_0\texttt{)}]\!] = \lambda\rho\kappa\,.\,\mathcal{C}[\![\Gamma^*]\!]\,\rho\,(\mathcal{E}[\![\mathrm{E}_0]\!]\rho\kappa)$$

$$\mathcal{E}[\![\texttt{(reflect } \mathrm{E}_0 \ \mathrm{E}_1 \ \mathrm{E}_2 \ \mathrm{E}_3\texttt{)}]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2\,\mathrm{E}_3]\!]\,\rho$$
$$(\lambda\epsilon^*\sigma\,.\,intern\,(\epsilon^* \downarrow 1)$$
$$(\lambda\tau\,.\,int\,\tau\,(up\mathtt{U}\,(\epsilon^* \downarrow 2))$$
$$(up\mathtt{K}\,(\epsilon^* \downarrow 3))\,(up\mathtt{S}\,(\epsilon^* \downarrow 4)))$$
$$\sigma$$

$$\mathcal{E}[\![\texttt{(reify } \mathrm{E}_0 \ \mathrm{K}_0\texttt{)}]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho$$
$$(single(\lambda\epsilon\sigma\,.\,applicate\,\epsilon\,\langle\mathcal{K}[\![\mathrm{K}_0]\!]\,,down\mathtt{U}\,\rho,$$
$$down\mathtt{K}\,\kappa, down\mathtt{S}\,\sigma\rangle\,\sigma))$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle$$

$$\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}^*]\!] =$$
$$\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,(single(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle\,\S\,\epsilon^*))))$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\,\Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

3.1.5. Auxiliary functions

$lookup : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$
$lookup = \lambda\rho\mathrm{I}\,.\,(\rho \downarrow 2)((\rho \downarrow 1)\mathrm{I})$

$extends : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$
$extends =$
$$\quad \lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho,$$
$$extends\,\langle\rho \downarrow 1\,,\,(\rho \downarrow 2)[(\alpha^* \downarrow 1)/(\rho \downarrow 1)(\mathrm{I}^* \downarrow 1)]\,\rangle$$
$$(\mathrm{I}^* \dagger 1)$$
$$(\alpha^* \dagger 1)$$

$wrong : \mathtt{X} \to \mathtt{C}$ [implementation-dependent]

$send : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$
$send = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
  $\lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$
    $wrong$ `"wrong number of return values"`

$new : \mathtt{S} \to (\mathtt{L} + \{error\})$ [implementation-dependent]

$hold : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$
$hold = \lambda\alpha\kappa\sigma \,.\, send\,(\sigma\alpha \downarrow 1)\kappa\sigma$

$assign : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$
$assign = \lambda\alpha\epsilon\theta\sigma \,.\, \theta(update\;\alpha\epsilon\sigma)$

$update : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$
$update = \lambda\alpha\epsilon\sigma \,.\, \sigma[\langle\epsilon, true\rangle/\alpha]$

$tievals : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$
$tievals =$
  $\lambda\psi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,$
    $new\,\sigma \in \mathtt{L} \to tievals\,(\lambda\alpha^* \,.\, \psi(\langle new\,\sigma \mid \mathtt{L}\rangle \,\S\, \alpha^*))$
      $(\epsilon^* \dagger 1)$
      $(update(new\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),$
      $wrong$ `"out of memory"`σ

$truish : \mathtt{E} \to \mathtt{B}$
$truish = \lambda\epsilon \,.\, (\epsilon = false \vee \epsilon = null) \to false, true$

$permute : \mathrm{Exp}^* \to \mathrm{Exp}^*$ [implementation-dependent]

$unpermute : \mathtt{E}^* \to \mathtt{E}^*$ [inverse of permute]

$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$applicate =$
  $\lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, wrong$ `"bad procedure"`

$sequence : \mathtt{T}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{S} \to \mathtt{A}$
$sequence =$
  $\lambda\tau^*\rho\kappa\sigma \,.\, \#\tau^* = 0 \to wrong$ `"empty sequence"` $\sigma,$
    $\#\tau^* = 1 \to int\,(\tau^* \downarrow 1)\,\rho\,\kappa\,\sigma,$
      $int\,(\tau^* \downarrow 1)\,\rho\,(\lambda\epsilon^* \,.\, sequence\,(\tau^* \dagger 1)\,\rho\,\kappa)$

$intern : \mathtt{E} \to (\mathtt{T} \to \mathtt{A}) \to \mathtt{S} \to \mathtt{A}$ [ommited]

$downU : \mathtt{U} \to \mathtt{E}$
$downU =$
  $\lambda\rho \,.\, \rho$ in $\mathtt{E}$

$downK : \mathtt{K} \to \mathtt{E}$
$downK =$
  $\lambda\kappa \,.\, \kappa$ in $\mathtt{E}$

```
down𝕊 : 𝕊 → 𝔼
down𝕊 =
  λσ . σ  in 𝔼

up𝕌 : 𝔼 → 𝕌
up𝕌 =
  λϵ . ϵ | 𝕌

up𝕂 : 𝔼 → 𝕂
up𝕂 =
  λϵ . ϵ | 𝕂

up𝕊 : 𝔼 → 𝕊
up𝕊 =
  λϵ . ϵ | 𝕊

int* : 𝕋* → 𝕌 → 𝕂 → ℂ
int* =
  λτ*ρκ . #τ* = 0 → κ ⟨⟩,
                   int (τ* ↓ 1)
                       ρ
                       (single(λϵ . int* (τ* † 1) ρ (λϵ* . κ (⟨ϵ⟩ § ϵ*))))


int : 𝕋 → 𝕌 → 𝕂 → 𝕊 → 𝔸
int =
  λτρκσ .
   cases τ of
    is𝕋k(k) → κ⟨k⟩σ
    is𝕋var(i) → hold ((ρ ↓ 2) i)
                     (single (λϵ . ϵ = undefined → wrong "undefined variable" ,
                                                    (send ϵ κ)))
                     σ
    is𝕋λ(l) →
      (λi*e* . new σ ∈ 𝕃→
                   send (⟨new σ | 𝕃,
                         λϵ*κ′ . #ϵ* = #i* →
                           tievals(λα* . ((λρ′ . sequence ϵ*)
                                          (extends ρ i* α*)))
                                   ϵ*,
                           wrong "wrong number of arguments"⟩ in 𝔼)
                        κ
                        (update (new σ | 𝕃) unspecified σ),
                   wrong "out of memory" σ)
      (l ↓ 1)
      ((l ↓ 2) § ⟨l ↓ 3⟩)
    is𝕋if(l) →
      int (l ↓ 1)
          ρ
```

$$
\begin{array}{l}
\quad (single(\lambda\epsilon\,.\,truish\,\epsilon \rightarrow \\
\qquad\qquad\qquad\qquad int\,(l \downarrow 2)\,\rho\,\kappa\,\sigma, \\
\qquad\qquad\qquad\qquad int\,(l \downarrow 3)\,\rho\,\kappa\,\sigma)) \\
\quad\;\; \sigma \\
is\mathtt{T}_{\mathrm{set}}(l) \rightarrow \\
\quad int\,(l \downarrow 2) \\
\quad\;\; \rho \\
\quad (single(\lambda\epsilon\,.\,assign((\rho \downarrow 2)(l \downarrow 1)) \\
\qquad\qquad\qquad\qquad \epsilon \\
\qquad\qquad\qquad\qquad (send\,unspecified\,\kappa))) \\
\quad\;\; \sigma \\
is\mathtt{T}_{\mathrm{seq}}(l) \rightarrow \\
\quad sequence\,((l \downarrow 1)\,\S\,\langle l \downarrow 2\rangle)\,\rho\,\kappa\,\sigma \\
is\mathtt{T}_{\mathrm{call}}(l) \rightarrow \\
\quad int^*\,(permute\,l) \\
\qquad \rho \\
\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,applicate(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\kappa) \\
\qquad\qquad (unpermute\,\epsilon^*))) \\
\qquad \sigma \\
is\mathtt{T}_{\mathrm{rei}}(l) \rightarrow \\
\quad int\,(l \downarrow 1) \\
\quad\;\; \rho \\
\quad\;\; (single\,(\lambda\epsilon\,.\,applicate\,\epsilon\,\langle l \downarrow 2, down\mathtt{U}\,\rho, down\mathtt{K}\,\kappa, down\mathtt{S}\,\sigma\rangle)) \\
\quad\;\; \sigma \\
\\
\\
is\mathtt{T}_{\mathrm{ref}}(l) \rightarrow \\
\quad int^*\,l \\
\quad\;\; \rho \\
\quad\;\; (\lambda\epsilon^*\,.\,intern\,(\epsilon^* \downarrow 1) \\
\qquad\qquad\qquad (\lambda\tau\,.\,int\,\tau\,(up\mathtt{U}\,(\epsilon^* \downarrow 2))\,(up\mathtt{K}\,(\epsilon^* \downarrow 3))\,(up\mathtt{S}\,(\epsilon^* \downarrow 4))) \\
\qquad\qquad\qquad \sigma) \\
\quad\;\; \sigma \\
is\mathtt{E}(x) \rightarrow wrong\,\texttt{"illegal object"}\,\sigma \\
\mathrm{endcases}
\end{array}
$$

3.2. Comments on the formal specification

This semantic description uses the three variables ρ, κ, σ as metavariables for environments, continuations and stores. The denotation of an expression is a function of these three variables.
The introduction of the `reflect` primitive, as we have already pointed out, is very similar to the introduction of `eval`. A consequence of this is that some expressed values will have to be assigned a new meaning. For instance, in Lisp, the quote operation maps Identifiers onto Symbols. But what could be the meaning of an expression like

```
(eval (quote x))
```

Usually, `(quote x)` denotes a symbol. But symbols are not syntactic objects and we have no mapping from Symbols to Identifiers. To remove the nonstructural loop, we will describe `eval` as

Figure 1: textual forms

a function from `E` to `E`. But a suitable description of `eval` requires another type of environments. Instead of a mapping from Identifiers to Locations, we would prefer a mapping from Symbols to Locations.
In our description, environments are of type (Ide $\rightarrow$ `Q`) $\times$ (`Q` $\rightarrow$ `L`); we can both access (`Q` $\rightarrow$ `L`) and (Ide $\rightarrow$ `L`) because we maintain our environments so that (Ide $\rightarrow$ `L`) will be obtained by composition of the two parts (Ide $\rightarrow$ `Q`) and (`Q` $\rightarrow$ `L`).
Quoted forms are mapped onto a domain of constants. We will also have to reflect them. To make the manipulation of such abstract quotations easier, we have added a new domain: the domain of textual forms (also in [6]).
Textual forms are special values. They are representations of what we could call first order values. For instance, symbols, strings, constants and numbers are first order values in opposition to proper function of [`E` $\rightarrow$ `E`]. For constants, the idea is that they mirror the syntactic structure already parsed. So they also contain some abstract forms like lambda forms, conditional forms, etc., which are distinguished by a parser but merely considered as constants.
The `intern` function converts first order values into textual forms. This function is intended to perform lexical parsing on abstract forms. Second order values cannot be interned because they don't have a canonical text form and this would generate an error continuation.
The meaning of `reflect` is described by means of an auxiliary function `interpret`, which corresponds to the `eval` primitive of a LISP interpreter. We could have written it as a function from `E` to `E`, but to make its description easier we have used the domain of textual forms. `Interpret` will not operate directly on `E`, nor on pure syntactic objects, but on the domain of textual forms (see figure 1).

Figure 2: semantic mappings

The meaning of `interpret` is given by an operational description. There was no other solution but to simulate the `eval` primitive. `Interpret` takes four arguments, a textual form, an environment, a continuation and a store.

Now we come to the `reify` and `reflect` primitives.

A call to `reify` is of the form (`reify` E_0 K_0). E_0 denotes a function which will be called with the constant K_0. More precisely, the function denoted by E_0 will take four arguments, a constant, an environment, a continuation and a store. The main effect of `reify` is to apply that function to the constant K_0, the current environment, the current continuation and the current store.

One of the difficulties here is that all these functions manipulate abstract functions like environments, continuations and stores. But these entities are not supposed to be in the domain `E` of expressed values. In order to manipulate them, we need functions defining some corresponding structures in `E`. We have respectively called these functions downU, downQ and downS. Conversely, we have upU, upQ and upS which convert manipulable environment, continuation and store into *true* environment, continuation and store. `E` will also be extended with the corresponding domains. We can now complete our figure of semantic mappings (see figure 2).

`Reflect` is intended to restart an evaluation in the context given by the last three arguments. It takes four arguments. The first denotes a value which may be turned into a textual form. The three other arguments respectively denote values which may be possibly turned into environment, continuation and store respectively.

4. Implementation

The language has been implemented in SKIM. The implementation matches the denotational description as far as possible. For instance, corresponding to the semantical equation which gives the meaning of an expression reduced to an identifier,

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,&\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\ &(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \rightarrow\\ &\qquad \mathit{wrong}\,\texttt{"undefined variable"},\\ &\qquad \mathit{send}\,\epsilon\,\kappa))\end{aligned}$$

the following code has been produced:

```
(define (E e r k)
  (if (atom? e)
      (hold (lookup r e)
    (single (lambda(e) (if (eq? e :undefined)
    (wrong "undefined variable" e)
    (send e k)))))
  ... ))
```

But the implementation of the projections of environment, continuation and store onto `E` has raised some problems. Not just any implementation would be suitable in the context of migration.

4.1. Reflection and migration

To be really useful for migration, the reified structures of environment, continuation and store should be not only *denotable* but also *decomposable* by the user. By *decomposable* we mean that these structures should have a representation expressible in terms of constituents storable on a persistent medium.
This requirement seems to impose that functions be decomposable. For environments and stores, a solution is easy to come by, since these functions can be represented as finite structures. But the treatment of continuations is not so easy because continuations are functions from $\texttt{E}^*$ to `C`, and therefore the domain on which they operate is infinite.
A similar problem was raised in [16]. The use of Skim continuations as a state vector for a Skim computation is not possible in a migration context, because Skim continuations contain references to the language in which Skim is implemented (like return adresses in C).
The FRSI was built mainly to solve that problem. But here again, the FRSI continuations seem to be of no help for migration because they contain references to Skim objects (Skim being also the language that we have chosen to write our denotational description in).
The problem we are discussing here is an implementation problem: how can we implement the two functions *down*`K` and *up*`K` of our formal description, so that we can dump a continuation onto a file ?

4.2. A solution

The situation is not as bad as it looks at a first glance. With our new interpreter a step forward has been made: we can now encode the FRSI continuations in Skim and we are able to decompose them at the Skim level.
We have added a metafunction `external` which allows to call an external function (here a Skim function). We have used this mechanism as a means of passing values between FRSI and SKIM, thus rendering continuations effectively decomposable somewhere, if not in FRSI itself.

It is not surprising that we have represented our FRSI continuations in Skim because Skim is the language in which our metadescription was written. So the best way for encoding the continuation of FRSI is obviouly a Skim object, and the `external` mechanism we have used could be viewed as just a particular implementation of *up*`K` and *down*`K`.

5. Conclusion

Reflective languages cannot be given a complete denotational description because reflection is *intrinsically* operational. In this paper we have presented a reflective Scheme. We have given a *relative* denotational description of this language and we have solved the implementation problems raised by its use for migration.

References

[1] J. Rees and W. Clinger (eds), *Revised Revised Revised Report on Scheme*, 1986.

[2] G.L.J. Steele and G.J. Sussman, The Revised Report on Scheme, a Dialect of Lisp, *MIT Artificial Intelligence Memo 452*, January 1978.

[3] W. Clinger (ed), The Revised Revised Report on Scheme or an UnCommon Lisp, *MIT Artificial Intelligence Memo 848*, August 1985. Also published as *Computer Science Department Technical Report No.174*, Indiana University, June 1985.

[4] R. Dumeur, *SKIM: un interpréteur SCHEME optimisant*, Mémoire de maitrise, Université Paris VIII, Juin 1987.

[5] A. Deutsch, *Conception, implémentation et validation d'un compilateur Scheme*, Mémoire de maitrise, Université Paris VIII, Juin 1987.

[6] C. Recanati, *LAMBDIX: un interprète LISP à liaison lexicale et évaluation paresseuse*, Thèse de 3eme cycle, Laboratoire de Recherche en Informatique, Université Paris XI, Orsay, dec. 1986.

[7] E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Language Theory*, MIT Press, Cambridge, 1977.

[8] B.C. Smith, Reflection and Semantics in LISP, *CSLI report No.84 8*, Standford University, december 84.

[9] B. C. Smith and J. des Rivières, The Implementation of Procedurally Reflective Languages, *Proc. 1984 ACM symposium on Lisp and Functional Programming*, August 1984, 348-355.

[10] D.P. Friedman and M. Wand, Reification - Reflection without metaphysics, *Proc. 1984 ACM symposium on Lisp and Functional Programming*, August 1984, 348-355.

[11] D.P. Friedman and M. Wand, The mystery of the tower revealed - A non-reflective description of the reflective tower, *Proc. 1986 ACM symposium on Lisp and Functional Programming*, August 1986, 298-307.

[12] D.P. Friedman and M. Wand B.F Duba, Getting the Levels Right, *(Preliminary Report)*, Indiana and Northeastern University, 1986.

[13] P. Yelland, Denotational Semantics for Reflection in a Procedural Language. *Esprit project N° 1228 (Chameleon), Technical report 87/6.*

[14] P. Yelland, Implementation of a Reflective Language. *Esprit project N° 1228 (Chameleon), Technical report 87/7.*

[15] J. Marks, Chameleon Reflections. *Esprit project N° 1228 (Chameleon), Technical report 87/20.*

[16] A. Deutsch, C. Recanati, I. Filotti, A scheme for Scheme migration. *Esprit project N° 1228 (Chameleon), Technical report 87/35.*

[17] A. Deutsch, C. Recanati, I. Filotti, C. Consel, The tower model of reflection and reification in Scheme - a prototype implementation. *Esprit project N° 1228 (Chameleon), Technical report 87/37.*

[18] A. Deutsch and C. Recanati, A Scheme implementation of multitasking. *Esprit project N° 1228 (Chameleon), Technical report 87/38.*

[19] A. Deutsch and C. Recanati, A Scheme persistent storage system. *Esprit project N° 1228 (Chameleon), Technical report 87/39.*