Sémantique des Langages de Programmation (SemLP)
DM : Region Types

I) Submission

Submission Date : 21/05/2017
Submission Format : Submit a virtual machine (.ova) [1] with

1. an executable of the interpreter,
2. the source code of the interpreter,
3. a REAMDE file explaining :
   - the command line options,
   - the different components,
   - how to write examples
4. a suite of examples with which you tested the implementation.

II) The language

We consider a simple extension to the monadic *call-by-value* λ-calculus of the lecture notes with :

1. Natural values,
2. Boolean values and
3. *if-then-else* conditional expressions, and
4. Recursion.

The syntax of this extended language is given in Figure 1. The symbol $\oplus$ represents a binary natural operator, $\ominus$ represents a natural binary comparator, and $\oslash$ represents a boolean binary operator. The special value $()$ is named *unit*, and it is the only value of type Unit.
We omit the semantics of this language, since we will not implement it directly. Instead, we concentrate now on a language that adds the notion of memory *regions*, that is, dynamically allocated portions of the memory, that can be manipulated in a stack-like form. These regions were first introduced by [3]. They were later improved in [1], and a usage of this concept for a language similar to that of the lecture notes can be found in [2].
The syntax of this language is given in Figure 2. As one can see from this syntax, we add the syntactic categories of region names ranged with r, and region variables, ranged with the meta-variable ρ. Notice that the values of Figure 1 are now terms annotated with a region using the syntax @. This indicates at runtime the region in which the value has to be stored. Consequently, we need syntax to create regions. This is achieved with the

1. Put it in a shared repository (eg. Dropbox or Google Drive). In the case where your implementation compiles and runs without problem in UNIX you can just submit the sources with an appropriate Makefile.

$$
\begin{array}{lcll}
x & \in & \mathcal{V}ar & \text{(var. names)} \\
f & \in & \mathcal{L}Var & \text{(letrec var. names)} \\
n & \in & \mathbb{N} & \text{(naturals)} \\
b & \in & \{\mathsf{true}, \mathsf{false}\} & \text{(booleans)} \\
 & & & \\
v & ::= & x \mid f \mid \lambda x.\ M \mid n \mid b \mid () & \text{(values)} \\
 & & & \\
M & ::= & v \mid (MM) & \text{(terms)} \\
 & \mid & M \oplus M \mid\ -M \mid M \otimes M \mid M \oslash M \mid \neg M & \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M & \\
 & \mid & \mathsf{letrec}\ f(x) = M\ \mathsf{in}\ M & \\
 & & & \\
E & ::= & (EM) \mid (vE) & \text{(ev. contexts)} \\
 & \mid & E \oplus M \mid v \oplus E \mid\ -E \mid ... & \\
 & \mid & \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{let}\ x = E\ \mathsf{in}\ M & \\
 & \mid & \mathsf{letrec}\ f(x) = E\ \mathsf{in}\ M &
\end{array}
$$

FIGURE 1 – Monadic Region-free calculus

command $\mathsf{letregion}\ \rho\ \mathsf{in}\ M$ which allocates a new region bound to the region variable ρ for the duration of the expression M. When M produces a value this region can be diposed of. Finally, as it can be see from the syntax of letrec expressions, these are polymorphic on the regions they operate on.
We can now present the semantics of our language with regions in Figure 3. At runtime we consider that configurations of our language are triples of the form :

$$\langle s, \delta, M \rangle$$

where s is a *store* mapping region names to *regions*. Regions in turn ar mappings from offsets (denoted by o) to values of the language. We shall denote by a an address in the store, that is a pair (r, o) consisting of a region mane and an offset. We notice that values are either variables, or addresses. That is, all values in this language are *boxed* – i.e. stored in the heap. The δ component of the configuration is a variable environment, mapping variables to values. Finally, we find the term being evaluated M.
The semantic judgment has the form :

$$\langle s, \delta, M \rangle \to (v, s')$$

representing the call-by-value big-step reduction of a configuration as described above, to a pair containing the final value v and the modified store s'.
Finally, functions are stored as closures. A closure is a tripe of the form :

$$(\!|\, x, M, \delta \,|\!)$$

where x is the formal variable of the function, M is its body, and δ is the environment of the function. Since functions are region polymorphic in λ-regions, we have another type of closures that includes the names of the region formal parameters of the function :

$$(\!|\, \vec{\rho}, x, M, \delta \,|\!)$$

The semantic rules of Figure 3 show how closures are to be used.
Finally we remark that we use the plus symbol $(+)$ to denote functional extension.

$$
\begin{array}{lcll}
x & \in & \mathcal{V}ar & \text{(var. names)} \\
n & \in & \mathbb{N} & \text{(naturals)} \\
b & \in & \{\mathsf{true}, \mathsf{false}\} & \text{(booleans)} \\
r & \in & \mathcal{R}eg & \text{(region names)} \\
\rho & \in & \mathcal{R}egVar & \text{(region var)} \\
p & ::= & r \mid \rho & \text{(regions)} \\
\\
v & ::= & x \mid a & \text{(values)} \\
\\
M & ::= & v \mid f[\vec{p}\,] \mathbin{@} p \mid \lambda x.\ M \mathbin{@} p \mid n \mathbin{@} p \mid b \mathbin{@} p \mid () \mathbin{@} p \mid (MM) & \text{(terms)} \\
 & \mid & M \oplus M \mathbin{@} p \mid -M \mathbin{@} p \mid M \circledcirc M \mathbin{@} p \mid M \oslash M \mathbin{@} p \mid \neg M \mathbin{@} p & \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ M & \\
 & \mid & \mathsf{letrec}\ f[\vec{\rho}\,](x) \mathbin{@} p = M\ \mathsf{in}\ M & \\
 & \mid & \mathsf{letregion}\ \rho\ \mathsf{in}\ M & \\
\\
E & ::= & (EM) \mid (vE) & \text{(ev. contexts)} \\
 & \mid & E \oplus M \mid v \oplus E \mid -E \mid ... & \\
 & \mid & \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
 & \mid & \mathsf{let}\ x = E\ \mathsf{in}\ M &
\end{array}
$$

FIGURE 2 – Monadic Region calculus

Exercice 1 : Implementing λ-regions

1. In the semantics above there are constructs of the language that are omitted such as the evaluation of arithmetic and boolean expressions, and conditionals. Complete the semantics of Figure 3 to include these constructs.
2. Implement in your favourite programming language an interpreter for the semantics of λ-regions. You can start from the AST of the language. A parser is not required. [2]

III) Types and transformations

Indeed, we are interested in executing programs written with the syntax of Figure 1, but making use of the semantics of Figure 3. To that end we will implement a compiler guided by a type and effect system.
The syntax of types is given by the following grammar :

$$
\begin{array}{lcll}
\varphi & \in & \mathcal{P}(\bigcup_p \{\mathsf{get}(p) \mid \mathsf{put}(p)\} \cup \mathit{EffVar}) & \text{effect} \\
\epsilon & \in & \mathit{EffVar} & \text{effect variable} \\
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau \xrightarrow{\epsilon.\varphi} \tau \mid \alpha & \text{basic type} \\
\mu & ::= & (\tau, p) & \text{decorated type} \\
\sigma & ::= & \tau \mid \forall \alpha\ .\ \sigma & \text{type scheme} \\
\pi & ::= & \underline{\tau} \mid \forall \alpha\ .\ \pi \mid \forall \varphi\ .\ \pi \mid \forall \rho\ .\ \pi & \text{region type scheme}
\end{array}
$$

Importantly, expressions do not only produce a type, but also an effect. Effects φ are sets of atomic effects of the form $\mathsf{get}(p)$ meaning that the expression might read a value

2. However, a file containing examples using all of the constructs of the language is required. Providing a parser will be considered for extra credit.

$$\frac{\delta(x) = v}{\langle s, \delta, x\rangle \to (v, s)} \qquad \frac{\begin{array}{c}\delta(f) = a \qquad s(a) = (\!|\, \vec{\rho}, x, M, \delta_0 \,|\!) \qquad |\vec{\rho}| = |\vec{r}| \\ o \notin \mathsf{dom}(s(r')) \qquad sv = (\!|\, x, M[\vec{r}/\vec{\rho}], \delta_0 \,|\!)\end{array}}{\langle s, \delta, f[\vec{r}]@r'\rangle \to ((r', o), s + \{r' \to r' + \{o \mapsto sv\}\})}$$

$$\frac{o \notin \mathsf{dom}(s(r)) \qquad a = (r, o)}{\langle s, \delta, (\lambda x.M) \ @ \ r\rangle \to (a, s + \{a \to (\!|\, x, M, \delta \,|\!)\})} \qquad \frac{\begin{array}{c}\langle s, \delta, M\rangle \to (a_1, s_1) \qquad s_1(a_1) = (\!|\, x, N', \delta' \,|\!) \\ \langle s_1, \delta, N\rangle \to (v_2, s_2) \qquad \langle s_2, \delta' + \{x \to v_2\}, N'\rangle \to (v, s')\end{array}}{\langle s, \delta, MN\rangle \to (v, s')}$$

$$\frac{\langle s, \delta, M\rangle \to (v_1, s_1) \qquad \langle s_1, \delta + \{x \to v_1\}, N\rangle \to (v, s')}{\langle s, \delta, \mathsf{let}\ x = M\ \mathsf{in}\ N\rangle \to (v, s')}$$

$$\frac{o \notin \mathsf{dom}(s) \qquad \delta' = \delta + \{f \to (r, o)\} \qquad \langle s + \{(p, o) \to (\!|\, \vec{\rho}, x, N, \delta' \,|\!)\}, \delta', N\rangle \to (v, s')}{\langle s, \delta, \mathsf{letrec}\ f[\vec{r}\,](x) \ @ \ p = M\ \mathsf{in}\ N\rangle \to (v, s')}$$

$$\frac{r \notin \mathsf{dom}(s) \qquad \langle s + \{r \to \emptyset\}, \delta, M[r/\rho] \to (v, s')\rangle}{\langle s, \delta, \mathsf{letregion}\ \rho\ \mathsf{in}\ M\rangle \to (v, s'\backslash\{r\})}$$

FIGURE 3 – Semantics for λ-regions

stored in region p, or $\mathsf{put}(p)$ meaning that the expression might write a value in region p. Furthermore, for unification of effects in the type system, we add effect variables ϵ.
As expected, we have the usual types int, bool, and type variables α for polymorphism. Moreover, arrow types are now decorated with a pair (ϵ, φ) representing an effect variable ϵ used for effect unification, and the expected effect (i.e. set of regions read and written) by an application of the function. The productions σ and π distinguish plain from region polymorphic functions – which have to be applied to regions to produce a plain function –.
This type system serves both to type expressions, and to guide compilation from the source language of Figure 1 to the language of Figure 3. To that end, the judgments are of the form

$$\Gamma \vdash M \Rightarrow M' : \sigma, \varphi$$

where : – Γ is a typing context, that is a mapping from program variabels to types, – M is a term in the source language, – M' is its corresponding term in the target language, – σ is the type of the expression, and – φ, is the effect resulting from the evaluation of this expression.
The function $\mathsf{Observe}(A)(\varphi)$ represents the observable effect of A and is defined as :

$$\mathsf{Observe}(A)(\varphi) = \{\mathsf{put}(p), \mathsf{get}(p) \mid p \in \mathsf{fv}(A)\} \cup \{\epsilon \mid \epsilon \in \varphi \cup \mathsf{fev}(A)\}$$

where $\mathsf{fev}(A)$ are the free effect variables of A.
For explanations about the individual rules refer to [3].

Exercice 2 : Compiling to λ-regions

Implement the type system of Figure 4. As a first step, you can consider the implementation that does not provide recursive functions.

$$\frac{\Gamma(x) = (\sigma, p) \qquad \sigma \geqslant \tau}{\Gamma \vdash x \Rightarrow x : (\tau, p), \emptyset} \qquad \frac{\Gamma + \{x \mapsto \mu_1\} \vdash M \Rightarrow M' : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{\Gamma \vdash (\lambda x.M) \Rightarrow (\lambda x.M') \mathbin{@} p : ((\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2), p), \{\mathsf{put}(p)\}}$$

$$\frac{\Gamma \vdash M \Rightarrow M' : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \qquad \Gamma \vdash N \Rightarrow N' : \mu', \varphi_2}{\Gamma \vdash MN \Rightarrow M'N' : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon\} \cup \{\mathsf{get}(p)\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash M \Rightarrow M' : (\tau_1, p_1), \varphi_1 \qquad \sigma_1 = \forall \vec{\alpha}\ \forall \vec{\epsilon}\ .\ \tau_1 \\ \mathsf{fv}(\vec{\alpha}, \vec{\epsilon}) \cap \mathsf{fv}(\Gamma, \varphi_1) = \emptyset \qquad \Gamma + \{x \mapsto (\sigma_1, p_1)\} \vdash N \Rightarrow N' : \mu, \varphi_2\end{array}}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N \Rightarrow \mathsf{let}\ x = M'\ \mathsf{in}\ N' : \mu, \varphi_1 \cup \varphi_2}$$

$$\frac{\begin{array}{c}\pi = \forall \vec{\rho}\ \forall \vec{\epsilon}\ .\ \underline{\tau} \qquad \mathsf{fv}(\vec{\rho}, \vec{\epsilon}) \cap \mathsf{fv}(\Gamma, \varphi_1) = \emptyset \qquad \Gamma + \{f \mapsto (\pi, p)\} \vdash (\lambda x.\ M) \Rightarrow (\lambda x.\ M') \mathbin{@} p : (\tau, p), \varphi_1 \\ \pi' = \forall \vec{\alpha}\ .\ \pi \qquad \mathsf{fv}(\vec{\alpha}) \cap \mathsf{fv}(\Gamma, \varphi_1) = \emptyset \qquad \Gamma + \{f \mapsto (\pi', p)\} \vdash N \Rightarrow N' : \mu, \varphi_2\end{array}}{\Gamma \vdash \mathsf{letrec}\ f(x) = M\ \mathsf{in}\ N \Rightarrow \mathsf{letrec}\ f[\vec{\rho}\,](x) \mathbin{@} p = M'\ \mathsf{in}\ N' : \mu, \varphi_1 \cup \varphi_2}$$

$$\frac{\Gamma(f) = (\pi, p') \qquad \pi = \forall \rho_0 \cdots \rho_k\ \forall \vec{\alpha}\ \forall \vec{\epsilon}\ .\ \underline{\tau}' \qquad \pi \geqslant \tau \text{ via } S \qquad \varphi = \{\mathsf{get}(p'), \mathsf{put}(p)\}}{\Gamma \vdash f \Rightarrow f[S(\rho_1), \ldots, S(\rho_k)] \mathbin{@} p : (\tau, p), \varphi}$$

$$\frac{\Gamma \vdash M \Rightarrow M' : \mu, \varphi \qquad \varphi' = \mathsf{Observe}(\Gamma, \mu)(\varphi) \qquad \{\rho_1 \cdots \rho_k\} = \mathsf{frv}(\varphi \backslash \varphi')}{\Gamma \vdash M \Rightarrow \mathsf{letregion}\ \rho_1 \cdots \rho_k\ \mathsf{in}\ M' : \mu, (\varphi / \{\mathsf{put}(\rho_i), \mathsf{get}(\rho_i) \mid 1 \leqslant i \leqslant k\})}$$

FIGURE 4 – λ-region compilation

Exercice 3 :

Provide sufficient examples to cover all of the constructs of the language. Implement a logging mechanism that shows the life span of the different regions used. [3]

IV) Optimizations

While the language of Figure 3 allows to allocate and deallocate memory on demand, the stack-based principle it uses requires that the certain regions be allocated before or after the places where they are strictly utilized. To overcome this problem, an alternative to the letregion construct is to have allocate and deallocate constructs that allow for a much finer control of region lifespans. This approach was presented in [1].

Exercice 4 :

1. Implement an interpreter for the modified language of [1].
2. Implement the type and effect system of [1].
3. Compare the life span of memory regions as produced by the system of [3] and this one.

3. As an optional parameter to the interpreter of the preceding question.

Références

[1] Aiken, A., Fähndrich, M., and Levien, R. Better static memory management : Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995* (1995), pp. 174–185.

[2] Amadio, R. M., and Régis-Gianas, Y. Certifying and reasoning on cost annotations of functional programs. In *Foundational and Practical Aspects of Resource Analysis - Second International Workshop, FOPARA 2011, Madrid, Spain, May 19, 2011, Revised Selected Papers* (2011), pp. 72–89.

[3] Tofte, M., and Talpin, J. Region-based memory management. *Inf. Comput. 132*, 2 (1997), 109–176.