An Intrinsic Denotational Semantics for a Lazy Functional Language

Leonardo Rodríguez

FaMAF,
Universidad Nacional de Córdoba,
Córdoba, Argentina
lrodrig2@famaf.unc.edu.ar

Abstract. In this paper we present a denotational semantics for a lazy functional language. The semantics is intrinsic in the sense that it defines meaning for typing derivations instead of language expressions. We contrast our semantics with the well-known evaluation rules defined by Sestoft [17] and show that these rules preserve types and meaning.

Keywords: denotational semantics, lazy evaluation, type theory

1 Introduction

In a lazy functional language, function arguments are evaluated only if needed and at most once. The evaluation is performed in normal order and with *sharing* of arguments evaluation. This paper presents a denotational semantics for a lazy language that models this evaluation strategy. The semantics is *intrinsic* in the sense of Reynolds [15, 16], since it defines meaning to typing judgements rather than to terms themselves, and as a consequence, only well-typed terms have meaning.

The semantics of lazy languages have been largely studied, and there are many operational specifications and abstract machines based on graph reduction [10, 8], super-combinators [7], and other techniques. Launchbury [12] defined a big-step operational semantics for an extended lambda calculus. The sharing of evaluation is modelled using *heaps* mapping variables to its values, which are updated when the evaluation of an expression is finished. Sestoft [17] revised the semantics by providing a way to locally check freshness of variables during evaluation, among others improvements. This paper takes the same language used by Sestoft but with the inclusion of a type-system. We define an intrinsic denotational semantics for the language, and show that the evaluation rules preserve both types and meaning.

There are other denotational definitions of the semantics for lazy functional languages [3, 4, 9]. Launchbury [12] defined a denotational semantics and presented a proof of adequacy with respect to his evaluation rules. However, Breitner [6] have recently found some issues in the proof, and then adjusted the semantics to correct them. Nakata [13] presented an alternative definition to

the denotational semantics for recursive local declarations. In [13], like in this paper, a type system is included in the language, but its semantic definition is nonetheless untyped.

2 Syntax and Semantics

We use the same language as in [12, 17], a lambda calculus with recursive local declarations which presents terms in a restricted syntax:

Definition 1 (Language terms).

$$e ::= \lambda x.\, e \mid e\, x \mid x \mid \mathit{let}\, \{x_i \mapsto e_i\}\, \mathit{in}\, e$$

The purpose of this restricted form is to ensure that every function argument has been previously bound by a local declaration (*let*), and therefore it will be shared in the heap, as will become clear later. Note that a general lambda expression may be translated into this restricted syntax by introducing new let-bindings.

Notation: Before we continue let us fix some notations about finite maps (used to represent heaps, contexts and environments). Let $M : D \to R$ be a finite map from a set D (the domain of M) to some set R (the range of M). We write $M[x \mapsto r]$ for the extension of M with a new map $\{x \mapsto r\}$. Sometimes we write $M[x_i \mapsto r_i]$ as a shortening of $M[x_1 \mapsto r_1] \ldots [r_n \mapsto r_n]$, and if $D_0 \subseteq D$ we write $M|_{D_0}$ for the restriction of M to the domain D_0. Finally, if M and M' are maps with disjoint domain, we write $M \sqcup M'$ for the combination of the two maps in a single one.

Figure 1 shows Sestoft's evaluation rules. A heap Γ is a finite map from variables to expressions, and a pair of the form (Γ, e) where e is an expression is called a configuration. A judgement of the form $(\Gamma, e) \Downarrow_A (\Delta, w)$ says that in the heap Γ, the expression e will evaluate to w producing a new heap Δ.

The evaluation rules are annotated with a set A of the variables whose value is being computed at the time. The only place where this set is updated is in the Var rule when the value of x is about to be computed.

$$\textsc{Abst}\ \frac{}{(\Gamma,\, \lambda x.\, e) \Downarrow_A (\Gamma,\, \lambda x.\, e)} \qquad \textsc{Var}\ \frac{(\Gamma,\, e) \Downarrow_{A \cup \{x\}} (\Delta,\, w)}{(\Gamma[x \mapsto e],\, x) \Downarrow_A (\Delta[x \mapsto w],\, w)}$$

$$\textsc{App}\ \frac{(\Gamma,\, e) \Downarrow_A (\Delta,\, \lambda y.\, e') \qquad (\Delta,\, e'[x \backslash y]) \Downarrow_A (\Theta,\, w)}{(\Gamma,\, e\, x) \Downarrow_A (\Theta,\, w)}$$

$$\textsc{Let}\ \frac{(\Gamma[z_i \mapsto \hat{e}_i],\, \hat{e}) \Downarrow_A (\Delta,\, w)}{(\Gamma,\, \mathit{let}\, \{x_i \mapsto e_i\}\, \mathit{in}\, e) \Downarrow_A (\Delta,\, w)}$$

Fig. 1. Big-step operational semantics

The rule Var is where sharing becomes evident. Once the expression e is evaluated, the variable x is updated in heap Δ with its new value w, avoiding in this way to evaluate the variable x again in the future.

In the Let rule we write $\hat{e}$ for the substitution $e[z_1 \backslash x_1, \ldots, z_n \backslash x_n]$. The variables z_i have to be fresh: they must not occur in Γ, A or $\mathit{let}\,\{x_i \mapsto e_i\}\,\mathit{in}\,e$. Notice that, unlike in [12], the checking of freshness can be done locally (that is, looking only at the configuration being evaluated and not at the entire evaluation tree).

It is necessary to ensure that the substitution $e'[x \backslash y]$ does not capture the variable x (in the App rule), and also that the variable x does not occur in the domain of Δ (in the Var rule), and hence the extension $\Delta[x \mapsto w]$ does not overwrite any map of Δ. In order to guarantee those properties it is required for the evaluation to produce only "A-good" configurations, as defined in [17]. For convenience we reproduce the definition here:

Definition 2. *A configuration (Γ, e) is A-good if and only if*
1. $A \cap dom(\Gamma) = \emptyset$, 2. $Fv(\Gamma, e) \subseteq A \cup dom(\Gamma)$, 3. $Bv(\Gamma, e) \cap (A \cup dom(\Gamma)) = \emptyset$.

Here $Fv(\Gamma, e)$ denotes the set of free variables of the entire configuration, including the expressions in the range of Γ. Similarly, the set $Bv(\Gamma, e)$ contains all the bound variables of the configuration (Γ, e). The following lemma, proved in [17], shows that indeed "A-good" is preserved by evaluation.

Lemma 1. *If (Γ, e) is A-good and $(\Gamma, e) \Downarrow_A (\Delta, w)$ is derivable, then (Δ, w) is A-good and $dom(\Gamma) \subseteq dom(\Delta)$.*

3 Type System

In Figure 2, we define the typing rules for expressions, heaps and configurations. A typing judgment for an expression e has the usual form $\pi \vdash e : \theta$, where π is

$$\textsc{TyAbs}\ \frac{\pi[x \mapsto \theta] \vdash e : \theta'}{\pi \vdash \lambda x.\, e : \theta \to \theta'} \qquad \textsc{TyApp}\ \frac{\pi \vdash e : \theta \to \theta' \qquad \{x \mapsto \theta\} \in \pi}{\pi \vdash e\, x : \theta'}$$

$$\textsc{TyVar}\ \frac{\{x \mapsto \theta\} \in \pi}{\pi \vdash x : \theta} \qquad \textsc{TyLet}\ \frac{\pi[x_i \mapsto \theta_i] \vdash e_i : \theta_i \quad \pi[x_i \mapsto \theta_i] \vdash e : \theta'}{\pi \vdash \mathit{let}\,\{x_i \mapsto e_i\}\,\mathit{in}\,e : \theta'}$$

$$\textsc{TyEmpty}\ \frac{}{\pi' \vdash \diamond : []} \qquad \textsc{TyExt}\ \frac{\pi'[x \mapsto \theta] \vdash \Gamma : \pi \qquad \pi' \sqcup \pi[x \mapsto \theta] \vdash e : \theta}{\pi' \vdash \Gamma[x \mapsto e] : \pi[x \mapsto \theta]}$$

$$\textsc{TyConf}\ \frac{\pi' \vdash \Gamma : \pi \qquad \pi' \sqcup \pi \vdash e : \theta}{(\pi', \pi) \vdash (\Gamma, e) : \theta}$$

Fig. 2. Typing rules

a context and θ is a type. We have two type constructors, the basic type b and arrow types of the form $\theta \to \theta'$. Contexts are finite maps from variables to types.

On the other hand, a typing judgement for a heap Γ has the form $\pi' \vdash \Gamma : \pi$, where both π' and π are contexts. The first context π' is necessary since a heap may contain free variables: in the Var evaluation rule, the variable x may occur free in the range of the heap and still be removed from its domain and included in the set A. Thus, the context π' is intended to type each variable in A, whereas the context π is meant to type each variable in the domain of Γ.

We have a single rule to type a configuration (Γ, e) that combines a typing derivation for the heap Γ and a typing derivation for the expression e. Note that the use of the operation $\pi' \sqcup \pi$ has the implicit requirement for the domain of π' and π to be disjoint. This is ensured if the configuration (Γ, e) is $dom(\pi')$-good. The following lemma states that the evaluation rules preserves types:

Lemma 2 (Type preservation). *Let (Γ, e) and (Δ, w) be configurations, π' and π_0 contexts, and θ a type such that:*

1. (Γ, e) is $dom(\pi')$-good, 2. $(\pi', \pi_0) \vdash (\Gamma, e) : \theta$, 3. $(\Gamma, e) \Downarrow_{dom(\pi')} (\Delta, w)$.

Then, there is a context π_1 such that $\pi_0 \subseteq \pi_1$ and $(\pi', \pi_1) \vdash (\Delta, w) : \theta$.

Proof. The proof is by structural induction on the evaluation rules. In each case, it is necessary to perform inversion in the typing derivation and to use Lemma 1.

4 Denotational Semantics

We used a domain-theoretic setting to define the semantics of the language. The meaning of a type θ is a domain $[\![\, \theta \,]\!]$ and the meaning of a context π is an environment $[\![\, \pi \,]\!]$ (a named finite product ordered pointwise).

In Figure 3 we present some of the equations of the semantics. We define three functions $\mathcal{E}[\![\, \_ \,]\!]_{\pi,\theta}$, $\mathcal{H}[\![\, \_ \,]\!]_{\pi,\pi'}$ and $\mathcal{C}[\![\, \_ \,]\!]_{\pi',\theta}$ that assign a continuous function to a typing derivation for a expression, a heap and a configuration, respectively.

It can be proved that this semantics is *coherent*: different typing derivations with the same conclusion have the same meaning. This property allow us to write without ambiguity $\mathcal{E}[\![\, e \,]\!]_{\pi,\theta}$ for the semantics of any typing derivation with conclusion $\pi \vdash e : \theta$ (and the same holds for the other forms of judgement). We refer to [14] for a proof of coherence of the semantics for a language larger than the one we use in this paper.

Notation: Let us clarify some notation we use in Figure 3. The symbol $\hat{\lambda}$ is used as a meta-binder to avoid confusion with the symbol λ used in abstractions. If η is an environment, we write $\eta \swarrow x$ for its projection on the variable x. Finally, we write $\mathbf{Y}_D\, f$ for the least fixed-point of a continuous function $f : D \to D$ where D is a domain.

The following lemma says that evaluation rules preserve meaning. This lemma corresponds to "Theorem 2" in [12] (correctness of denotational semantics), but this time proved for the revised semantics of Sestoft and including only well-typed configurations.

$$\mathcal{E}[\![\,\_\,]\!]_{\pi,\,\theta} : [\![\,\pi\,]\!] \to [\![\,\theta\,]\!]$$

$$\mathcal{E}[\![\,\lambda x.\, e\,]\!]_{\pi,\,\theta\to\theta'}\,\eta = \hat{\lambda}\, d\,.\ \mathcal{E}[\![\,e\,]\!]_{\pi[x\,\mapsto\,\theta],\,\theta'}\,(\eta[x\ \mapsto\ d])$$
$$\mathcal{E}[\![\,e\ x\,]\!]_{\pi,\,\theta'}\,\eta = \mathcal{E}[\![\,e\,]\!]_{\pi,\,\theta\to\theta'}\,\eta\,(\mathcal{E}[\![\,x\,]\!]_{\pi,\,\theta}\,\eta)$$
$$\mathcal{E}[\![\,x\,]\!]_{\pi,\,\theta}\,\eta = \eta\ \swarrow\ x$$
$$\mathcal{E}[\![\,let\,\{x_i\ \mapsto\ e_i\}\ in\ e\,]\!]_{\pi,\,\theta}\,\eta = \mathcal{E}[\![\,e\,]\!]_{\pi[x_i\,\mapsto\,\theta_i],\,\theta}\,\eta'$$
$$\eta' = \mathbf{Y}_{\,\pi[x_i\,\mapsto\,\theta_i]}\,(\hat{\lambda}\,\eta'\,.\ \eta[x_i\ \mapsto\ \mathcal{E}[\![\,e_i\,]\!]_{\pi[x_i\,\mapsto\,\theta_i],\,\theta_i}\,\eta'])$$

$$\mathcal{H}[\![\,\_\,]\!]_{\pi',\,\pi} : [\![\,\pi'\,]\!] \to [\![\,\pi\,]\!]$$

$$\mathcal{H}[\![\,\diamond\,]\!]_{\pi',\,[]}\ \eta = ()$$
$$\mathcal{H}[\![\,\Gamma[x\ \mapsto\ e]\,]\!]_{\pi',\,\pi[x\,\mapsto\,\theta]}\ \eta = \mathbf{Y}_{\,[\![\,\pi[x\,\mapsto\,\theta]\,]\!]}\,(\hat{\lambda}\,\eta'\,.\ \eta''[x\ \mapsto\ d])$$
$$\text{where } d = \mathcal{E}[\![\,e\,]\!]_{\pi'\,\sqcup\,\pi[x\,\mapsto\,\theta],\,\theta}\,(\eta\ \sqcup\ \eta')$$
$$\eta'' = \mathcal{H}[\![\,\Gamma\,]\!]_{\pi'[x\,\mapsto\,\theta],\,\pi}\,(\eta[x\ \mapsto\ d])$$

$$\mathcal{C}[\![\,\_\,]\!]_{\pi',\,\theta} : [\![\,\pi'\,]\!] \to [\![\,\theta\,]\!]$$

$$\mathcal{C}[\![\,(\Gamma,\,e)\,]\!]_{\pi',\,\theta}\,\eta = \mathcal{E}[\![\,e\,]\!]_{\pi'\,\sqcup\,\pi,\,\theta}\,(\eta\ \sqcup\ \mathcal{H}[\![\,\Gamma\,]\!]_{\pi',\,\pi}\,\eta)$$

Fig. 3. Denotational semantics: some equations.

Lemma 3 (Semantic preservation). *Let* $(\Gamma,\,e)$ *and* $(\Delta,\,w)$ *be configurations,* π', π_0, π_1 *be contexts, and* θ *a type such that,*

1. $(\pi',\,\pi_0) \vdash (\Gamma,\,e)\ :\ \theta$,
2. $(\Gamma,\,e)$ *is* $dom(\pi')$*-good,*
3. $\pi_0 \subseteq \pi_1$,
4. $(\pi',\,\pi_1) \vdash (\Delta,\,w)\ :\ \theta$.

Then, if $(\Gamma,\,e)\ \Downarrow_{dom(\pi')}\ (\Delta,\,w)$*, for all* $\eta \in [\![\,\pi'\,]\!]$ *it holds:*
1. $\mathcal{C}[\![\,(\Gamma,\,e)\,]\!]_{\pi',\,\theta}\,\eta = \mathcal{C}[\![\,(\Delta,\,w)\,]\!]_{\pi',\,\theta}\,\eta$, *2.* $\mathcal{H}[\![\,\Gamma\,]\!]_{\pi',\,\pi_0}\,\eta = (\mathcal{H}[\![\,\Delta\,]\!]_{\pi',\,\pi_1}\,\eta)|_{dom(\pi_0)}$.

Proof. The proof is by structural induction on the evaluation rules. It is necessary to use Lemma 2 to construct the typing derivations required to apply inductive hypothesis in each case.

The complete proof of this lemma is longer than the untyped version presented in [12], but each step in the proof is type-driven and has the simplicity provided by the typed framework.

5 Further Work

We have not yet proved computational adequacy of the semantics in the sense of [12, Sec. 5]. For instance, we should prove that if the semantics of a term is non-bottom, then the term evaluates to a normal form.

Our goal behind this intrinsic definition of the semantics is to prove the correctness of Sestoft's abstract machine [17] using type-indexed logical relations.

In our experience, intrinsic semantics are more suitable for formalization in a proof assistant with dependent types since all the semantic functions are total (without the need of type-error constants that usually complicate the semantic equations). We expect to formalize the results in this work using the Coq [2] proof assistant together with a domain theory library developed by Benton et al. [5]. We extended this library and used it to a formalize a denotational semantics for a call-by-name functional language, and a proof of correctness for a compiler targeting the Krivine abstract machine [11]. Our development is available online [1].

References

1. Some Formalizations in Coq, `http://cs.famaf.unc.edu.ar/~leorodriguez/compilercorrectness/`
2. The Coq Proof Assistant, `http://coq.inria.fr/`
3. Abramsky, S.: The Lazy Lambda Calculus. In: Turner, D. (ed.) Research topics in functional programming. pp. 65–116. Addison-Wesley (1990)
4. Abramsky, S., Ong, C.L.: Full Abstraction in the Lazy Lambda Calculus. Information and Computation 105(2), 159–267 (1993)
5. Benton, N., Kennedy, A., Varming, C.: Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages (2010), unpublished
6. Breitner, J.: The Correctness of Launchbury's Natural Semantics for Lazy Evaluation. Archive of Formal Proofs (2013)
7. Hughes, R.J.M.: Super-combinators: a New Implementation Method for Applicative Languages. In: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming. pp. 1–10. LFP '82, ACM, New York, NY, USA (1982)
8. Jones, P., L, S.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. Journal of Functional Programming 2(2), 127 202 (April 1992)
9. Josephs, M.B.: The Semantics of Lazy Functional Languages. Theoretical Computer Science 68(1), 105–111 (1989)
10. Kieburtz, R.B.: The G-machine: A Fast, Graph-reduction Evaluator. In: Proc. Of a Conference on Functional Programming Languages and Computer Architecture. pp. 400–413. Springer-Verlag New York, Inc., New York, NY, USA (1985)
11. Krivine, J.L.: A Call-by-name Lambda-calculus Machine. Higher Order Symbolic Computation. 20(3), 199–207 (Sep 2007)
12. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In: POPL. pp. 144–154 (1993)
13. Nakata, K.: Denotational Semantics for Lazy Initialization of Letrec: Black Holes as Exceptions Rather than Divergence. In: 7th Workshop on Fixed Points in Computer Science (2010)
14. Reynolds, J.C.: The Coherence of Languages with Intersection Types. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software. pp. 675–700. TACS '91, Springer-Verlag, London, UK, UK (1991)
15. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press, New York, NY, USA (1999)
16. Reynolds, J.C.: The Meaning of Types – From Intrinsic to Extrinsic Semantics. Tech. Rep. RS-00-32, BRICS (December 2000)
17. Sestoft, P.: Deriving a Lazy Abstract Machine. Journal of Functional Programing. 7(3), 231–264 (May 1997)