Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness

Seyed H. HAERI (Hossein)
MuSemantik Ltd., 6.02, Appleton Tower, 11 Crichton Street, Edinburgh, UK. EH8 9LE
email: hossein@musemantik.com

***Abstract*—For the purpose of adding to the time or space efficiency, selective enforcement of strictness is commonly practiced in today's lazy programming. Although it plays a key role in equational reasoning about programs, many few studies have considered observational equivalence between lazy programs in presence of selective strictness (OELPPSS).**

Gabbay et al. were first to consider OELPPSS and Haeri later completed their work. Both Gabbay et al. and Haeri build on a variation of the operational semantics of van Eekelen and de Mol which, in return, extends Launchbury's semantics for lazy evaluation to selective strictness. Gabbay et al. and Haeri choose to manipulate the operational semantics of van Eekelen and de Mol to prevent increase in heap expressiveness upon expression evaluation. This improvement helped them to prove their desired observational equivalences using a novel proof technique called: *induction on the number of manipulated bindings* (INMB). They used INMB to prove a handful of interesting results including a couple of observational equivalences. However, their operational semantics suffers from restrictions in expressiveness.

In this paper, we present yet another variation of van Eekelen and de Mol. Our operational semantics is as expressive as that of van Eekelen and de Mol. We prove that INMB is valid for our operational semantics too. Therefore, all the interesting results of Gabbay et al. and Haeri including their observational equivalences remain valid for our system as well. This is whilst, like that of Gabbay et al. and Haeri, our operational semantics avoids increase in heap expressiveness upon expression evaluation.

I. INTRODUCTION

The first operational semantics for lazy evaluation that correctly models sharing is that of Launchbury [1], which is based on the simple notion of heaps. van Eekelen and de Mol [2] later extend Launchbury's semantics to model lazy evaluation in presence of selective strictness. Gabbay et al. [3] and Haeri [4] are the first to consider observational equivalence between lazy programs in presence of selective strictness. However, some reasonable lazy programs such as let $y = \lambda x.x$ in $\lambda x.y$ and let $x' = \lambda x.x$ in (let $\{x_1 = \lambda x.x, x_2 = \lambda x.x_1\}$ in x_2) x' do not reduce in their system. I.e., the results of [3,4] work for a restricted model of lazy evaluation.

The novel technique of induction on the number of manipulated bindings was first introduced by [3] and [4]. The key results which validate this technique are three theorems on *atomic* variables. (Atomic variables can be thought of as the units of change in lazy evaluation.) The first two results tell us how to obtain a new (valid) derivation by removal/restoration of atomic variables in a derivation. The third states that every non-trivial lazy computation evaluates at least one atomic variable. In this paper, we prove these three theorems (Theorems IV.9, IV.10, and IV.11, respectively) for our own operational semantics. As a result, the observational equivalences proved in [3,4] are valid in our system too.

The main contributions of our paper are as follows: We provide a new operational semantics for lazy evaluation in presence of selective strictness. Like [3,4], we choose to *garbage collect* let-bindings. Thus, our operational semantics — unlike its predecessors [1,2] — does not suffer from the increase of heap expressiveness upon let-expression evaluation. On the other hand, unlike [3,4], our garbage-collecting (let) rule, retains the (possibly manipulated) let-bindings as a part of the returning value. Hence, as stated by Theorem VI.2, our operational semantics is as expressive as that of van Eekelen and de Mol [2]. In other words, unlike its antecedents [3,4], our operational semantics does not suffer from restriction in expressiveness. We show the usefulness of our operational semantics by proving the validity of INMB for it. We provide the first precise formulation of the INMB principle.

This paper is structured as follows: We start in §II by presenting our syntax and operational semantics. §III explores some interplays between heaps and derivation trees. In §IV, we study atomic variables and prove our main results which validate INMB for our system. INMB is outlined in §V. To demonstrate the use of INMB, §V provides an example proof using it and also mentions a couple of interesting results proved in [3,4] using this new inductive principle. Out literature review comes in §VI. Finally, §VII concludes this paper and explores future work.

Whilst only proof sketches are provided here, proofs which differ between our system and [3,4] come in [5]. We do not reproduce the similar proofs of [4]. Proof of Theorem VI.2 comes in [6]. We inherit syntax, definitions, and some explanations from [3,4].

II. SYNTAX AND OPERATIONAL SEMANTICS

We start by presenting some syntax (Definition II.1) to serve our big step operational semantics (Definition II.7). Like [2], our operational semantics extends that of Launchbury for selective strictness.

Definition II.1. *Fix a countably infinite set of* ***variable symbols****. $x, y, z, \ldots$ and $x_1, x_2, x_3, \ldots$ will range over variable*

symbols. Define ***expressions*** *and* ***values*** *by*

$$e ::= x \mid \lambda x.e \mid e\,x \mid \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e \mid e\;\mathsf{seq}\;e$$
$$v ::= \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e \qquad n \geq 0$$

e, e', e_1, ... will range over expressions. v, v', v_1, ... will range over values. $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e$ *when $n = 0$ is a syntactic sugar for $\lambda x.e$, i.e., when there is no* let*-bindings. We inherit Launchbury's [1] standard restriction in that functions can only be applied to variables. As stated in [1], this does not reduce expressiveness because we also have* let*.*

Unlike Launchbury [1], we quotient syntax up to α-equivalence of λ- and let*-bound variables. For example, $\lambda x.x = \lambda y.y$. The reasons behind this design choice are discussed in full detail in [3]–[5]. In particular, this design choice dismisses the need for local freshness check in [7].*

Remark II.2. Call $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e$ when $n \geq 1$ **let-surrounded abstractions**. In considering let-surrounded abstractions values, we follow Ariola and Felleisen [8]. Hereafter, unless stated otherwise for $\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,\lambda x.e$ we assume that $n \geq 1$.

Definition II.3. *Define $fv(e)$ the* ***free variables*** *of e by:*

$$fv(x) = \{x\} \qquad fv(\lambda x.e) = fv(e) \setminus \{x\}$$
$$fv(e\,x) = fv(e) \cup \{x\}$$
$$fv(\mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e) = (fv(e) \cup \textstyle\bigcup_{i=1}^{n} fv(e_i)) \setminus \{x_i\}_{i=1}^{n}$$
$$fv(e'\;\mathsf{seq}\;e) = fv(e') \cup fv(e).$$

See Remark II.8 for why we choose $(fv(e) \cup \bigcup_{i=1}^{n} fv(e_i)) \setminus \{x_i\}_{i=1}^{n}$ and not $(fv(e) \setminus \{x_i\}_{i=1}^{n}) \cup \bigcup_{i=1}^{n} fv(e_i)$. (This is essentially to inherit the way [1] deals with let recursion.)

Remark II.4. We inherit Launchbury's assumption in that "all bound variables are distinct" [1, §3.1]. As a result, we do not need to distinguish between $\mathsf{let}\,\{y_1 = e_1, y_2 = e_2\}\,\mathsf{in}\,\lambda x.e$ and $\mathsf{let}\,\{y_1 = e_1\}\,\mathsf{in}\,\{y_2 = e_2\}\,\mathsf{in}\,\lambda x.e$. We therefore choose the former expression as a syntactic shorthand for the latter. An interesting result of this is that the latter (with the nested let-bindings) is then a value in our system.

Notation II.5. *Write $e[y/x]$ for the usual capture-avoiding substitution of x by y in e. For example, $(\lambda y.x)[y/x] = \lambda y'.y$, when x, y, and y' are distinct variables. Moreover, for $e' = \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e$, when $y \notin \{x_i\}_{i=1}^{n} \cup fv(e')$, define $e'[x/y] = \mathsf{let}\,\{x_i = e_i[x/y]\}_{i=1}^{n}\,\mathsf{in}\,e[x/y]$. Use the standard renaming for capture avoiding when $y \in \{x_i\}_{i=1}^{n} \cup fv(e')$.*

Definition II.6. *For a partial function f, define $dom(f) = \{x \mid f(x)$ defined$\}$. Call a partial function Γ mapping variable symbols to expressions and such that $dom(\Gamma)$ is finite, a* ***heap****. Γ, Δ, Θ, and Ξ will range over heaps. When $x \notin dom(\Gamma)$, define $(\Gamma, x \mapsto e)$ such that*

- $(\Gamma, x \mapsto e)(x) = e$, *and*
- $(\Gamma, x \mapsto e)(y) = \Gamma(y)$ *when* $y \neq x$.

Besides,

- $(\Gamma, x_i \mapsto e_i)_{i=1}^{1} = (\Gamma, x_1 \mapsto e_1)$, *and*
- $(\Gamma, x_i \mapsto e_i)_{i=1}^{n} = ((\Gamma, x_i \mapsto e_i)_{i=1}^{n-1}, x_n \mapsto e_n)$.

Definition II.7. *Define a big-step* ***operational semantics*** *$\Gamma : e \Downarrow \Delta : v$ as shown in Fig. 1 where:*

- *In* $(\mathbf{var_x})$*, by Definition II.6, we assume $x \notin dom(\Gamma)$.*
- *In* $(\mathbf{let})$*, 'x_i fresh' means $x_i \notin dom(\Gamma)$ and $x_i \notin fv(\Gamma(x))$ for $x \in dom(\Gamma)$, and similarly for Δ.* [1]

Intuitively, $\Gamma : e \Downarrow \Delta : v$ is: 'Trying to evaluate expression e in heap Γ will result in value v whilst the (probably manipulated) bindings are stored in Δ'.

Remark II.8. In line with Launchbury's convention on distinct variable naming, for any expression $e = \mathsf{let}\,\{x_i{=}e_i\}_{i=1}^{n}\,\mathsf{in}\,e'$ to be evaluated, all occurrences of x_i's in e address the local let-bound x_i's. Especially, so are the free occurrences of x_i's which are therefore needed to be distinguished from free variables of e. A particularly interesting effect that this has in our operational semantics is how we define free variables for let expressions (Definition II.3).

The first difference between our operational semantics and that of Launchbury [1] is that we replace their *(Lambda)* rule with our $(\mathbf{val})$ one. This is a reflection of us extending their notion of values with the let-surrounded abstractions. The second impact of this extension is in our rule for application: The returned value of the left branch above the line in $(\mathbf{app})$ might be a let-surrounded abstraction.

We 'garbage-collect'[2] the bindings $x_i \mapsto e'_i$ in the final heap (the Δ) in $(\mathbf{let})$, unlike [1,9]. (Note that e'_i might be the same as e_i or not.) This design choice brings us benefits. Thanks to 'garbage-collection' that, for instance, let-bindings are not propagated 'outside their scope' to the final heap and we do not have to reason explicitly 'up to' these choices thus. (Examples of results which fail without a garbage-collecting $(\mathbf{let})$ rule are Theorem 8.4 and Corollary 8.6 in [4].)

The rule for let in [1,9] does not garbage collect. For them, heaps are left with extra bindings after evaluation of let-expressions. The semantics in [1,9] allows variables to escape their scope during evaluation, which is forbidden in our semantics. For example, in [1], evaluation of $\mathsf{let}\,x = \lambda y.y\,\mathsf{in}\,\lambda z.(xz)$ finishes by adding $x \mapsto \lambda y.y$ to the original heap; this could then 'accidentally' bind x occurring elsewhere in the next expressions to evaluate. Launchbury is well aware of this issue and comments on it [1, §3.1]. To avoid 'accidental name-clash' in evaluations in his system, Launchbury imposes a pre-evaluation normalisation for expressions which renames all the variables already bound in heap. This is fine, if we just want to evaluate a particular expression, normalised, in a particular heap. However, for reasoning about the evaluation of classes of programs and proving operational equivalences between them, a garbage-collecting rule for let is better. (See [4, §11.3] and [5, §6.3] for more.)

Garbage-collecting $(\mathbf{let})$ rules were first introduced in [3] and [4]. The difference between our $(\mathbf{let})$ and that of theirs

[1] Consistent with [1] and subsequent work, we allow the possibility that $x_i \in fv(e_j)$ or $x_i \in fv(e'_j)$ for $1 \leq i, j \leq n$.

[2] Whilst some might not consider this the best naming, Launchbury [1, §6.2] is the first to use 'garbage-collection' in this specific meaning.

$$\frac{}{\Gamma : v \Downarrow \Gamma : v}\ (\mathbf{val}) \qquad \frac{\Gamma : e \Downarrow \Delta : v}{(\Gamma, x{\mapsto}e) : x \Downarrow (\Delta, x{\mapsto}v) : v}\ (\mathbf{var_x}) \qquad \frac{\Gamma : e_1 \Downarrow \Theta : v_1 \quad \Theta : e_2 \Downarrow \Delta : v_2}{\Gamma : e_1 \ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2}\ (\mathbf{seq})$$

$$\frac{\Gamma : e \Downarrow \Theta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda y.e' \quad \Theta : (\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e')[x/y] \Downarrow \Delta : v \qquad n \geq 0}{\Gamma : e\ x \Downarrow \Delta : v}\ (\mathbf{app})$$

$$\frac{(\Gamma, x_i{\mapsto}e_i)_{i=1}^n : e \Downarrow (\Delta, x_i{\mapsto}e_i')_{i=1}^n : v \qquad \text{when } x_i \text{ fresh}, 1 \leq i \leq n}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i'\}_{i=1}^n\ \mathsf{in}\ v}\ (\mathbf{let})$$

Fig. 1. Our Operational Semantics

is that they do not return let-bindings as a part of the value, whereas we do. It happens that this gives us more freedom than them: In their (let) rule, in addition to our freshness conditions, they force $x_i \notin fv(v)$ too. This is to avoid future referral to garbage-collected variables. For the operational semantics of [3] and [4], any such referral would halt the evaluation because the garbage-collected variables are no longer accessible. This is not the case in our system because the garbage-collected variables — although no longer bound in the heaps — are still available as the let-bindings in the returned expression. Therefore, we do not need $x_i \notin fv(v)$. Refer to [5, Chapter 6] to see how the following lazy program thus reduces in our system but not in that of [3] and [4]: $\mathsf{let}\ x' = \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ \{x_1 = \lambda x.x, x_2 = \lambda x.x_1\}\ \mathsf{in}\ x_2)\ x'$.

Our rules are applicable based on a *best-fit* strategy. Therefore, the final rule in $\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ v \Downarrow \Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ v$ is always (val) as opposed to (let). In retrospect, in $\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : v$, the final rule is (let).

[3] explains the intuition of $e_1\ \mathsf{seq}\ e_2$ as "evaluate e_1; throw the result away ... but keep the heap and use it to evaluate e_2". Our (seq) rule implements this operational behaviour, and is similar to the (StrictLet) rule of CLEAN [2,10].

Notation II.9. *A **derivation** is the labelled tree — labelled with terms, heaps, and derivation rules from Definition II.7 — that justifies a reduction $\Gamma : e \Downarrow \Delta : v$. Π, Π_1, Π', Π_x, ... will range over derivations. Write '$\Gamma : e \Downarrow \Delta : v$' as shorthand for "$\Gamma : e \Downarrow \Delta : v$ is derivable". Write '$\Gamma : e \Downarrow_\Pi \Delta : v$' as shorthand for "$\Pi$ is a derivation of $\Gamma : e \Downarrow \Delta : v$". "_" will represent the wildcard in our notation, so that, by '$\Gamma : e \Downarrow \Delta : \_$', we are clarifying our lack of interest in the final value obtained after evaluating e in Γ. Write '$\Gamma : e \Downarrow_\Pi$' for '$\Gamma : e \Downarrow_\Pi \_ : \_$'. For a derivation $\Gamma : \_ \Downarrow$, call Γ the **original** heap. For derivations Π_1 and Π_2, notation $\Pi_2 \subset \Pi_1$ indicates that $\_ : \_ \Downarrow_{\Pi_1}$ contains $\_ : \_ \Downarrow_{\Pi_2}$. Due to space restrictions, rules of Definition II.7 are illustrated in a 'turnstile' fashion hereafter. For example, our (seq) rule will become*

$$(\Gamma : e_1 \Downarrow \Theta : v_1, \Theta : e_2 \Downarrow \Delta : v_2) \vdash^{(\mathbf{seq})} \Gamma : e_1\ \mathsf{seq}\ e_2 \Downarrow \Delta : v_2.$$

III. Heaps and Derivations

In this section we first consider a few fundamental properties of our system which we use later. Next, we provide some results about the interplay between a heap Γ and the derivations Π in which Γ is the original heap.

A. Fundamental Properties

The particularly important result of this section is Determinism (Theorem III.2). A variation of Theorem III.2 is correct for Launchbury's system [1], as well as that of van Eekelen and de Mol [2,9,11]. The rest of results in this section, although not as important as Determinism, will be handy later.

Notation III.1. *Write $\Pi =_{\mathsf{let}\alpha} \Pi'$ when Π and Π' are equal up to renaming* let*-bound variables.*

The following theorem shows that derivations are unique up to an easy renaming, and reduction is deterministic:

Theorem III.2. *If $\Gamma : e \Downarrow_\Pi \Delta : v$ and $\Gamma : e \Downarrow_{\Pi'} \Delta' : v'$ then $\Pi =_{\mathsf{let}\alpha} \Pi'$, $\Delta = \Delta'$, and $v = v'$.*

Lemma III.3. *Suppose that $\Gamma(x)$ is a value v_x and $\Gamma : e \Downarrow_\Pi \Delta : v$. Then, for every heap Ξ in Π, $\Xi(x) = v_x$. In particular, $\Delta(x) = v_x$.*

Lemma III.4. *If $\Gamma(x) = v$ then $\Gamma : x \Downarrow \Gamma : v$.*

B. The Interplay

In this section, we divide $(\mathbf{var_x})$ instances into two groups: Trivial and Non-Trivial (Definition III.5). We then formalise our intuition that trivial instances do not change the heap (Lemma III.7) whilst non-trivial ones do (Lemma III.8). These results are then used to prove that each derivation contains at most one non-trivial $(\mathbf{var_x})$ instance (Theorem III.9).

Definition III.5. *Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Write $V(\Pi)$ for the set of $x \in dom(\Gamma)$ such that Π contains an instance of $(\mathbf{var_x})$. Suppose Π contains an instance of $(\mathbf{var_x})$ like*

$$\Gamma' : e_x \Downarrow_{\Pi'} \Gamma' : v_x \vdash^{(\mathbf{var_x})} (\Gamma', x \mapsto e_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x.$$

*Call the instance **trivial** when Π' consists of a single instance of (val) (equivalently, when $e_x = v_x$). Call the instance **non-trivial** otherwise.*

Definition III.6. *Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. Define $diff(\Pi) = \{x \in dom(\Gamma) \mid \Gamma(x) \neq \Delta(x)\}$.*

Lemma III.7. *Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$ and $x \in V(\Pi)$. Then, $x \notin diff(\Pi)$ iff $\Gamma(x)$ is a value.*

Lemma III.8. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then:* $x \in \mathit{diff}(\Pi) \Leftrightarrow \Gamma(x)$ *is not a value and* $\Delta(x)$ *is a value* $\Leftrightarrow \Pi$ *contains a non-trivial instance of* $(\mathbf{var_x})$ *for some* $x \in \mathit{dom}(\Gamma)$.

Finally, here is a less obvious result:

Theorem III.9. *Suppose* $\Gamma : \_ \Downarrow_\Pi$. *For each* $x \in \mathit{dom}(\Gamma)$, *there is at most one non-trivial instance of* $(\mathbf{var_x})$ *in* Π.

Proof: Rule-based induction and Lemmata III.7 and III.8. ∎

C. Essential Parts of a Heap for a Derivation

The mission of this section is to provide criteria for distinguishing the essential parts of a heap for a derivation. That is, the parts of a heap Γ without which a derivation $\Gamma : e \Downarrow \Delta : v$ is not derivable. Theorem III.14 formalises this.

Definition III.10. *For an* $x \in \mathit{dom}(\Gamma)$, *define* $\Gamma[x \mapsto e]$ *as:*

- $\Gamma[x \mapsto e](x) = e$
- $\Gamma[x \mapsto e](y) = \Gamma(y)$ *when* $(y \neq x)$.

Furthermore,

- $\Gamma[x_i \mapsto e_i]_{i=1}^{1} = \Gamma[x_1 \mapsto e_1]$, *and*
- $\Gamma[x_i \mapsto e_i]_{i=1}^{n} = (\Gamma[x_i \mapsto e_i]_{i=1}^{n-1})[x_n \mapsto e_n]$.

Suppose $\Gamma : e \Downarrow_\Pi \Delta : v$. *Then, define* $\Pi[x \mapsto e]$ *as the labelled tree obtained from* Π *by replacing every heap* Θ *in* Π *with* $\Theta[x \mapsto e]$ *if* $x \in \mathit{dom}(\Theta)$; *otherwise, leave* Θ *unchanged.*

Lemma III.11. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and* $\Gamma(x)$ *is a value* v_x. *Then,* $\Pi[x \mapsto v_x] = \Pi$.

Lemma III.12. *For any heap* Γ *and variable* y *such that* $y \notin \mathit{dom}(\Gamma)$, $(\Gamma, y \mapsto e_y)[x \mapsto e_x] = (\Gamma[x \mapsto e_x], y \mapsto e_y)$.

Lemma III.13. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *where* $x \in \mathit{dom}(\Gamma) \setminus V(\Pi)$. *Then for any* e_x, $\Gamma[x \mapsto e_x] : e \Downarrow_{\Pi[x \mapsto e_x]} \Delta[x \mapsto e_x] : v$, $\mathit{diff}(\Pi[x \mapsto e_x]) = \mathit{diff}(\Pi)$, *and* $V(\Pi[x \mapsto e_x]) = V(\Pi)$.

Proof: By inductive transformation of Π into $\Pi[x \mapsto e_x]$. ∎

Theorem III.14 expresses that if $\Gamma : e \Downarrow_\Pi \Delta : v$, then Π only depends on the Γ restricted to $V(\Pi)$:

Theorem III.14. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *and suppose* $\Gamma(z) = \Gamma'(z)$ *for every* $z \in V(\Pi)$. *Then there exist* Δ' *and* Π' *such that:* $\Gamma' : e \Downarrow_{\Pi'} \Delta' : v$, $\mathit{diff}(\Pi') = \mathit{diff}(\Pi)$, $V(\Pi') = V(\Pi)$, *and* $\Delta'(z) = \Delta(z)$ *for every* $z \in V(\Pi)$.

IV. Atomic Variables and INMB

A. Individual Properties

Now that we have identified trivial and non-trivial $(\mathbf{var_x})$ instances, and identified the essential parts of a heap for a derivation, we can move on to consider a notion of *unit of change*. Intuitively, these are computations which do cause changes in a heap, but cause a minimal change. For this, we particularly underpin the variables evaluation of which causes such an impact — which we call *atomic*. Definition IV.1 makes this formal, and the rest of this section explores the characteristic properties of atomic variables. These properties will then be used in §IV-B to prove our main results.

Definition IV.1. *Call* x ***atomic*** *in* Γ *when there exist* Δ_x, v_x, *and* Π_x *such that* $\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ *and* $\mathit{diff}(\Pi_x) = \{x\}$. *Write* $\mathit{atomic}(\Gamma)$ *for the set of atomic variable symbols in* Γ.

So, x is atomic in Γ when it can be computed without **affect**ing the rest of the heap. Therefore, as Corollary IV.3 will prove, if $\Gamma(x)$ is a value then x is not atomic in Γ. An important observation is that although the evaluation of an atomic variable does not affect the rest of the heap, other bindings of the heaps may well be required for its evaluation. For example, for $\Gamma = \{x_1 \mapsto (\lambda x.x)\ x_2,\ x_2 \mapsto \lambda x.x\}$, the variable x_1 is atomic in Γ. This is because for $\Gamma : x_1 \Downarrow_\Pi \_ : \lambda x.x$, the only affected binding is that of x_1 itself, i.e., $\mathit{diff}(\Pi) = \{x_1\}$. This is despite the fact that $x_2 \in V(\Pi)$ too.

It turns out that the notion of atomic variables is extremely useful in the proofs to follow. Note that a variable is said to be atomic with respect to a *heap*, as opposed to a derivation.

We will prefer the following slightly more succinct characterisation of atomicity:

Lemma IV.2. $x \in \mathit{atomic}(\Gamma)$ *iff there exist* v_x *and* Π_x *such that* $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ *and* $\mathit{diff}(\Pi_x) = \{x\}$.

Corollary IV.3. *Suppose* $\Gamma(x)$ *is a value. Then,* $x \notin \mathit{atomic}(\Gamma)$.

Proof: Take $\Gamma(x) = v_x$ for some value v_x. By Lemma III.4, $\Gamma : x \Downarrow_\Pi \Gamma : v_x$. Therefore, $\mathit{diff}(\Pi) = \varnothing$. In particular, $\mathit{diff}(\Pi) \neq \{x\}$. By Lemma IV.2, $x \notin \mathit{atomic}(\Gamma)$. ∎

Lemma IV.4 states that atomicity does not change upon evaluation when the variable is not touched. On the other hand, Lemma IV.5 explains when, from atomicity of a variable in one heap, we can deduce its atomicity in another heap.

Lemma IV.4. *Suppose* $x \in \mathit{atomic}(\Gamma)$ *and* $\Gamma : x \Downarrow \_ : v_x$. *Suppose also* $\Gamma : e \Downarrow_\Pi \Delta : v$, *and* $x \notin V(\Pi)$. *Then,* $x \in \mathit{atomic}(\Delta)$, *and* $\Delta(x) = v_x$.

Lemma IV.5. *Suppose* $\Gamma : x \Downarrow_{\Pi_x}$ *and* $\Gamma(z) = \Gamma'(z)$ *for every* $z \in V(\Pi_x)$. *Then,* $x \in \mathit{atomic}(\Gamma)$ *implies* $x \in \mathit{atomic}(\Gamma')$.

Proof: Using Theorem III.14. ∎

B. Atomic Variables and Derivations

This section provides three pivotal results which validate INMB, a.k.a., induction on the size of $\mathit{diff}(\Pi)$ for our operational semantics: Theorems IV.9 and IV.10, respectively, explain how to *remove* and *restore* an atomic variable from/in a derivation to obtain another (valid) derivation. Theorem IV.11 proves that any derivation either manipulates no bindings, or, of the manipulated bindings at least one binds an atomic variable in the original heap. (See Notation II.9 for the definition of original heap.) We start by providing a new way of transforming derivations with special care on atomic variables (Definition IV.6). On paying special attention to atomic variables, we are following [5] rather than [3,4].

Definition IV.6. *Suppose* $\Gamma : e \Downarrow_\Pi \Delta : v$ *such that* $x \in V(\Pi)$. *Suppose also that* $x \in \mathit{atomic}(\Gamma)$, *so that in particular*

$\Gamma : x \Downarrow_{\Pi_x} \Delta_x : v_x$ *for some* Π_x, Δ_x, *and* v_x. *We define a labelled tree* $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ *by inductively transforming* Π *based on its final rule. (Following our convention, we do not distinguish between* $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta[x \mapsto v_x] : v$ *and* $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ *when appropriate.) For each subcase,*

- *when* $x \notin dom(\Gamma)$ *or* $x \in V(\Pi) \setminus atomic(\Gamma)$, *define*

$$\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi. \tag{IV.1}$$

- *when* $x \notin V(\Pi)$, *define*

$$\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi[x \mapsto v_x]. \tag{IV.2}$$

The subcases where $x \in atomic(\Gamma) \cap V(\Pi)$ *are defined below:*

- $(\mathbf{var_x})$. Π *takes the form*
$$\Gamma' : e_x \Downarrow_{\_} \Gamma' : v_x \Big|^{(\mathbf{var_x})} (\Gamma', x \mapsto e_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x.$$
Define $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ *to be*
$$(\Big|^{(\mathbf{val})} \Gamma' : v_x \Downarrow \Gamma' : v_x) \Big|^{(\mathbf{var_x})} (\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x.$$
- $(\mathbf{let})$ *and* $(\mathbf{var_y})$ *for* y *other than* x. Π *takes the form*
$$\Gamma' : e' \Downarrow_{\Pi'} \Delta' : v' \Big|^{(\mathbf{r})} \Gamma : e \Downarrow \Delta : v$$
where $(\mathbf{r}) \in \{(\mathbf{var_y}), (\mathbf{let})\}$. *Define* $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ *to be*
$$\Gamma'[x \mapsto v_x] : e' \Downarrow_{\Pi'\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta'[x \mapsto v_x] : v' \Big|^{(\mathbf{r})} \Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v.$$
- $(\mathbf{app})$ *and* $(\mathbf{seq})$. Π *takes the form*
$$(\Gamma : e_1 \Downarrow_{\Pi_1} \Theta : v_1, \Theta : e_2 \Downarrow_{\Pi_2} \Delta : v_2) \Big|^{(\mathbf{r})} \Gamma : e \Downarrow \Delta : v$$
where $(\mathbf{r}) \in \{(\mathbf{app}), (\mathbf{seq})\}$. *Define* $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ *to be*
$$(\Gamma[x \mapsto v_x] : e_1 \Downarrow_{\Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Theta[x \mapsto v_x] : v_1, \Theta[x \mapsto v_x] : e_2 \Downarrow_{\Pi_2\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta[x \mapsto v_x] : v_2) \Big|^{(\mathbf{r})} \Gamma[x \mapsto v_x] : e \Downarrow \Delta[x \mapsto v_x] : v.$$

Note that $(\mathbf{val})$ *can never be the final rule when* $x \in atomic(\Gamma) \cap V(\Pi)$ *because* $V(\Pi) = \varnothing$ *in this case. Therefore, the recipe for this case is* (IV.2) *where* $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi[x \mapsto v_x] = \Pi$ *especially. (See Definition III.10.)*

Remark IV.7. The following points about the $(\mathbf{var_x})$ case in Definition IV.6 are worth considering:

- Recall from Theorem III.9 that there will be at most one non-trivial instance of $(\mathbf{var_x})$ in Π. In fact, the only difference between Definition IV.6 and Definition III.10 is that the former replaces the unique non-trivial instance of $(\mathbf{var_x})$ for an atomic variable x with a trivial one, whilst the latter does not.
- Suppose $\Gamma : x \Downarrow$ and $\Gamma : e \Downarrow_{\Pi}$ where $\Gamma' : x \Downarrow_{\Pi'}$ and $\Pi' \subset \Pi$. (See Notation II.9.) The fact that $\Gamma : x \Downarrow$ guarantees that x does indeed reduce in Γ'. Furthermore, as seen in Lemma IV.5, if $(\mathbf{val})$ is not the only rule used in Π' , then x is atomic in Γ' too.
- Because $x \in atomic(\Gamma)$, by Lemma IV.2, for any instance of $(\mathbf{var_x})$ in Π like $(\Gamma', x \mapsto \_) : x \Downarrow (\Delta', x \mapsto \_) : v_x$, we are guaranteed that $\Gamma' = \Delta'$. Note that this includes both trivial and non-trivial instances of $(\mathbf{var_x})$.

Lemma IV.8. *Suppose* $\Gamma : e \Downarrow_{\Pi} \Delta : v$ *and* $x \in V(\Pi) \cap atomic(\Gamma)$. *For any sub-derivation* Π' *of* Π *which contains no non-trivial instance of* $(\mathbf{var_x})$, $\Pi'\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi'[x \mapsto v_x]$.

Proof: Suppose $\Gamma' : \_ \Downarrow_{\Pi'}$. There are two cases:

- $x \notin V(\Pi')$. The result is correct by construction. (See (IV.2).)
- $x \in V(\Pi')$. By construction (Definition III.6), the fact that Π' contains no non-trivial instances of $(\mathbf{var_x})$, means that $x \notin diff(\Pi')$. Therefore,
  - on one hand, by Lemma IV.2, $x \notin atomic(\Gamma')$. Hence, by construction $\Pi'\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi'$. (See (IV.1).)
  - on the other hand, by Lemma III.7, $\Gamma'(x)$ is a value like v_x. Hence, by Lemma III.11, $\Pi'[x \mapsto v_x] = \Pi'$.

■

Theorem IV.9. *Suppose* $\Gamma : e \Downarrow_{\Pi} \Delta : v$ *and* $x \in V(\Pi)$. *Suppose* $x \in atomic(\Gamma)$; *so in particular* $\Gamma : x \Downarrow_{\Pi_x} \Gamma[x \mapsto v_x] : v_x$ *for some* v_x. *Then* $\Delta(x) = v_x$ *and* $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : v$. *Furthermore,* $diff(\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle) = diff(\Pi) \setminus \{x\}$ *and* $V(\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle) \cup V(\Pi_x) = V(\Pi)$.

Proof: Induction on Π based on its final rule:

- $(\mathbf{var_x})$. Π takes the form
$$\Gamma' : e \Downarrow_{\_} \Delta' : v \Big|^{(\mathbf{var_x})} (\Gamma', x \mapsto e) : x \Downarrow (\Delta', x \mapsto v) : v$$
where $\Gamma = (\Gamma', x \mapsto e)$ and $\Delta = (\Delta', x \mapsto v)$. (Note that, as also notified in Remark IV.7, $\Delta' = \Gamma'$ here.) By construction (Definition IV.6), in this case, $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ will take the form
$$(\Big|^{(\mathbf{val})} \Gamma' : v_x \Downarrow \Gamma' : v_x) \Big|^{(\mathbf{var_x})} (\Gamma', x \mapsto v_x) : x \Downarrow (\Gamma', x \mapsto v_x) : v_x.$$
Observe that, by Determinism (Theorem III.2), $\Pi =_{\mathsf{let}\alpha} \Pi_x$. (See Notation III.1 for $=_{\mathsf{let}\alpha}$.) Thus, $x \in atomic(\Gamma)$ implies $diff(\Pi) = diff(\Pi_x) = \{x\}$ by Lemma IV.2, and
$$\varnothing = diff(\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle) = diff(\Pi) \setminus \{x\}$$
$$V(\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle) \cup V(\Pi_x) = \{x\} \cup V(\Pi) = V(\Pi).$$

- $(\mathbf{app})$. Π takes the form
$$(\Gamma : e' \Downarrow_{\Pi_1} \Theta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ \lambda z.e'', \Theta : (\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e'')[y/z] \Downarrow_{\Pi_2} \Delta : v) \Big|^{(\mathbf{app})} \Gamma : e'\ z \Downarrow \Delta : v$$
where $n \geq 0$. There are now two cases:
  - $x \in V(\Pi_1)$. By inductive hypothesis $\Theta(x) = v_x$ and $\Gamma[x \mapsto v_x] : e' \Downarrow_{\Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Theta : \lambda z.e''$. By Lemma III.3, $\Delta(x) = \Theta(x) = v_x$. On the other hand, because $\Theta(x) = v_x$, by Corollary IV.3, $x \notin atmoic(\Pi_2)$. Therefore, if $x \in V(\Pi_2)$, then $\Pi_2\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi_2$ by construction. (See (IV.1).) If $x \notin V(\Pi_2)$, then $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi[x \mapsto v_x]$ by construction. (See (IV.2).) However, given that $\Theta(x) = v_x$, by Lemma III.11, $\Pi_2[x \mapsto v_x] = \Pi_2$. Hence, in both cases, by the $(\mathbf{app})$ case of Definition IV.6, we can combine $\Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle$ with Π_2

to get $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ and observe that $\Gamma[x \mapsto v_x] : e'\ y \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : v$. The result follows.

- $x \notin V(\Pi_1)$. It follows that $x \in V(\Pi_2)$. By Lemma III.13, $\Gamma[x \mapsto v_x] : e' \Downarrow_{\Pi_1[x \mapsto v_x]} \Theta[x \mapsto v_x] : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda z.e''$. Moreover, by construction, $\Pi_1[x \mapsto v_x] = \Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle$. (See (IV.2).) Therefore, $\Gamma[x \mapsto v_x] : e' \Downarrow_{\Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Theta[x \mapsto v_x] : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda z.e''$. On the other hand, by Lemma IV.4, $x \in \mathit{atomic}(\Theta)$, and $\Theta : x \Downarrow \Theta_x : v_x$ for some Θ_x. By inductive hypothesis $\Delta(x) = v_x$ and $\Theta[x \mapsto v_x] : (\mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e'')[y/z] \Downarrow_{\Pi_2\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : v$. We combine $\Pi_1\langle\!\langle x \mapsto v_x \rangle\!\rangle$ with $\Pi_2\langle\!\langle x \mapsto v_x \rangle\!\rangle$ to get $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$ and observe that $\Gamma[x \mapsto v_x] : e'\ y \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : v$. (Note that, by the (app) case of Definition IV.6, this combination is indeed $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$.) The result follows.

- (let). Π takes the form

$$\dfrac{(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi'} (\Delta, x_i \mapsto e_i')_{i=1}^n : v}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i'\}_{i=1}^n\ \mathsf{in}\ v}\ (\mathbf{let})$$

Observe that, by Lemma IV.5, $x \in \mathit{atomic}(\Gamma)$ implies $x \in \mathit{atomic}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$. Therefore, by inductive hypothesis:

- $(\Gamma, x_i \mapsto e_i)_{i=1}^n[x \mapsto v_x] : e \Downarrow_{\Pi'\langle\!\langle x \mapsto v_x \rangle\!\rangle} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$ which, by Lemma III.12, can be rewritten as $(\Gamma[x \mapsto v_x], x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi'\langle\!\langle x \mapsto v_x \rangle\!\rangle} (\Delta, x_i \mapsto e_i')_{i=1}^n : v$. Extending with (let), we deduce $\Gamma[x \mapsto v_x] : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : \mathsf{let}\ \{x_i{=}e_i'\}_{i=1}^n\ \mathsf{in}\ v$. By the (let) case of Definition IV.6, we know that this combination is indeed $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle$, as desired.
- $(\Delta, x_i \mapsto e_i')_{i=1}^n(x) = v_x$ which, given that x_i's are fresh, implies $\Delta(x) = v_x$.

Other cases are no harder. ■

Theorem IV.10. *Suppose $x \in \mathit{atomic}(\Gamma)$ and $\Gamma : x \Downarrow \_ : v_x$. Suppose $\Gamma\langle\!\langle x \mapsto v_x \rangle\!\rangle : e \Downarrow_{\Pi'} \Delta : v$ and $x \in V(\Pi')$. Then $\Gamma : e \Downarrow_{\Pi} \Delta : v$ such that $\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle = \Pi'$.*

Proof: Similar to Theorem IV.9 except that, this time, we inductively transform $\boldsymbol{\Pi'}$ to Π. ■

Theorem IV.11. *Suppose $\Gamma : e \Downarrow_{\Pi} \Delta : v$. Then exactly one of the following possibilities holds: (1) $\mathit{diff}(\Pi) = \varnothing$. Or, (2) there exists an $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$.*

Proof: By induction on Π based on the final rule:

- (val). Π takes the form

$$\dfrac{}{\Gamma : v \Downarrow \Gamma : v}\ (\mathbf{val})$$

in which case $\mathit{diff}(\Pi) = \varnothing$.

- (let). Π takes the form

$$\dfrac{(\Gamma, x_i \mapsto e_i)_{i=1}^n : e \Downarrow_{\Pi'} (\Delta, x_i \mapsto e_i')_{i=1}^n : v}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i'\}_{i=1}^n\ \mathsf{in}\ v}\ (\mathbf{let})$$

If $\mathit{diff}(\Pi') = \varnothing$, then $(\Gamma, x_i \mapsto e_i)_{i=1}^n(y) = (\Delta, x_i \mapsto e_i')_{i=1}^n(y)$ for every $y \in \mathit{dom}((\Gamma, x_i \mapsto e_i)_{i=1}^n)$. Because x_is are fresh (Definition II.7), for every $z \in \mathit{dom}(\Gamma)$, this implies $\Gamma(z) = \Delta(z)$. Hence, $\mathit{diff}(\Pi) = \varnothing$ too.
Suppose $\mathit{diff}(\Pi') \neq \varnothing$. Then, by inductive hypothesis, there exists a variable x which is atomic in $(\Gamma, x_i \mapsto e_i)_{i=1}^n$. If $x \in \mathit{dom}(\Gamma)$, then, by Lemma IV.5, x is also atomic in Γ, as desired. If every such x is in $\{x_1, \ldots, x_n\}$, then $\mathit{diff}(\Pi) = \varnothing$ and the result follows.

Other cases are no harder. ■

V. The New Inductive Principle and Uses

Induction on the number of the manipulated bindings (Inmb) was first introduced in [3]. Later on, [4] completed their proofs and showed the validity of this new inductive principle for their restricted operational semantics. Finally, [5] proved the correctness of this inductive principle for a non-restricted operational semantics for lazy evaluation with selective strictness. Neither [3] nor its two succors [4,5] however outlined a mathematical formulation of induction on the number of manipulated bindings. Here is our formulation of this principle:

$$\dfrac{\begin{array}{c}[P(\Pi);\ \mathit{diff}(\Pi) = \varnothing] \\ \wedge \\ [\forall k.(P(\Pi);\ \|\mathit{diff}(\Pi)\| = k) \Rightarrow \\ (P(\Pi);\ \|\mathit{diff}(\Pi)\| = k+1)]\end{array}}{\forall n.\ P(\Pi);\ \|\mathit{diff}(\Pi)\| = n}\ (\textsc{Inmb})$$

In words, (Inmb) states that: "$P(\Pi)$ holds if: (1) $P(\Pi)$ holds when no binding gets manipulated over derivation Π. And, (2) validity of $P(\Pi)$ when there are k manipulated bindings in derivation Π implies validity of $P(\Pi)$ when there are $k+1$ manipulated bindings."

As an example for Inmb, we next prove Theorem V.4 using this technique. All the preliminary results and definitions are from [4] so we drop the proofs except for Theorem V.4. We start by introducing a notion of equivalence *between heaps*:

Definition V.1. *Define $\Gamma_1 \approx \Gamma_2$ by:*

$$\forall e.\forall v.\Big((\exists \Delta_1.\Gamma_1 : e \Downarrow \Delta_1 : v) \Leftrightarrow (\exists \Delta_2.\Gamma_2 : e \Downarrow \Delta_2 : v)\Big).$$

Lemma V.2. *$\approx$ is an equivalence relation.*

Lemma V.3. *Suppose $x \in \mathit{atomic}(\Gamma)$, so in particular $\Gamma : x \Downarrow \Gamma[x \mapsto v_x] : v_x$ for some v_x. Then $\Gamma \approx \Gamma[x \mapsto v_x]$.*

Theorem V.4. *If $\Gamma : e \Downarrow_{\Pi} \Delta : v$ then $\Gamma \approx \Delta$.*

Proof: Inmb. There are two cases:

- $\mathit{diff}(\Pi) = \varnothing$. The result follows by reflexivity of $\approx$ (Lemma V.2).
- $\mathit{diff}(\Pi) \neq \varnothing$. By Theorem IV.11, there exists an $x \in \mathit{diff}(\Pi)$ such that $x \in \mathit{atomic}(\Gamma)$, so in particular $\Gamma : x \Downarrow \Gamma[x \mapsto v_x] : v_x$ for some v_x. By Lemma V.3, $\Gamma \approx \Gamma[x \mapsto v_x]$. By Theorem IV.9, $\Gamma[x \mapsto v_x] : e \Downarrow_{\Pi\langle\!\langle x \mapsto v_x \rangle\!\rangle} \Delta : v$ (Definition IV.6). By Theorem IV.9, $\mathit{diff}(\Pi[x \mapsto v_x]) = \mathit{diff}(\Pi) \setminus \{x\}$. By inductive hypothesis, $\Gamma[x \mapsto v_x] \approx \Delta$. We use transitivity of $\approx$ (Lemma V.2).

■

Using a more complicated version of our (INMB), [3,4] prove a number of observational equivalences between lazy programs in presence of selective strictness. For demonstration, we only present two types of observational equivalences they define:

Definition V.5. *Define* $e_1 \approxeq_h e_2$ *by:*

$$\forall \Gamma, \Delta.\ (\exists v_1.\ \Gamma : e_1 \Downarrow \Delta : v_1) \Leftrightarrow (\exists v_2.\ \Gamma : e_2 \Downarrow \Delta : v_2).$$

Intuitively, $e_1 \approxeq_h e_2$ when e_1 and e_2 compute the same final heaps, given the same initial heap; $\approxeq_h$ does not examine the final values, which may differ.

Definition V.6. *Define* $e_1 \approxeq_s e_2$ *by:*

$$\forall \Gamma, v, \Delta.\ (\Gamma : e_1 \Downarrow \Delta : v) \Leftrightarrow (\Gamma : e_2 \Downarrow \Delta : v).$$

Intuitively, $e_1 \approxeq_s e_2$ when, given the same initial heap, e_1 and e_2 compute the same final value and final heap.

Two interesting examples proved in [3] and [4] using INMB are then:

1) $\forall e_1, e_2.\ e_1 \ \mathsf{seq}\ e_2 \approxeq_h e_2 \ \mathsf{seq}\ e_1$.
2) $\forall e_1, e_2, e_3.\ e_1 \ \mathsf{seq}\ e_2 \ \mathsf{seq}\ e_2 \approxeq_s e_2 \ \mathsf{seq}\ e_1 \ \mathsf{seq}\ e_3$

VI. Related Work

Ong and Abramsky [12]–[14] studied call-by-need, but they did not capture sharing. Both [15] and [1] present a big-step semantics for call-by-need. Ariola et al. [8,16,17] considered call-by-need (as a calculus) rather than lazy evaluation, but did not capture recursive let-bindings. We choose to build on Launchbury's big step semantics [1] which models lazy evaluation at the level of abstraction suitable for big-step observational equivalence.

Earlier work addresses issues raised by selective strictness. Harrison et al. [18] provides a *calculational* approach, in which every detail of how to implement a HASKELL interpreter in HASKELL itself is explained. Later, [19] uses *P*-logic [20] to specify and verify properties of programs with selective strictness. In [2], van Eekelen and de Mol extend Launchbury's semantics to selective strictness. Other related works of this group are [9] and [11]. We choose to directly reason on derivation trees of an operational semantics which is a variation of that of [2]. Johann and Voigtländer [21] study free theorems in presence of selective strictness. [22]–[26] all extend this work but for call-by-name rather than call-by-need. Finally, Hidalgo-Herrero [27] offers a denotational semantics in which they can prove similar identities to the ones proved in [3] and [4].

The study of observational equivalence between lazy programs in presence of selective strictness starts in Gabbay et al. [3]. Haeri [4] fills all the missing details in [3]. Both the operational semantics of [3] and that of [4] however have a restricted expressiveness. Whilst retaining all the interesting features of [3] and [4], Haeri [5] solved the expressiveness problem. Haeri [5] accomplishes this by carrying the let-bindings around with the evaluated expression rather than retaining them in the returned heap. Finally, Haeri [6] provides yet another operational semantics based on that of van Eekelen and de Mol [2]. Haeri proves equivalences between the operational semantics in [5] and [6]. His latter operational semantics is both simpler and proved to be as expressive as the latter operational semantics.

An improvement in [5] and [6] (and thus this paper) over [3] and [4] is that the former group does not force the side condition $x_i \notin fv(v)$ for (**let**). Hence, a group of arguably reasonable lazy programs which do not reduce in [3] and [4] do reduce in our system. As explained in [5, §6.2], let $y = \lambda x.x$ in $\lambda x.y$ is an example of this group where $y \in fv(\lambda x.y)$.

The motivation of [3] and [4] for adding the $x_i \notin fv(v)$ side condition (Fig. 2) is to prevent the reduction of certain expressions from midway failure. $x_i \notin fv(v)$ aims this through illegalising the expressions their lack of bindings is expectable over the next evaluation stages. As explained in [5, §6.1], an example of such expressions is $e = \mathsf{let}\ x' = \lambda x.x\ \mathsf{in}\ (\mathsf{let}\ \{x_1 = \lambda x.x, x_2 = \lambda x.x_1\}\ \mathsf{in}\ x_2)\ x'$. In their system, without the $x_i \notin fv(v)$ side condition, the reduction of e would halt once it reaches the evaluation of x_1. This is because, at that point, x_1 is no longer bound in the respective heap. Lazy programs like e which do not reduce in [3] and [4] do reduce in our system.

Despite these differences, it is easy to prove the following result. Note that, the inverse implication is not correct.

Theorem VI.1. *Suppose* $\Gamma : e \Downarrow^{\mathcal{G}} \Delta : \lambda x.e'$. *Then,* $\Gamma : e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda x.e'$ *for some* $\{x_i\}_{i=1}^n$ *where* $n \geq 0$.

As proved in [6], our system is as expressive as that of van Eekelen and de Mol [2]:

Theorem VI.2. *Suppose* $\Gamma : e \Downarrow^{vd} (\Delta, x_i \mapsto e_i)_{i=1}^n : \lambda x.e'$ *such that* $dom(\Gamma) = dom(\Delta)$. *Then,* $\Gamma : e \Downarrow \Delta : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^n\ \mathsf{in}\ \lambda x.e'$.

VII. Conclusion and Future Work

In this paper, we extended Launchbury's model for lazy evaluation: In §II, we follow Ariola and Felleisen by including let-surrounded abstractions in values. Consequently, we end up having a new application rules. Besides, our (**let**) rule, in the same time that performs garbage-collection in the relevant heap, retains the (probably manipulated) let-bindings by returning let-surrounded values. As explained in §VI, our special form of garbage-collection, retains enough information for later reference. Consequently, we dismiss the restrictive side condition in (**let**$^{\mathcal{G}}$) which prevented [3] and [4] from fully modelling lazy evaluation. As a result, we accomplish full modelling of lazy evaluation in the same time that our operational semantics avoids heap growth of expressiveness upon evaluation of let-expressions. Finally, our operational semantics borrows a variation of van Eekelen and de Mol's (**StrictLet**) [2,11] for selective strictness. Our operational semantics is as expressive as that of van Eekelen and de Mol.

Different interplays between heaps and derivations are discussed in §III. Atomic variables and their individual properties are investigated in §IV-A. The most important part of this paper is §IV-B where the pivotal role atomic variables play

Syntax: $e ::= x \mid \lambda x.e \mid e\,x \mid \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \mid e\ \mathsf{seq}\ e \qquad v ::= \lambda x.e$

Semantics:

$$\frac{}{\Gamma : \lambda x.e \Downarrow^{\mathcal{G}} \Gamma : \lambda x.e}\ (\mathbf{lam}) \qquad \frac{\Gamma : e \Downarrow^{\mathcal{G}} \Theta : \lambda y.e' \quad \Theta : e'[x/y] \Downarrow^{\mathcal{G}} \Delta : v}{\Gamma : e\,x \Downarrow^{\mathcal{G}} \Delta : v}\ (\mathbf{app})$$

$$\frac{(\Gamma, x_i{\mapsto}e_i)_{i=1}^{n} : e \Downarrow^{\mathcal{G}} (\Delta, x_i{\mapsto}e_i')_{i=1}^{n} : v \quad (x_i \notin fv(v), x_i \text{ fresh})}{\Gamma : \mathsf{let}\ \{x_i{=}e_i\}_{i=1}^{n}\ \mathsf{in}\ e \Downarrow^{\mathcal{G}} \Delta : v}\ (\mathbf{let}^{\mathcal{G}})$$

Fig. 2. Differing Parts of the Syntax and Operational Semantics of [3] and [4]

in derivations is studied. Proofs of §IV-B validate induction on the number of manipulated bindings (INMB) for our operational semantics. INMB is outlined in §V with a few interesting uses of it. Related work is finally reviewed in §VI.

Various routes can be taken for future work: Extending our work to parallel lazy languages like GPH [28] is one example. The study of observational approximation for certain conditions when the change in heap expressiveness is permissible is another option. For applications like [3,4], in proving observational equivalence between lazy programs, heaps very much play the customary role of *Contexts*. However, one might still consider investigating more powerful notions capable enough to cope with customary contexts too. For example, extending current technology not to distinguish between the lazy behaviour of $\lambda x.x+1$ and $\lambda x.1+x$ is desirable. Finally, investigating the validity of a context lemma [29] for the $\approx_s$ of [3,4] is also worth considering.

ACKNOWLEDGEMENTS

This paper is particularly inspired by the constructive discussions the author has had with Prof Yolanda Ortega-Mallén, Dr Murdoch J. Gabbay, and Prof Phil W. Trinder. Special thanks to Dr Brian Campbell for reviewing our paper. We would also like to thank Prof Don Sannella, Dr Jeremy Gibbons, and Dr Patrik Jansson for their suitable comments.

REFERENCES

[1] J. Launchbury, "A natural semantics for lazy evaluation," in *POPL '93*. ACM, 1993, pp. 144–154.

[2] M. van Eekelen and M. de Mol, "Mixed lazy/strict graph semantics," in *Implementation and Application of Func. Lang., 16th Int'l Workshop*, ser. TR 0408, Christian-Albrechts-Universität zu Kiel, C. Grelck and F. Huch, Eds., Luebeck, Germany, Sep. 2004, pp. 245–260.

[3] M. J. Gabbay, S. H. Haeri, Y. Ortega-Mallén, and P. Trinder, "Reasoning about selective strictness: operational equivalence, heaps and call-by-need evaluation, new inductive principles," 2009, submitted to ICFP'09.

[4] S. H. Haeri, "Reasoning about selective strictness: Operational equivalence, heaps and call-by-need evaluation, new inductive principles," 2009, MPhil Thesis, available from the author.

[5] ——, "Observational Equivalence between Lazy Programs in Presence of Selective Strictness," MuSemantik, Technical Report, 2009, available from the author.

[6] ——, "An Improved Unrestricted Operational Semantics for Lazy Evaluation with Selective Strictness," MuSemantik, Technical Report, 2010, available from the author.

[7] P. Sestoft, "Deriving a lazy abstract machine," *J. of Func. Prog.*, vol. 7, no. 3, pp. 231–264, 1997.

[8] Z. Ariola and M. Felleisen, "The call-by-need λ-calculus," *J. of Func. Prog.*, vol. 7, no. 3, pp. 265–301, 1997.

[9] M. van Eekelen and M. de Mol, "Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics," in *Proc. of the 14th Int'l Workshop on the Implementation of Func. Lang.*, ser. TR 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, R. Peña, Ed., Madrid, Spain, Sep. 2002, pp. 357–373.

[10] R. Plasmeijer and M. van Eekelen, *Concurrent* CLEAN *Language Report (version 2.0)*, December 2001.

[11] M. van Eekelen and M. de Mol, *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*. Radboud University Nijmegen, 2007, ch. Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101.

[12] C.-H. L. Ong, "The lazy lambda calculus: An investigation in the foundations of functional programming," Ph.D. dissertation, Imperial College of Science and Technology, University of London, May 1988.

[13] S. Abramsky, "The Lazy Lambda Calculus," in *Research Topics in Func. Prog.*, D. A. Turner, Ed. Addison Wesley, 1990, pp. 65–116.

[14] S. Abramsky and C.-H. L. Ong, "Full abstraction in the lazy lambda calculus," *Information and Computation*, vol. 105, no. 2, pp. 159–267, Aug. 1993.

[15] J. Seaman and S. Purushothaman, "An operational semantics of sharing in lazy evaluation," *Sci. of Comp. Prog.*, vol. 27, no. 3, pp. 289–322, 1996.

[16] Z. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler, "A call-by-need lambda calculus," in *POPL '95*. New York, NY, USA: ACM, 1995, pp. 233–246.

[17] J. Maraist, M. Odersky, and P. Wadler, "The call-by-need λ-calculus," *J. of Func. Prog.*, vol. 8, no. 3, pp. 275–317, 1998.

[18] W. Harrison, T. Sheard, and J. Hook, "Fine control of demand in Haskell," in *MPC'02, Dagstuhl, Germany*, ser. Lecture Notes in Computer Science, vol. 2386. Springer Verlag, July 2002, pp. 68–93.

[19] W. L. Harrison and R. B. Kieburtz, "The logic of demand in Haskell," *J. of Func. Prog.*, vol. 15, no. 6, pp. 837–891, 2005.

[20] R. Kieburtz, "P-logic: property verification for Haskell programs," OGI, Tech. Rep., 2002.

[21] P. Johann and J. Voigtländer, "Free theorems in the presence of *seq*," in *POPL'04: Venice, Italy*, ser. SIGPLAN Notices, N. D. Jones and X. Leroy, Eds., vol. 39. ACM Press, Jan. 2004, pp. 99–110.

[22] ——, "The impact of *seq* on free theorems-based program transformations," *Fundamenta Informaticae*, vol. 69, no. 1–2, pp. 63–102, 2006.

[23] J. Voigtländer and P. Johann, "Selective strictness and parametricity in structural operational semantics," Technische Universität Dresden, Tech. Rep. TUD-FI06-02, Jun. 2006.

[24] ——, "Selective strictness and parametricity in structural operational semantics, inequationally," *Theo. Comp. Sci.*, vol. 388, no. 1–3, pp. 290–318, 2007.

[25] D. Seidel and J. Voigtländer, "Taming selective strictness," in *Arbeitstagung Programmiersprachen, Lübeck, Germany, Proceedings*, ser. Lecture Notes in Informatics, vol. 154. GI, Oct. 2009, pp. 2916–2930.

[26] ——, "Taming selective strictness," Technische Universität Dresden, Tech. Rep. TUD-FI09-06, Jun. 2009.

[27] M. Hidalgo-Herrero, "Semánticas formales para un lenguaje funcional paralelo," Ph.D. thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2004.

[28] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton-Jones, "Algorithm + Strategy = Parallelism," *J. of Func. Prog.*, vol. 8, no. 1, pp. 23–60, Jan. 1998.

[29] A. Gordon, "Bisimilarity as a theory of functional programming," in *Proc. of 11th Conf. on Math. Foundations of Prog. Semantics*, 1995.