Mixed Lazy/Strict Graph Semantics

Marko van Eekelen and Maarten de Mol

marko@cs.kun.nl, maartenm@cs.kun.nl,
Department of Software Technology, Nijmegen University, The Netherlands.

Abstract. Explicitly enforcing strictness is often used by functional programmers as an important tool for making applications fit time and space efficiency requirements. Few functional programmers however, are familiar with the consequences of explicitly enforcing strictness for formal reasoning about their programs. Some "folklore" knowledge has emerged but this is based on experience rather than on rigid proof. Up to now no formal model has been available for reasoning about enforcing strictness in denotational and operational semantics. This greatly hampered formal reasoning on mixed lazy/strict programs.
This paper presents a model for formal reasoning with enforced strictness based on John Launchbury's lazy graph semantics. Lazy graph semantics are widely considered to be essential for lazy functional programming languages. In this paper Launchbury's semantics are extended with an explicit strict let construct. Correctness and adequacy of our *mixed* lazy/strict graph semantics is proven. Using these mixed semantics we formalise and prove some of the available "folklore" knowledge.

1 Introduction and motivation

Strictness is a property of a function. A function f is strict in its argument if, according to the language semantics, $f\bot = \bot$, where $\bot$ is the symbol representing the undefined value. Strictness analysis is used to derive strictness properties for given programs. If the results of such an analysis are indicated via strictness annotations then these annotations do not change the semantics at all (assuming that the analysis is correct of course).

Therefore, it is often recommended to use strictness annotations only when strictness holds mathematically. For the cases of explicit strictness that have the *intention* to change the semantics, this recommendation is not sensible at all. Although it is seldom mentioned in papers and presentations, such explicit strictness that changes the semantics, is present in almost every lazy programming language (and in almost every program) that is used in real-world examples. In such programs, strictness is used:

– for improving the *efficiency of data structures* (e.g. strict lists),
– for improving the *efficiency of evaluation* (e.g. functions that are made strict in some arguments due to strictness analysis or due to the programmers annotations),

– for *enforcing the evaluation order* in interfacing with the outside world (e.g. an interface to an external C-call is defined to be strict in order to ensure that the arguments are fully evaluated before the external call is issued).

Language features that are used to denote this strictness include:

– type annotations (in functions and in data structures: Clean),
– special data structures (unboxed arrays: Clean, Haskell),
– special primitives (seq and deepSeq: Haskell),
– special inside implementations (monads: Haskell),
– special language constructs (let!, #!: Clean),
– special tools (strictness analyzer: Clean).

Implementers of real-world applications make it their job to know about strictness aspects, because without strictness annotations essential parts of their programs would not work properly. It is mandatory for the compiler to generate code that takes these annotations into account. For reasoning about these programs, however, they tend to forget strictness altogether. Usually, strictness is not taken into account in a formal graph semantics for a programming language. Disregarding strictness can lead to unexpected non-termination when programs are changed by hand or automatically transformed.

For reasoning with strictness, there is only little theory and guidance available so far. In this paper we develop an appropriate mixed denotational and operational semantics for formal reasoning about programs in a mixed lazy/strict context. For these semantics the required properties, such as correctness and computational adequacy will be proven (Sect. 2). As an example of the use of our mixed semantics we prove several folklore theorems about comparing lazy and strict graph semantics in Sect. 3. In Sect. 4 we formally prove that with mixed semantics it is possible to express in the language itself the semantical difference between Ω and $\lambda x.\Omega$ (while in Launchbury's model these two expressions can only be distinguished from outside the language). Finally, Sect. 5 and 6 discuss related and future work and give concluding remarks.

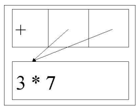
2 Mixed lazy/strict graph semantics

Since we consider graphs as an essential part of the semantics of a lazy language, we have chosen to extend Launchbury's graph semantics [6]. Cycles (using recursion), black hole detection, garbage collection and cost of computation can be analyzed formally using these semantics. Launchbury has proven that his operational graph rules are *correct* and *computationally adequate* with respect to the corresponding denotational semantics. Informally, correctness means that an expression which operationally reduces to a value will denotationally be equal to that value. Computational adequacy informally means that if the meaning of an expression is defined denotationally it is also defined operationally and vice-versa. We will prove that our mixed semantics are correct and computationally adequate in Sect. 2.5. Sections 2.1-2.4 introduce the required preliminaries.

2.1 Basic idea of Launchbury's natural graph semantics

Basically, sharing is represented as *let*-expressions. In contrast to creating a node for every application, nodes are created only for those parts of the expression that are to be shared as is illustrated below.

$$\begin{array}{l} let\ x = \ 3 * 7 \\ in\ x + x \end{array}$$



represents the graph on the right:

Graph reduction is formalised by a system of derivation rules. Graph nodes are represented by variable definitions in an environment. A typical graph reduction proof is given below. Each reduction step corresponds to applying a derivation rule (assuming extra arithmetic application rules; the standard rules are given in Sect. 2.4).

```
⎡{ } : let x = 3 * 7 in x + x
⎢ ⎡{ x ↦ 3 * 7 } : x + x
⎢ ⎢ ⎡{ x ↦ 3 * 7 } : x
⎢ ⎢ ⎢ ⎡{ x ↦ 3 * 7 } : 3 * 7
⎢ ⎢ ⎢ ⎢{ x ↦ 3 * 7 } : 21
⎢ ⎢ ⎢ ⎣*
⎢ ⎢ ⎢ { x ↦ 21 } : 21
⎢ ⎢ ⎣Var
⎢ ⎢ ⎡{ x ↦ 21 } : x
⎢ ⎢ ⎢{ x ↦ 21 } : 21
⎢ ⎢ ⎣Var
⎢ ⎢ { x ↦ 21 } : 42
⎢ ⎣+
⎢ { x ↦ 21 } : 42
⎣Let
```

2.2 Notational conventions.

We will use the following notational conventions:

- x, y, v, x_1 and x_n are variables,
- e, e', e_1, e_n, f, g and h are expressions,
- z and z' are *values* (i.e. expressions of the form $\lambda\ x.\ e$ and constants, when the language is extended with constants),
- the notation $\hat{z}$ stands for a renaming (α-conversion) of a value z such that *all* lambda bound and let-bound variables in z are replaced by fresh ones.
- Γ, Δ and Θ are taken to be heap variables (a heap is assumed to be a set of *variable bindings*, i.e. pairs of distinct variables and expressions),

– a binding of a variable x to an expression e is written as $x \mapsto e$,
– ρ, ρ', ρ_0 are *environments* (an environment is a function from variables to values),
– the judgment $\Gamma : e \Downarrow \Delta : z$ means that in the context of the heap Γ a term e reduces to the value z with the resulting set of bindings Δ,
– and finally σ and τ are taken to be derivation trees for such judgments.

2.3 Mixed lazy/strict expressions

We extend the expressions of Launchbury's system with a strict variant of recursive let-expressions. Experience with two different lazy programming languages that allow explicit enforcing of strictness (Haskell[4] and Clean[8]), seems to indicate that a non-recursive strict let-expression (non-recursiveness is then assumed to be enforced syntactically), is sufficient from the point of view of the programmer. From a semantic point of view it will turn out that allowing recursion does not impose any problems. Recursion is essential for the definition of cyclic graph structures. Therefore, we suggest to the designers of Haskell and Clean to allow recursion for their strictness constructs. This will give the programmer more means for explicit control of evaluation of cyclic structures.

In strict let-expressions only one variable can be defined in contrast to multiple ones for standard lazy let-expressions. This is natural since the order of evaluation is important. With multiple variables an extra mechanism for specifying their order of evaluation would have to be introduced. With single variable let-expressions an ordering is imposed easily by nesting of let-expressions.

With the extension of these strict let-expressions the class of expressions to consider is given by the following grammar:

$$\begin{array}{rl} x \in Var & \\ e \in Exp ::= & \lambda\ x.\ e \\ | & e\ x \\ | & x \\ | & let\ \ x_1\ =\ e_1\ \cdots\ x_n =\ e_n\ in\ e \\ | & let!\ x_1\ =\ e_1\ in\ e \end{array}$$

As in Launchbury's semantics we assume that the program under consideration is first translated to a form of lambda terms in which all arguments are variables (expressing sharing explicitly). This is achieved by a normalisation procedure which first performs a renaming (α-conversion) using completely fresh variables ensuring that all bound variables are distinct and then introduces a non-strict let definition for each argument of each application (this let-introduction is defined below as *). The semantics are defined on normalized terms only.

$$\begin{array}{lll} (e\ x)^* & = (e^*)(x^*) & \textbf{if } x \textbf{ is a variable} \\ & = let\ y = (x^*)\ in\ (e^*)\ y\ \textbf{otherwise} & \quad \textit{where}\ \ y\ \textit{is a fresh variable} \\ (\lambda x.e)^* & = \lambda x.(e^*) & \\ (x)^* & = x & \\ (let\ \ x_1\ =\ e_1 \cdots x_n = e_n\ in\ e)^* & = let\ \ x_1\ = (e_1^*) \cdots x_n = (e_n^*)\ in\ (e^*) & \\ (let!\ x_1\ =\ e_1\ in\ e)^* & = let!\ x_1\ = (e_1^*)\ in\ (e^*) & \end{array}$$

2.4 Definition of mixed lazy/strict graph semantics

Before defining mixed semantics we recall the basic rules of Launchbury's natural (operational) semantics: the *Lam*bda, *App*lication, *Var*iable and *Let*-rule.

$$\frac{}{\Gamma : \lambda\ x.e \Downarrow \Gamma : \lambda\ x.e} \quad Lam$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda\ y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z} \quad App$$

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}} \quad Var$$

$$\frac{(\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e \Downarrow \Delta : z} \quad Let$$

We will extend the rules above with a recursive *Str*ictLet rule. This operational *Str*ictLet rule is quite similar to the rule for a normal let, but it adds a condition to enforce the shared evaluation of the expression. We will prove the required properties for the introduced rule.

Definition 1. Operational Mixed Lazy/Strict Graph Semantics.

The let! derivation rule has two requirements. One for the evaluation of e_1 (expressing that it is required to evaluate it on forehand) and one for the standard lazy evaluation of e. Sharing in the evaluation is achieved as follows. First, for the evaluation of e_1 the environment Γ is extended with $x_1 \mapsto e_1$ and in order to achieve shared evaluation x_1 is taken as the term to be evaluated. Then, in the resulting environment Θ the reference to x_1 will still be present but (due to the Var-rule) it will be referring to the evaluated result. This environment is taken as the environment for the shared evaluation of e.

$$\frac{(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1 \qquad \Theta : e \Downarrow \Delta : z}{\Gamma : let!\ x_1\ =\ e_1\ in\ e\ \Downarrow \Delta : z} \, Str$$

It may be apparent that a strict let will behave the same as a normal let when e_1 has a weak head normal form. Otherwise, no derivation will be possible for the strict let. This is for instance the case when the evaluation of e_1 requires recursively the evaluation of x_1.

If we would replace let!'s by standard let's in any expression, the weak head normal form of that expression would not change. However, if we would replace in an expression let's by let!'s, then the weak head normal form of that expression would either stay the same or it would become undefined. Among others these properties will be proven in Sect. 3.

Of course, not only the operational semantics have to be extended but we also have to extend the definition of the lazy denotational meaning function with rules for the meaning of the new let! construct.

As in [6] we have a mathematical function domain, containing at least a lifted version of its own function space, ordered in the standard way with least element $\perp$ following Abramsky and Ong [1], [2], and we use Fn and $\downarrow_{Fn}$ as lifting and projection functions.

An *environment* ρ is a function from variables to values where the domain of values is the function domain. We use the following ordering on environments expressing that larger environments bind more variables but have the same values on the same variables: $\rho \leq \rho'$ is defined as $\forall x.[\rho(x) \neq \perp \Rightarrow \rho(x) = \rho'(x)]$. The *initial environment*, indicated by ρ_0, is the function that maps all variables to $\perp$. We use a special semantic function on environments $\{\!\{\ \}\!\}$. It resolves the possible recursion and is defined as: $\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\rho = \mu\rho'.\rho \sqcup (x_1 \mapsto [\![e_1]\!]_{\rho'} \cdots x_n \mapsto [\![e_n]\!]_{\rho'})$ where μ stands for the least fixed point operator and $\sqcup$ denotes the least upper bound of two environments. It is important to note that for this definition to make sense the environment must be *consistent* with the heap (i.e. if they bind the same variable then there must exist an upper bound on the values to which each binds each such variable).

Definition 2. Denotational Mixed Lazy/Strict Graph Semantics.

$$
\begin{array}{ll}
[\![\lambda x.e]\!]_\rho & = Fn\ (\lambda v.[\![e]\!]_{\rho \sqcup (x \mapsto v)}) \\
[\![e\ x]\!]_\rho & = ([\![e]\!]_\rho) \downarrow_{Fn} ([\![x]\!]_\rho) \\
[\![x]\!]_\rho & = \rho(x) \\
[\![let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e]\!]_\rho & = [\![e]\!]_{\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\rho} \\
[\![let!\ x_1\ =\ e_1\ in\ e]\!]_\rho & = \perp\ ,\ \text{if } [\![x_1]\!]_{\{\!\{x_1 \mapsto e_1\}\!\}\rho} = \perp \\
 & = [\![e]\!]_{\{\!\{x_1 \mapsto e_1\}\!\}\rho}
\end{array}
$$

In extension to [6] we defined above a meaning for *let*!-expressions. This meaning is given by a case distinction. If the meaning of the expression to be shared is $\perp$, then the meaning of the *let*!-expression as a whole becomes $\perp$. Otherwise, the meaning is simply the same as the meaning of the corresponding normal *let*-expression. As with the operational rules, we distinguish between the recursive case (requiring the use of $\{\!\{\}\!\}$) and the non-recursive case.

2.5 Correctness and Computational Adequacy

In this section we will show that each of the theorems stated for natural lazy semantics in [6] also holds for mixed lazy/strict semantics. The first theorem deals with proper use of names.

Theorem 1 (Distinct Names). If $\Gamma : e \Downarrow \Delta : z$ and $\Gamma : e$ is *distinctly named* (i.e. every binding occurring in Γ and in e binds a distinct variable which is also distinct from any free variables of $\Gamma : e$), then every heap/term pair occurring in the proof of the reduction is also distinctly named.

Proof. The cases of the StrictLet rules are trivial since no renaming takes place there. The proof for the other cases is exactly the same as in [6].

Theorem 2 essentially states that reductions preserve meaning on terms and that they possibly only change the meaning of heaps by adding new bindings.

Theorem 2 (Correctness).

$$\Gamma : e \Downarrow \Delta : z \Rightarrow \forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho \wedge\ [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$$

Proof. Induction on the structure of the derivation for $\Gamma : e \Downarrow \Delta : z$. There are five cases:

CASE 1-4. *(Lambda),* Application, Variable *and* Let The proofs for these cases are essentially the same as the ones in [6]. They are not listed here. For interested readers the full proof is given in Appendix A.1.

CASE 5. *(StrictLet)* $\dfrac{(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1 \qquad \Theta : e \Downarrow \Delta : z}{\Gamma : \mathit{let!}\ x_1\ =\ e_1\ \mathit{in}\ e\ \Downarrow \Delta : z}\mathit{Str}$

Assume by induction: [IH1]: $\forall\rho.\ \{\!\{\Gamma, x_1 \mapsto e_1\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ and
[IH2]: $\forall\rho.\ [\![x_1]\!]_{\{\!\{\Gamma, x_1\mapsto e_1\}\!\}\rho} = [\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho}$ and
[IH3]: $\forall\rho.\ \{\!\{\Theta\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and
[IH4]: $\forall\rho.\ [\![e]\!]_{\{\!\{\Theta\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

To prove:
[1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and [2]: $\forall\rho.\ [\![\mathit{let!}\ x_1\ =\ e_1\ \mathit{in}\ e\ ]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

Proof (StrictLet).
[1]: $\{\!\{\Gamma\}\!\}_\rho$
$\leq \{\!\{\Gamma, x_1 \mapsto e_1\}\!\}_\rho$ (variable x_1 not bound in Γ)
$\leq \{\!\{\Theta\}\!\}\rho$ [IH1]
$\leq \{\!\{\Delta\}\!\}\rho$ [IH3]
[2]: $[\![\mathit{let!}\ x_1 = e_1\ \mathit{in}\ e]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= [\![e]\!]_{\{\!\{x_1\mapsto e_1\}\!\}(\{\!\{\Gamma\}\!\}\rho)}$
since $[\![x_1]\!]_{\{\!\{x_1\mapsto e_1\}\!\}(\{\!\{\Gamma\}\!\}\rho)} = [\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho}$ (due to [IH2]) and $[\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho} \neq \bot$
(z_1 is a value: denotationally a lifted function which cannot be $\bot$)
$= [\![e]\!]_{\{\!\{(\Gamma, x_1\mapsto e_1)\}\!\}\rho}$ (variable x_1 not bound in Γ)
$= [\![e]\!]_{\{\!\{\Theta\}\!\}\rho}$ [IH1] (and the definition of $\leq$)
$= [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$ [IH4]

The Computational Adequacy theorem below states that a term with a heap has a valid reduction if and only if they have a non-bottom denotational meaning starting with the initial environment ρ_0.

Theorem 3 (Computational Adequacy).

$$[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot \Leftrightarrow (\exists\Delta, z\ .\ \Gamma : e \Downarrow \Delta : z)$$

Proof. The proof of [6] requires just a single adaption:

Launchbury's proof uses an alternative *resourced* denotational semantics $\mathcal{N}[\![]\!]$ by adding an extra resource argument to the meaning function that works as a counter since it is decremented with every application of the meaning function. In this resourced semantics an environment does not bind a variable to a value directly but here an environment is a function that takes a variable and produces a function that given a resource produces a value. Such environments will be denoted with ϕ or ψ. Again we define a semantic environment function resolving recursion as follows $\mathcal{N}\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\phi = \mu\psi.\phi \sqcup (x_1 \mapsto \mathcal{N}[\![e_1]\!]_\psi \cdots x_n \mapsto \mathcal{N}[\![e_n]\!]_\psi)$.

This resourced semantics has to be extended in order to incorporate the rule for the strict let expression. The extended definition of a resourced mixed semantics $\mathcal{N}[\![]\!]$ is given below.

Definition 3. Resourced Denotational Mixed Semantics.

$$\begin{array}{ll}
\mathcal{N}[\![e]\!]_\phi\ \bot & = \bot \\
\mathcal{N}[\![\lambda x.e]\!]_\phi\ (S\ k) & = Fn\ (\lambda\psi.\mathcal{N}[\![e]\!]_{\phi \sqcup (x \mapsto \psi)}\ k) \\
\mathcal{N}[\![e\ x]\!]_\phi\ (S\ k) & = (\mathcal{N}[\![e]\!]_\phi\ k) \downarrow_{Fn} (\mathcal{N}[\![x]\!]_\phi\ k) \\
\mathcal{N}[\![x]\!]_\phi\ (S\ k) & = \phi\ x\ k \\
\mathcal{N}[\![let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e]\!]_\phi\ (S\ k) & = \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\phi}\ k \\
\mathcal{N}[\![let!\ x_1\ =\ e_1\ in\ e]\!]_\phi\ (S\ k) & = \bot\ ,\ \text{if } \mathcal{N}[\![x_1]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1\}\!\}\phi}\ k = \bot \\
 & = \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1\}\!\}\phi}\ k
\end{array}$$

With these extended resourced semantics the proof proceeds as in [6]. The full proof is given in Appendix A.2.

3 Relation to lazy semantics

In this section we will prove some "folklore" knowledge of programmers that use explicit strictness in a lazy functional programming language.

A *expressions that are bottom lazily, will also be bottom when we make something strict*;

B *when strictness is added to an expression that is non-bottom lazily, either the result stays the same or it becomes bottom*;

C *expressions that are non-bottom using strictness will (after* !*-removal) also be non-bottom lazily with the same result.*

This "folklore" ABC of using strictness must be first turned into formal statements. The concept of result will be formalised by operational meaning. !*-removal* for expressions and environments is formalised below.

Definition 4. Removal of !'s for expressions. *The function* $^{-!}$ *is defined on expressions such that* $e^{-!}$ *is the expression* e *in which every let!-expression is replaced by the corresponding let-expression:*

$$\begin{array}{ll}
(x)^{-!} & = x \\
(\lambda x.e)^{-!} & = \lambda x.(e^{-!}) \\
(e\ x)^{-!} & = (e^{-!})(x^{-!}) \\
(let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e)^{-!} & \\
 & = let\ x_1\ = e_1^{-!} \cdots x_n = e_n^{-!}\ in\ e^{-!} \\
(let!\ x_1\ =\ e_1\ in\ e)^{-!} & = let\ x_1\ = e_1^{-!}\ in\ e^{-!}
\end{array}$$

Definition 5. Removal of !'s for environments. *The function* $^{-!}$ *is defined on environments such that* $\Gamma^{-!}$ *is the environment* Γ *in which in every binding every expression* e *is replaced by the corresponding expression* $e^{-!}$*:*

$$\begin{array}{ll} (\Gamma, x \mapsto e)^{-!} & = (\Gamma^{-!}, x \mapsto e^{-!}) \\ \{\ \}^{-!} & = \{\ \} \end{array}$$

Note that the empty environment is not indicated by the standard symbol for an empty set, $\emptyset$, but instead by $\{\ \}$ as in [6].

We can now formalise the "folklore" ABC. The standard lazy denotational and operational meanings of [6] is indicated by $[\![]\!]^{lazy}$ and $\Downarrow^{lazy}$.

Theorem 4 (Formal Folklore ABC).

A : $[\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = \bot$
$\Rightarrow [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \bot$
B : $[\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} \neq \bot$
$\Rightarrow ([\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \bot \vee \exists z, \Delta, \Theta[\Gamma : e \Downarrow \Delta : z \wedge \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}]$
C : $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot$
$\Rightarrow ([\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} \neq \bot \wedge \exists z, \Delta, \Theta[\Gamma : e \Downarrow \Delta : z \ \wedge \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}]$

Proof. The proofs are straightforward combining computational adequacy (Theorem 3) and the three Theorems 5, 6 and 7 below.

Theorem 5 (Meaning of !-removal).

$$[\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = \bot \Rightarrow [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \bot$$

Proof. Follows directly from the definition of the meaning function $[\![\ ]\!]$.

Theorem 6 (Compare with Lazy Reduction).

$$\Gamma : e \Downarrow \Delta : z \ \Rightarrow \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!} \wedge [\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}} = [\![z^{-!}]\!]^{lazy}_{\{\!\{\Delta^{-!}\}\!\}}$$

Proof. Assume we have $\Gamma : e \Downarrow \Delta : z$ with derivation tree σ. First, apply the transformation $^{-!}$ to the leaves of σ which is valid since it is clear that for each leaf, which is always of the form $\Gamma : \lambda\ x.e \Downarrow \Gamma : \lambda\ x.e$, also $\Gamma^{-!} : \lambda\ x.e^{-!} \Downarrow^{lazy} \Gamma^{-!} : \lambda\ x.e^{-!}$ will be valid. Then, replace in σ each occurrence of a *StrictLet*-rule by a *Let*-rule, leaving out the subtree corresponding to the precondition $(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1$. This results in a new derivation tree τ which is a valid derivation tree for $\Downarrow^{lazy}$. However, this derivation tree contains the new environment Θ in which some extra non-lazy evaluations are stored. Since the meaning of $e^{-!}$ is not $\bot$, it holds that $[\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}} = [\![z^{-!}]\!]^{lazy}_{\{\!\{\Gamma, x_1 \mapsto e_1\}\!\}}$ and hence $[\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}} = [\![z^{-!}]\!]^{lazy}_{\{\!\{\Delta^{-!}\}\!\}}$. Inductively, this gives the required result since apart from !-appearance in the leaves of the tree, the only sources of !-introduction in e are the *StrictLet*-rules.

Theorem 7 (Reduction and !-removal).

$$\Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Delta : z^{-!} \Rightarrow (\Gamma : e \Downarrow \Theta : z \wedge [\![z]\!]_{\{\!\{\Theta\}\!\}} = [\![z]\!]_{\{\!\{\Delta\}\!\}}) \vee [\![e]\!]_\rho = \bot$$

Proof. Suppose we have $\Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Delta : z^{-!}$. The derivation tree σ can be used to construct a derivation tree τ for $\Gamma : e \Downarrow \Theta : z$ by adding where appropriate !'s, *StrictLet*-rules and conditions with Θ environmnets as in the proof of Theorem 6. The only way to complete this tree is to prove these conditions. So, when all conditions can be proved $\Gamma : e \Downarrow \Theta : z$ holds (and the meanings for the environments are the same), otherwise there is simply no valid derivation tree and hence $[\![e]\!]_\rho = \bot$ according to Theorem 3.

4 Example proofs with mixed semantics

With a small example we will show how proofs can be made using mixed semantics; the proofs show formally that with mixed semantics it is possible to distinguish operationally between terms that were indistinguishable lazily.

Lazy semantics [6] makes it possible to yield $\lambda x.\Omega$ (Ω is defined below) and Ω as different results. However, in lazy semantics it is not possible to define a function f that *produces a different observational result* depending on which one is given as an argument. We say that two terms "produce a different observational result" if at least one term produces a basic value and the other one either produces a different basic value or $\bot$. This means that in lazy natural semantics $\lambda x.\Omega$ and Ω belong to a single equivalence class of which the members cannot be distinguished observationally by the programmer.

With mixed semantics a definition for such a distinguishing function f is given below. The result of f on $\lambda x.\Omega$ will be 42 and the result of f on Ω will be $\bot$. Note that it is *not* possible to return anything else than $\bot$ in the Ω case.

$$\begin{aligned} \Omega &\equiv (\lambda x.xx)(\lambda x.xx) \\ f &\equiv \lambda x.(let!\ y = x\ in\ 42) \end{aligned}$$

We will prove two properties:

$$\nexists \Delta, z.\ \{\} : f\ \Omega\ \Downarrow \Delta : z \tag{1}$$

$$\exists \Delta.\ \{\} : f\ (\lambda x.\Omega)\ \Downarrow \Delta : 42 \tag{2}$$

For the first property we have to prove that it is impossible to construct a finite derivation according to the operational semantics. Applying Theorem 3, the computational adequacy theorem, it is sufficient to show that the denotational meaning of $f\ \Omega$ is undefined. The proof is given below:

$[\![f\ \Omega]\!]_\rho$
$= [\![(\lambda x.let!\ y\ =\ x\ in\ 42)(\Omega)]\!]_\rho$
$= ([\![\lambda x.let!\ y\ =\ x\ in\ 42]\!]_\rho) \downarrow_{Fn} ([\![\Omega]\!]_\rho)$
$= (Fn\ (\lambda v.[\![let!\ y\ =\ x\ in\ 42]\!]_{\rho \sqcup (x \mapsto v)})) \downarrow_{Fn} ([\![\Omega]\!]_\rho)$
$= (\lambda v.[\![let!\ y\ =\ x\ in\ 42]\!]_{\rho \sqcup (x \mapsto v)})[\![\Omega]\!]_\rho$

$= [\![let!\ y\ =\ x\ in\ 42]\!]_{\rho \sqcup (x \mapsto [\![\Omega]\!]_\rho)}$
$= \bot$ since $[\![y]\!]_{\rho \sqcup (x \mapsto [\![\Omega]\!]_\rho) \sqcup (y \mapsto x)} = (\rho \sqcup (x \mapsto [\![\Omega]\!]_\rho) \sqcup (y \mapsto x))(y) = [\![\Omega]\!]_\rho = \bot$
The proof of the second property is written down with sub-derivations contained within square brackets and stating the name of the used derivation rule. To work with numerals we assume in the operational semantics the availability of a standard reduction rule (*Num*) that states that each numeral reduces to itself.

$$\left[\begin{array}{l}
\{\ \} : f\ (\lambda x.\Omega) \\
\{\ \} : (\lambda x.\ let!\ y = x\ in\ 42)\ (\lambda x.\Omega) \\
\left[\begin{array}{l} \{\ \} : (\lambda x.\ let!\ y = x\ in\ 42) \\ \{\ \} : (\lambda x.\ let!\ y = x\ in\ 42) \\ {}_{Lam} \end{array}\right. \\
\left[\begin{array}{l} \{\ \} : (let!\ y = x\ in\ 42)\ [\lambda x.\Omega/x] \\ \{\ \} : let!\ y = \lambda x.\Omega\ in\ 42 \\ \left[\begin{array}{l} \{y \mapsto \lambda x.\Omega\} : y \\ \left[\begin{array}{l} \{\ \} : \lambda x.\Omega \\ \{\ \} : \lambda x.\Omega \\ {}_{Lam} \end{array}\right. \\ \{y \mapsto \lambda x.\Omega\} : \lambda x.\Omega \\ {}_{Var} \end{array}\right. \\ \left[\begin{array}{l} \{y \mapsto \lambda x.\Omega\} : 42 \\ \{y \mapsto \lambda x.\Omega\} : 42 \\ {}_{Num} \end{array}\right. \\ \{y \mapsto \lambda x.\Omega\} : 42 \\ {}_{let!} \end{array}\right. \\
\{y \mapsto \lambda x.\Omega\} : 42 \\
{}_{App}
\end{array}\right.$$

5 Related and Future work

A formal semantics that is similar to Launchbury's, has been defined independently by Barendsen and Smetsers [3]. They address strictness analysis but they do not address explicitly enforced strictness. To transfer results it might be worthwhile to establish a formal correspondence between these two semantics.

With the purpose of deriving a lazy abstract machine Sestoft [9] has revised Launchbury's semantics. Launchbury's semantics require global inspection (which is unwanted for an abstract machine) for preserving the Distinct Names property. When an abstract machine is to be derived from our mixed semantics, analogue revisions will be required. As is further pointed out by Sestoft [9] the rules given by Launchbury are not *fully lazy*. Full laziness can be achieved by introducing new let-bindings for every maximal free expression.

Andrew Pitts [7] discusses non-termination issues of logical relations and operational equivalence in the context of the presence of existential types in a strict language. He provides some theory that might also be used to address the problems that arise in a mixed lazy/strict context. That would require a combination of his work and the work of Patricia Johann and Janis Voigtländer [5] who use a denotational approach to present some "free" theorems in the presence of Haskell's seq.

Acknowledgements We would like to thank the anonymous referees of an earlier version of this paper for their helpful reviews.

6 Conclusions

We have extended Launchbury's lazy graph semantics with a construct for explicit strictness. The resulting derivation system is shown to be correct and computationally adequate.

We have explored what happens when strictness is added or removed within mixed lazy/strict graph semantics. Correspondences and differences between lazy and mixed semantics have been established by studying the effects of removal and addition of strictness. Our results formalise the common "folklore" knowledge about the use of explicit strictness in a lazy context.

Mixed lazy/strict graph semantics differs significantly from lazy graph semantics. It is possible to write expressions that with mixed semantics distinguish between particular terms that have different lazy semantics while these terms can not be distinguished by an expression within that lazy semantics. This was shown formally as a small example of the use of mixed semantics in formal proofs.

References

1. S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.
2. S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, (105):159–267, 1993.
3. E. Barendsen and S. Smeters. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. World scientific, 1999.
4. P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, New York, 2000.
5. P. Johann and J. Voigtlaender. Free theorems in the presence of seq. In *Proceedings of the 31st International Conference on Principles of Programming Languages 2004 (POPL'04)*, pages 99–110. IEEE Press, 2004.
6. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
7. A. M. Pitts. Existential types: Logical relations and operational equivalence. In *Proceedings of the 25th International Conference on Automata Languages and Programming, ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.
8. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/∼clean/contents/contents.html.
9. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

A Proofs

This appendix lists some full proofs including those parts that are essentially the same as in [6].

A.1 Full Proof of Correctness Theorem

Theorem 2 Correctness

$$\Gamma : e \Downarrow \Delta : z \Rightarrow \forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho \wedge\ [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$$

Induction on the structure of the derivation for $\Gamma : e \Downarrow \Delta : z$. There are five cases:

CASE 1. *(Lambda)* $\dfrac{}{\Gamma : \lambda\ x.e \Downarrow \Gamma : \lambda\ x.e}\,Lam$

To prove: [1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Gamma\}\!\}\rho$ and [2]: $\forall\rho.\ [\![\lambda x.e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![\lambda x.e]\!]_{\{\!\{\Gamma\}\!\}\rho}$

Proof (Lambda). [1]: Follows directly from the reflexivity of $\leq$. [2]: Trivial.

CASE 2. *(Application)* $\dfrac{\Gamma : e \Downarrow \Delta : \lambda\ y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : z}{\Gamma : e\ x \Downarrow \Theta : z}\,App$

Assume by induction: [IH1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and
[IH2]: $\forall\rho.\ [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![\lambda y.e']\!]_{\{\!\{\Delta\}\!\}\rho}$ and
[IH3]: $\forall\rho.\ \{\!\{\Delta\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ and
[IH4]: $\forall\rho.\ [\![e'[x/y]]\!]_{\{\!\{\Delta\}\!\}\rho} = [\![z]\!]_{\{\!\{\Theta\}\!\}\rho}$

To prove: [1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ and [2]: $\forall\rho.\ [\![e\ x]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Theta\}\!\}\rho}$

Proof (Application).
[1]: Transitivity of $\leq$ applied to [IH1] and [IH3] yields $\{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$.
[2]: $[\![e\ x]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} \downarrow_{Fn} [\![x]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= [\![\lambda y.e']\!]_{\{\!\{\Delta\}\!\}\rho} \downarrow_{Fn} [\![x]\!]_{\{\!\{\Gamma\}\!\}\rho}$ [IH2]
$= (Fn\ (\lambda v.[\![e']\!]_{\{\!\{\Delta\}\!\}\rho \sqcup (y \mapsto v)})) \downarrow_{Fn} [\![x]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= (\lambda v.[\![e']\!]_{\{\!\{\Delta\}\!\}\rho \sqcup (y \mapsto v)})[\![x]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= [\![e']\!]_{\{\!\{\Delta\}\!\}\rho \sqcup (y \mapsto [\![x]\!]_{\{\!\{\Gamma\}\!\}\rho})}$
$= [\![e'[x/y]]\!]_{\{\!\{\Delta\}\!\}\rho}$ (standard λ-calculus substitution lemma)
$= [\![z]\!]_{\{\!\{\Theta\}\!\}\rho}$ [IH4]

CASE 3. *(Variable)* $\dfrac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}}\,Var$

Assume by induction:
[IH1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and [IH2]: $\forall\rho.\ [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$
To prove:
[1]: $\forall\rho.\ \{\!\{\Gamma, x \mapsto e\}\!\}\rho \leq \{\!\{\Delta, x \mapsto z\}\!\}\rho$ and [2]: $\forall\rho.\ [\![x]\!]_{\{\!\{\Gamma, x\mapsto e\}\!\}\rho} = [\![\hat{z}]\!]_{\{\!\{\Delta, x\mapsto z\}\!\}\rho}$

Proof (Variable).

[1]: $\{\!\{\Gamma, x \mapsto e\}\!\}\rho$

$= \mu\rho'.\rho \sqcup \{\!\{\Gamma\}\!\}\rho' \sqcup (x \mapsto [\![e]\!]_{\rho'})$

$= \mu\rho'.\rho \sqcup \{\!\{\Gamma\}\!\}\rho' \sqcup (x \mapsto [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho'})$

$= \mu\rho'.\rho \sqcup \{\!\{\Gamma\}\!\}\rho' \sqcup (x \mapsto [\![z]\!]_{\{\!\{\Delta\}\!\}\rho'})$ [IH2]

$\leq \mu\rho'.\rho \sqcup \{\!\{\Delta\}\!\}\rho' \sqcup (x \mapsto [\![z]\!]_{\{\!\{\Delta\}\!\}\rho'})$ [IH1]

$\leq \mu\rho'.\rho \sqcup \{\!\{\Delta\}\!\}\rho' \sqcup (x \mapsto [\![z]\!]_{\rho'})$

$= \{\!\{\Delta, x \mapsto z\}\!\}\rho$

[2]: $[\![x]\!]_{\{\!\{\Gamma, x\mapsto e\}\!\}\rho}$

$= \{\!\{\Gamma, x \mapsto e\}\!\}_{\rho}(x)$

$= \{\!\{\Delta, x \mapsto z\}\!\}_{\rho}(x)$ (definition of $\leq$)

$= [\![z]\!]_{\{\!\{\Delta, x\mapsto z\}\!\}\rho}$ (definition of the meaning function $[\![\ ]\!]$)

$= [\![\hat{z}]\!]_{\{\!\{\Delta, x\mapsto z\}\!\}\rho}$ (α-conversion of z)

CASE 4. *(Let)* $\dfrac{(\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e \Downarrow \Delta : z}\,Let$

Assume by induction: [IH1]: $\forall\rho.\ \{\!\{(\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n)\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and
[IH2]: $\forall\rho.\ [\![e]\!]_{\{\!\{(\Gamma, x_1\mapsto e_1\cdots x_n\mapsto e_n)\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

To prove:

[1]:$\forall\rho.\{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and [2]:$\forall\rho.[\![let\ x_1 = e_1 \cdots x_n = e_n\ in\ e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

Proof (Let).

[1]: $\{\!\{\Gamma\}\!\}_{\rho}$

$\leq \{\!\{\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\rho$ (variables $x_1 \cdots x_n$ not bound in Γ)

$\leq \{\!\{\Delta\}\!\}\rho$ [IH1]

[2]: $[\![let\ x_1 = e_1 \cdots x_n = e_n\ in\ e]\!]_{\{\!\{\Gamma\}\!\}\rho}$

$= [\![e]\!]_{\{\!\{x_1\mapsto e_1\cdots x_n\mapsto e_n\}\!\}(\{\!\{\Gamma\}\!\}\rho)}$

$= [\![e]\!]_{\mu\rho'.\{\!\{\Gamma\}\!\}\rho\sqcup(x_1\mapsto[\![e_1]\!]_{\rho'}\ \cdots\ x_n\mapsto[\![e_n]\!]_{\rho'})}$

$= [\![e]\!]_{\{\!\{\Gamma, x_1\mapsto e_1\cdots x_n\mapsto e_n\}\!\}\rho}$ (variables $x_1 \cdots x_n$ not bound in Γ)

$= [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$ [IH2]

CASE 5. *(StrictLet)* $\dfrac{(\Gamma, x_1 \mapsto e_1) : x_1 \Downarrow \Theta : z_1 \qquad \Theta : e \Downarrow \Delta : z}{\Gamma : let!\ x_1\ =\ e_1\ in\ e\ \Downarrow \Delta : z}\,Str$

Assume by induction: [IH1]: $\forall\rho.\ \{\!\{\Gamma, x_1 \mapsto e_1\}\!\}\rho \leq \{\!\{\Theta\}\!\}\rho$ and
[IH2]: $\forall\rho.\ [\![x_1]\!]_{\{\!\{\Gamma, x_1\mapsto e_1\}\!\}\rho} = [\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho}$ and
[IH3]: $\forall\rho.\ \{\!\{\Theta\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and
[IH4]: $\forall\rho.\ [\![e]\!]_{\{\!\{\Theta\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

To prove:

[1]: $\forall\rho.\ \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$ and [2]: $\forall\rho.\ [\![let!\ x_1\ =\ e_1\ in\ e\ ]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$

Proof (StrictLet).

[1]: $\{\!\{\Gamma\}\!\}_{\rho}$

$\leq \{\!\{\Gamma, x_1 \mapsto e_1\}\!\}_{\rho}$ (variable x_1 not bound in Γ)

$\leq \{\!\{\Theta\}\!\}\rho$ [IH1]
$\leq \{\!\{\Delta\}\!\}\rho$ [IH3]

[2]: $[\![let!\ x_1 = e_1\ in\ e]\!]_{\{\!\{\Gamma\}\!\}\rho}$
$= [\![e]\!]_{\{\!\{x_1 \mapsto e_1\}\!\}(\{\!\{\Gamma\}\!\}\rho)}$
since $[\![x_1]\!]_{\{\!\{x_1 \mapsto e_1\}\!\}(\{\!\{\Gamma\}\!\}\rho)} = [\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho}$ (due to [IH2]) and $[\![z_1]\!]_{\{\!\{\Theta\}\!\}\rho} \neq \bot$ (z_1 is a value: denotationally a lifted function which cannot be $\bot$)
$= [\![e]\!]_{\{\!\{(\Gamma, x_1 \mapsto e_1)\}\!\}\rho}$ (variable x_1 not bound in Γ)
$= [\![e]\!]_{\{\!\{\Theta\}\!\}\rho}$ [IH1] (and the definition of $\leq$)
$= [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$ [IH4]

A.2 Full Proof of Computational Adequacy Theorem

Theorem 3 Computational Adequacy

$$[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot \Leftrightarrow (\exists \Delta, z\ .\ \Gamma : e \Downarrow \Delta : z)$$

Proof. The two implications are proven separately.
$\Leftarrow$: Follows immediately from Theorem 2. The meaning of the resulting value z is a lifted function which cannot be $\bot$.
$\Rightarrow$: This part requires more effort. The denotational semantics are hard to relate to the operational semantics. So, we define alternative versions of the denotational and operational semantics. These alternative versions are equivalent to the original ones but they are more closely related to each other.

First, we give an alternative *resourced* denotational semantics $\mathcal{N}[\![]\!]$ by adding an extra resource argument to the meaning function that works as a counter since it is decremented with every application of the meaning function. This counter is taken from the countable chain domain C (defined by the domain equation $C = C_\bot$). On C we defined the injection function S such that the elements of C can be given by the sequence $\bot$, $S\bot$, $S(S\bot)$, ... with limit element $\omega \equiv S(S(S \cdots))$. In this resourced semantics an environment does not bind a variable to a value directly but here an environment is a function that takes a variable and produces a function that given a resource produces a value. Such environments will be denoted with ϕ or ψ. Again we define a semantic environment function resolving recursion as follows $_{\mathcal{N}\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\phi} = \mu\psi.\phi \sqcup (x_1 \mapsto \mathcal{N}[\![e_1]\!]_\psi \cdots x_n \mapsto \mathcal{N}[\![e_n]\!]_\psi)$. Using these notations and auxiliary definitions $\mathcal{N}[\![]\!]$ is defined as follows.

Definition 6. Resourced Denotational Mixed Semantics.

$$\begin{array}{ll}
\mathcal{N}[\![e]\!]_\phi\ \bot & = \bot \\
\mathcal{N}[\![\lambda x.e]\!]_\phi\ (S\ k) & = Fn\ (\lambda\psi.\mathcal{N}[\![e]\!]_{\phi \sqcup (x \mapsto \psi)}\ k) \\
\mathcal{N}[\![e\ x]\!]_\phi\ (S\ k) & = (\mathcal{N}[\![e]\!]_\phi\ k) \downarrow_{Fn} (\mathcal{N}[\![x]\!]_\phi\ k) \\
\mathcal{N}[\![x]\!]_\phi\ (S\ k) & = \phi\ x\ k \\
\mathcal{N}[\![let\ x_1\ = e_1 \cdots x_n = e_n\ in\ e]\!]_\phi\ (S\ k) & = \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\phi}\ k \\
\mathcal{N}[\![let!\ x_1\ =\ e_1\ in\ e]\!]_\phi\ (S\ k) & = \bot\ ,\ \text{if } \mathcal{N}[\![x_1]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1\}\!\}\phi}\ k = \bot \\
 & = \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1\}\!\}\phi}\ k
\end{array}$$

Clearly, given infinite resources $\mathcal{N}[\![]\!]$ equals $[\![]\!]$, i.e. if $\forall x.\rho\ x = \phi\ x\ \omega$ then $[\![e]\!]_\rho = \mathcal{N}[\![e]\!]_\phi\ \omega$. Furthermore, the following lemma holds (using for m applications of S the notation S^m):

Lemma 1. $\forall x.\rho\ x = \phi\ x\ \omega \wedge [\![e]\!]_\rho \neq \bot \Rightarrow \exists m.\mathcal{N}[\![e]\!]_\phi\ (S^m\bot) \neq \bot$

Proof. $\mathcal{N}[\![]\!]$ is a continuous function since it is defined using continuous functions only. So, it holds that if $[\![]\!]$ produces non-bottom for some term, so does some finite approximation.

Second, an alternative operational semantics $\Downarrow_{\mathcal{N}}$ is obtained by replacing the *App*lication and *Var*iable rules given in Sect. 2.4 by alternative ones. Below we state only the changed rules, the rest of the system (including the lazy and strict let-rules) remains the same.

$$\frac{\Gamma : e \Downarrow_{\mathcal{N}} \Delta : \lambda\ y.e' \qquad (\Delta, y \mapsto x) : e' \Downarrow_{\mathcal{N}} \Theta : z}{\Gamma : e\ x \Downarrow_{\mathcal{N}} \Theta : z}\ App$$

$$\frac{(\Gamma, x \mapsto e) : \hat{e} \Downarrow_{\mathcal{N}} \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow_{\mathcal{N}} \Delta : z}\ Var$$

The effect of these new rules is that they mimic the operations on environments in the alternative denotational semantics more closely. The new application rule adds an indirection for each lambda reduction, increasing the number of closures instead of performing a substitution directly. Furthermore, updating is removed by the new variable rule.

Lemma 2. $\Gamma : e \Downarrow \Delta : z \Leftrightarrow \Gamma : e \Downarrow_{\mathcal{N}} \Theta : z' \wedge \exists n.subst^n(\Theta : z') = \Delta : z$

Proof. *subst* removes the introduced closures and is defined as $subst((\Gamma, x \mapsto e) : z) = \Gamma : z\ [e/x]$. The proof follows by induction on the derivation tree.

The next lemma relates the alternative denotational and operational semantics to each other.

Lemma 3.
$\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\phi}(S^m\bot) \neq \bot \Rightarrow \exists \Delta, z.(x_1 \mapsto e_1 \cdots x_n \mapsto e_n) : e \Downarrow_{\mathcal{N}} \Delta : z$

Proof. By induction on m.

Now, we can finish our proof of the $\Rightarrow$ part of Theorem 3. Assume that $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot$. Then, by Lemma 1 there exists an m such that the corresponding approximating alternative denotational semantics is non-bottom. By Lemma 3 there exists a corresponding derivation according to the alternative operational semantics. Finally, by Lemma 2 there is a corresponding derivation conform the original operational semantics.