PROVING LAZY FOLKLORE WITH MIXED LAZY/STRICT SEMANTICS

MARKO VAN EEKELEN AND MAARTEN DE MOL

Institute for Computing and Information Sciences, Radboud University Nijmegen, NL
e-mail address: {marko, maartenm}@cs.ru.nl

Abstract. Explicit enforcement of strictness is used by functional programmers for many different purposes. Few functional programmers, however, are aware that explicitly enforcing strictness has serious consequences for (formal) reasoning about their programs. Some vague "folklore" knowledge has emerged concerning the correspondence between lazy and strict evaluation but this is based on experience rather than on rigid proof.

This paper employs a model for formal reasoning with enforced strictness based on John Launchbury's lazy graph semantics. In this model Launchbury's semantics are extended with an explicit strict let construct. Examples are given of the use of these semantics in formal proofs. We formally prove some "folklore" properties that are often used in informal reasoning by programmers.

This paper is written at the occasion of the celebration of the 60^{th} *anniversary of Henk Barendregt. Henk was the supervisor for the Ph.D. Thesis of Marko van Eekelen. This thesis was just one of the many results of the Dutch Parallel Reduction Machine project in which Henk played a central role.*

Quite some time ago, he brought the authors of this paper together knowing that they had common interests in formal proofs for functional programs. This lead to a Master Thesis, the Sparkle dedicated proof assistant for the language CLEAN, a pile of papers and a Ph.D. manuscript in preparation. Henk taught us how to perform research on a fundamental level without loosing sight of the applications of your work.

We are very grateful to him for enlightening us.

1. Introduction and motivation

Strictness is a mathematical property of a function. A function f is *strict* in its argument if its result is undefined when its argument is undefined, in other words: if $f\bot = \bot$, where $\bot$ is the symbol representing the undefined value.

Strictness analysis is used to derive strictness properties for given function definitions in programs written in a functional programming language. If the results of such an analysis are indicated in the program via strictness annotations then of course these annotations do not change the semantics at all. Therefore, it is often recommended to use strictness annotations only when strictness holds mathematically. These annotations are then meant to be used by the compiler for optimisation purposes only.

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

REFLECTIONS ON TYPE THEORY, λ-CALCULUS, AND THE MIND | Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday

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 call[1] 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: Clean and in data structures: Clean, Haskell),
- special data structures (unboxed arrays: Clean, Haskell),
- special primitives (seq: Haskell),
- special language constructs (let!, `#!`: Clean),
- special tools (strictness analyzers: Clean, Haskell).

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. Hence, it is not an option but it is an *obligation* for the compiler to generate code that takes these annotations into account. For reasoning about these annotated programs, however, one tends 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. So, strictness indicated via annotations must form a essential part of the semantics. This may have surprising consequences.

Example of semantic changes due to strictness annotations::
Consider for instance the following Clean definition of the function `f`, which by means of the `!`-annotation in the type is made explicitly strict in its first argument. In Haskell a similar effect can be obtained using an application of `seq`.

```
f :: !Int -> Int
f x = 5
```

Without the strictness annotation, the property $\forall_x[\texttt{f}\ x = 5]$ would hold unconditionally by definition. Now consider the effects of the strictness annotation in the type which makes the function `f` strict in its argument. Clearly, the proposition $\texttt{f}\ 3 = 5$ still holds. However, $\texttt{f undef} = 5$ does not hold, because `f undef` does not terminate due to the enforced evaluation of `undef`. Therefore, $\forall_x[\texttt{f}\ x = 5]$ does not hold unconditionally. The property can be fixed by adding a definedness condition using the special symbol $\perp$, denoting undefined. This results in $\forall_x[x \neq \perp\ \rightarrow \texttt{f}\ x = 5]$, which *does* hold for the annotated function `f`.

The example above illustrates that the definition of `f` cannot *unconditionally* be substituted in all its occurrences. It is only allowed to substitute `f` when it is **known** that its argument `x` is **not undefined**. This has a fundamental impact on the semantics of function application.

[1](An *external* call is a call to a function which is defined in a different (possibly imperative) programming language, e.g. C.

The addition of an exclamation mark by a programmer clearly has an effect on the logical properties of functions. The change of a logical property due to addition or removal of strictness can cause problems for program changes made by a programmer. If a programmer is unaware of the logical consequences, this can lead to errors not only at development time but also in the later stage of maintaining the program. A programmer will reason formally or informally about the program and make changes that are consistent with the perceived logical properties.

Changes in logical properties are not only important for the programmer but also for those who work on the compiler. Of course, it is obvious that code has to be generated to accommodate the strictness. Less obvious however, is the consequences adding strictness may have on the correctness of program transformations. There can be far-reaching consequences on various kinds of program transformations.

In other words: the addition or removal of strictness to programs may cause previously valid logical properties to be broken. From a proving point of view this is a real problem: suppose one has successfully proved a difficult property by means of a sequence of lemmata, then the invalidation of even a single lemma may cause a ripple effect throughout the entire proof! The adaptation to such a ripple effect is both cumbersome and resource-intensive.

Unfortunately, the invalidation of logical properties due to changed strictness annotations is quite common. This invalidation can usually be fixed by the addition of a condition for the strict case (see the example below).

Example of the addition of a condition::
$\forall_{f,g}\forall_{xs}[\texttt{map}\ (f \circ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)]$

Affected by strictness::
This property is valid for lazy lists, but invalid for element-strict lists.
Note that no assumptions can be made about the possible strictness of f or g. Instead, the property must hold for all possible functions f and g.

Invalid in the strict case because::
Suppose $xs = [12]$, $g\ 12 = \bot$ and $f\ (g\ 12) = 7$.
Then $\texttt{map}\ (f \circ g)\ xs = [7]$, both in the lazy and in the strict case.
However, $\texttt{map}\ f\ (\texttt{map}\ g\ xs) = [7]$ in the lazy case, but $\bot$ in the strict case.

Extra definedness condition for the lazy case::
The problematic case can be excluded by demanding that for all elements of the list $g\ x$ can be evaluated successfully.

Reformulated property for the strict case::
$\forall_{f,g,xs}[\forall_{x \in xs}[g\ x \neq \bot] \rightarrow \texttt{map}\ (f \circ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)]$.

However, quite surprisingly, it may also be that the invalidation of logical properties due to changed strictness annotations requires the *removal* of definedness conditions. Below an example is given where the strict case requires the removal of a condition which was required for the lazy case.

Example of the removal of a condition::
$\forall_{xs}[\mathit{finite}\ xs \rightarrow \texttt{reverse}\ (\texttt{reverse}\ xs) = xs]$

Affected by strictness::
This property is valid both for lazy lists and for spine-strict lists. However, the condition *finite* xs is satisfied automatically for spine-strict lists. In the spine-strict

case, the property can therefore safely be reformulated (or, rather, optimized) by removing the *finite* xs condition.

Invalid without finite condition in the lazy case because::
Suppose $xs = [1, 1, 1, \ldots]$.
Then $\texttt{reverse}\ (\texttt{reverse}\ xs) = \bot$, both in the lazy and in the strict case.
However, $xs = \bot$ in the strict case, while it is unequal to $\bot$ in the lazy case.

Reformulated property for the strict case::
$\forall_{xs}[\texttt{reverse}\ (\texttt{reverse}\ xs) = xs]$

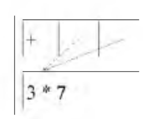
For reasoning with strictness, there is only little theory available so far. In this paper we develop an appropriate mixed denotational and operational semantics for formal reasoning about programs in a mixed lazy/strict context.

2. Mixed lazy/strict graph semantics

Since we consider graphs as an essential part of the semantics of a lazy language ([4, 18], we have chosen to extend Launchbury's graph semantics [14]. 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. Below, we 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 parts to be shared.

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



represents the graph on the right:

Graph reduction is formalized by a system of derivation rules. Graph nodes are represented by variable definitions in an environment. A typical graph reduction proof is given below. A linear notation is used. Below the correspondence is illustrated by showing the linear notation on the left and its equivalent graphical notation on the right.

$$\left[\begin{array}{l} \Gamma : e \\ subderivation_1 \\ \cdots \\ subderivation_n \\ \Delta : z \\ {}_{Let} \end{array}\right. \qquad \frac{subderivation_1 \quad \cdots \quad subderivation_n}{\Gamma : e \ \Downarrow \Delta : z}\ Let$$

Each reduction step corresponds to applying a derivation rule (assuming extra rules for numbers and arithmetic; the standard rules are given in Sect. 2.4). Below we give the derivation corresponding to the sharing example above. We leave out normalization and renaming of variables where this cannot cause confusion.

$$
\begin{array}{l}
\{\ \} : let\ x = 3 * 7\ in\ x + x \\
\quad \{\ x \mapsto 3 * 7\ \} : x + x \\
\quad\quad \{\ x \mapsto 3 * 7\ \} : x \\
\quad\quad\quad \{\ \} : 3 * 7 \\
\quad\quad\quad \{\ \} : 21 \\
\quad\quad\quad {}_{Num, Num, *} \\
\quad\quad \{\ x \mapsto 21\ \} : 21 \\
\quad\quad {}_{Var} \\
\quad\quad \{\ x \mapsto 21\ \} : 21 \\
\quad\quad \{\ x \mapsto 21\ \} : 21 \\
\quad\quad {}_{Var} \\
\quad \{\ x \mapsto 21\ \} : 42 \\
\quad {}_{+} \\
\{\ x \mapsto 21\ \} : 42 \\
{}_{Let}
\end{array}
$$

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 non-recursive strict variant of *let*-expressions.

From a semantic point of view a standard recursive *let*-expression combined with a strict non-recursive *let*-expression gives full expressiveness. Due to the possibility of recursion in the standard *let*, there is no need for adding recursion to the strict *let*. (Consider for example $let\ x\ =\ e\ x\ in\ let!\ y\ =\ x\ in\ e'$.)

So, we have chosen not to allow recursion in the strict *let*, although allowing a recursive strict *let* would not give any semantic problems (as shown in [19]). This corresponds to the semantics of the strictness constructs of Haskell [5, 12, 13] and Clean [6, 15, 16] that do not allow recursion for their strictness constructs.

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:

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

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 normalization 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. The semantics are defined on normalized terms only.

2.4. **Definition of mixed lazy/strict graph semantics.** We extend the basic rules of Launchbury's natural (operational) semantics (the *Lam*bda, *App*lication, *Var*iable and *Let*-rule) 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.

The added *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 by extending the environment Θ resulting form the evaluation of e_1 with $x_1 \mapsto z_1$. This environment is then taken as the environment for the evaluation of e.

A striking difference between a standard *let* and a strict *let* is that the environment is extended before the evaluation for a standard *let* and after the evaluation for a strict *let*. This will by itself never give different results since a strict *let* is non-recursive. A strict *let* will behave the same as a standard *let* when e_1 has a weak head normal form. Otherwise, no derivation will be possible for the strict *let*.

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 non-recursive *let*'s by *let*!'s, then the weak head normal form of that expression would either stay the same or it would become undefined. This is one of the "folklore" properties that is proven in Sect. 3.

Definition 2.1. Operational Mixed Lazy/Strict Graph Semantics.

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

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

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

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

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

Corresponding to the operational semantics given above, we define below the *denotational* meaning function including the *let*! construct. As in [14] we have a lifted function space ordered in the standard way with least element $\perp$ following Abramsky and Ong [1, 2].

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 some domain, containing at least a lifted version of its own function space. We use the following well-defined 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 which is continuous 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).

The denotational meaning function extends [14] with meaning for *let*!-expressions that 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 $\perp$. For the other case, the definition is similar to the meaning of a *let*-expression.

Definition 2.2. Denotational Mixed Lazy/Strict Graph Semantics.

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

2.5. **Correctness and Computational Adequacy.** Using the definitions above, correctness theorems as in [14] have been established (proofs can be found in [19]). The first theorem deals with proper use of names.

Theorem 2.3 (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.

Theorem 2.4 essentially states that reductions preserve meaning on terms and that they possibly only change the meaning of heaps by adding new bindings.

Theorem 2.4 (Correctness).

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

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 2.5 (Computational Adequacy).

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

3. Relation to lazy semantics

Consider the following "folklore" knowledge statements of programmers:

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.*

We will turn this "folklore" ABC of using strictness into formal statements. The phrase "*is bottom lazily*" is taken to mean that when lazy semantics is used the meaning of the expression is $\perp$. The phrase "*result*" indicates of course a partial result: this can be formalized with our operational meaning.

Theorem 3.5 will constitute the formal equivalents of these "folklore" statements. In order to formulate that theorem we first need formally define a few operations. For completeness we give below the full definition of the trivial operation of !*-removal*.

Definition 3.1. Removal of strictness within 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 3.2. Removal of strictness within 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 $e^{-!}$:

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

We followed here [14] indicating the empty environment by $\{\ \}$ instead of by $\emptyset$.

The analogue of !-removal is of course !-addition. We model addition of !'s to an expression e by creating a set of all those expressions that will be the same as e after !-removal. In this way we cover all possible ways of adding a !.

Definition 3.3. Addition of strictness to expressions and environments. The function *AddStrict* is defined on expressions and environments such that $\mathit{AddStrict}(e)$, resepectively $\mathit{AddStrict}(\Gamma)$ is the set of all expressions, respectively environments that can be obtained by replacing any number of lets in e, respectively Γ with let!s.

$$\mathit{AddStrict}(e) = \{e' \mid (e')^{-!} = e\}; \quad \mathit{AddStrict}(\Gamma) = \{\Gamma' \mid (\Gamma')^{-!} = \Gamma\}$$

The definition above induces the need of an extension of the semantics of expressions to a semantics of *sets* of expressions.

Definition 3.4. Semantics of sets of expressions. In order to formally reason about the semantics of expressions after the addition of strictness, it must be possible to apply the meaning predicate $[\![]\!]$ to sets of expressions and environments, instead of to single expressions and environments. This is realized as follows:

$$[\![E]\!]_{\{\!\{\Gamma s\}\!\}\rho_0} = \{[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \mid e \in E, \Gamma \in \Gamma s\}$$

We are now almost ready formalize the "folklore" ABC. We will use the standard lazy denotational and operational meanings of [14] and indicate them by $[\![]\!]^{lazy}$ and $\Downarrow^{lazy}$. It goes without saying that $[\![]\!]^{lazy}$ and $\Downarrow^{lazy}$ are equivalent to $[\![]\!]$ and $\Downarrow$ for expressions and environments that do not contain any strict *let* expressions.

Theorem 3.5 (Formal Folklore ABC)**.**

$$\begin{array}{ll} A: & [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \bot \\ & \Rightarrow [\![\mathit{AddStrict}(e)]\!]_{\{\!\{\mathit{AddStrict}(\Gamma)\}\!\}\rho_0} = \{\bot\} \\ B: & [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = z \\ & \Rightarrow [\![\mathit{AddStrict}(e)]\!]_{\{\!\{\mathit{AddStrict}(\Gamma)\}\!\}\rho_0} \subseteq \{\bot\} \cup \mathit{AddStrict}(z) \\ C: & [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = z \\ & \Rightarrow [\![e^{-!}]\!]_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = z^{-!} \end{array}$$

Proof. The proofs proceeds by straightforwardly combining computational adequacy (for lazy and for mixed semantics) and the three additional Theorems 3.6, 3.7 and 3.8 below that capture the essential properties of !-removal.

Consider e.g. property C: applying computational adequacy on $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = z$ yields that $\Gamma : e \Downarrow \Delta : z$, applying Theorem 3.7 gives $\exists\Theta.\ \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}$ and computational adequacy gives the required $[\![e^{-!}]\!]_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = z^{-!}$. □

Theorem 3.6 (Meaning of !-removal)**.**

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

Proof. Since by definition both for lazy and mixed semantics $[\![e]\!]_{\{\!\{x_1 \mapsto e_1\}\!\}\rho} = [\![e]\!]_{\rho \sqcup (x_1 \mapsto [\![e_1]\!]_\rho)}$, a difference between lazy and mixed meaning can only occur when the mixed semantics is $\bot$ due to a *let*!-rule. So, if $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \bot$ then $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = [\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} \neq \bot$. □

Theorem 3.7 (Compare with Lazy Reduction).

$$\Gamma : e \Downarrow \Delta : z \quad \Rightarrow \quad \exists \Theta.\ \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!} \wedge [\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}\rho_0} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho_0}$$

Proof. Assume we have $\Gamma : e \Downarrow \Delta : z$ with derivation tree σ. Compare the operational rules for *let*! and *let*. The condition on the right of the *let*! rule has (up to !-removal) the very same expressions as the *let*! rule but a different environment. This environment captures the 'extra' non-lazy reductions that are induced by the *let*!-rule. Clearly, there is an environment Θ such that $\Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Theta : z^{-!}$. By lazy correctness and computational adequacy $[\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = [\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}\rho_0} \neq \perp$. By mixed correctness and Theorem 3.6 it follows that $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho_0} = [\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = [\![z^{-!}]\!]^{lazy}_{\{\!\{\Theta\}\!\}\rho_0} \neq \perp$. □

Theorem 3.8 (Reduction and !-removal).

$$\Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Delta : z^{-!} \Rightarrow [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \perp \vee \exists \Theta.\ \Gamma : e \Downarrow \Theta : z \wedge [\![z]\!]_{\{\!\{\Theta\}\!\}\rho_0} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho_0}$$

Proof. Assume that $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \perp$ then by mixed computational adequacy $\exists \Theta.\ \Gamma : e \Downarrow \Theta : z$ and by mixed correctness, Theorem 3.6 and lazy correctness $[\![z]\!]_{\{\!\{\Theta\}\!\}\rho_0} = [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = [\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho_0}$. □

4. Example proofs with mixed semantics

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

The lazy semantics as defined by Launchbury [14] makes it possible to yield $\lambda x.\Omega$ (Ω is defined below) and Ω as different results. However, in such 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 [2]. 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 $\perp$. 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 $\perp$. Note that it is *not* possible to return anything else than $\perp$ in the Ω case.

Theorem 4.1 ($\lambda x.\Omega$ and Ω can be distinguished).

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

$$\nexists \Delta, z.\ \{\} : f\ \Omega\ \Downarrow \Delta : z \tag{4.1}$$

$$\exists \Delta.\ \{\} : f\ (\lambda x.\Omega)\ \Downarrow \Delta : 42 \tag{4.2}$$

Proof. For proving property 4.1 we have to prove that it is impossible to construct a finite derivation according to the operational semantics. Applying Theorem 2.5, the computational adequacy theorem, it is sufficient to show that the denotational meaning of $f\ \Omega$ is undefined. The proof is as follows using the denotational semantics:

$[\![f\ \Omega]\!]_{\rho_0}$

$= [\![(\lambda x.let!\ y\ =\ x\ in\ 42)(\Omega)]\!]_{\rho_0}$

$= ([\![\lambda x.let!\ y\ =\ x\ in\ 42]\!]_{\rho_0}) \downarrow_{Fn} ([\![\Omega]\!]_{\rho_0})$
$= (Fn\ (\lambda v.[\![let!\ y\ =\ x\ in\ 42]\!]_{\rho_0 \sqcup (x \mapsto v)})) \downarrow_{Fn} ([\![\Omega]\!]_{\rho_0})$
$= (\lambda v.[\![let!\ y\ =\ x\ in\ 42]\!]_{\rho_0 \sqcup (x \mapsto v)})[\![\Omega]\!]_{\rho_0}$
$= [\![let!\ y\ =\ x\ in\ 42]\!]_{\rho_0 \sqcup (x \mapsto [\![\Omega]\!]_{\rho_0})}$
$= \bot$ since $[\![x]\!]_{\rho_0 \sqcup (x \mapsto [\![\Omega]\!]_{\rho_0})} = (\rho_0 \sqcup (x \mapsto [\![\Omega]\!]_{\rho_0}))(x) = [\![\Omega]\!]_{\rho_0} = \bot$ since for Ω no derivation can be made. □

Proof. The proof of property 4.2 is given by a derivation in the operational semantics written down as in Sect 2.1. To work with numerals we assume the availability of a standard reduction rule (*Num*) that states that each numeral reduces to itself.

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

□

5. Related work

In [9] a case study is done in program verification using partial and undefined values. They assume proof rules to be valid for the programming language. The connection with our approach could be that our formal semantic approach can be used as a basis to prove their proof rules.

With the purpose of deriving a lazy abstract machine Sestoft [17] 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 [17] the rules given by Launchbury are not *fully lazy.* Full laziness can be achieved by introducing new let-bindings for every maximal free expression [11].

Another extension of Launchbury's semantics is given by Baker-Finch, King and Trinder in [3]. They construct a formal semantics for Glasgow Parallel Haskell on top of the standard Launchbury's semantics. Their semantics that are developed for dealing with parallelism, are equivalent to our semantics that are developed independently for dealing with strictness. Equivalence can be shown easily by translating **seq** into *let*!-expressions. They do not prove properties expressing relations between 'lazy' and 'strict' terms.

As part of the **Cover** project [7], it is argued in [8] that "loose reasoning" is "morally correct", i.e. that if, under the assumption that every subexpression is strict and terminating, you can prove your theorem than the theorem will also hold in the lazy case under certain conditions. However, the conditions that are found in this way, may be too restrictive for the lazy case. The Nijmegen proof assistant **Sparkle** [10] has several facilities for defining and proving the proper definedness conditions [20].

6. Conclusions

We have extended Launchbury's lazy graph semantics with a construct for explicit strictness. We have explored what happens when strictness is added or removed within such 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 formalize 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. We have proven this formally.

Acknowledgement

We would like to thank the anonymous referee(s) of (an earlier version of) this paper for the(ir) valuable comments.

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] C. Baker-Finch, D. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173, Montreal, Canada, 2000.

[4] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 1987.

[5] R. S. Bird. *Introduction to Functional Programming using Haskell, second edition.* Prentice Hall, 1998. ISBN 0-13-484346-0.

[6] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph writing. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 364–384, London, UK, 1987. Springer-Verlag.

[7] T. Coquand, P. Dybjer, J. Hughes, and M. Sheeran. Combining verification methods in software development. Project proposal, Chalmers Institute of Techology, Sweden, December 2001.

[8] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. *SIGPLAN Not.*, 41(1):206–217, January 2006.

[9] N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *Prcoeedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

[10] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *Selected Papers from the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–72, Stockholm, Sweden, 2001. Springer Verlag.

[11] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 265–276. ACM Press, 2001.

[12] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[13] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[14] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[15] R. Plasmeijer and M. van Eekelen. Keep it clean: a unique approach to functional programming. *SIGPLAN Not.*, 34(6):23–31, 1999.

[16] R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.

[17] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[18] M. van Eekelen. *Parallel Graph Rewriting - Some Contributions to its Theory, its Implementation and its Application*. PhD thesis, University of Nijmegen, The Netherlands, 1988.

[19] M. van Eekelen and M. de Mol. Mixed lazy/strict natural semantics. Technical Report NIII-R0402, Nijmegen Institute for Computing and Information Science, January 2004.

[20] M. van Eekelen and M. de Mol. Proof tool support for explicit strictness. In A. Butterfield, C. Grelck, and F. Huch, editors, *IFL*, volume 4015 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2005.