Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics

Marko van Eekelen, Maarten de Mol

Department of Computer Science
University of Nijmegen, the Netherlands
{marko,maartenm}@cs.kun.nl

Abstract. Many functional programmers are familiar with the concept of enforcing strictness for making applications fit their time and space efficiency requirements. Few functional programmers however, are familiar with the consequences of enforcing strictness for formal reasoning about their programs.
This paper attempts to fill the gap between the few and the many. Some typical examples are given of the use and the meaning of explicit strictness. We show how formal reasoning can be made easier by the introduction of auxiliary functions in the program.
John Launchbury's [Lau93] natural lazy semantics for lazy evaluation is extended with an explicit strict let construct. We show that this rule extends the semantics in a natural way. In fact, using our 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.

1 Introduction

Discrepancies between formal reasoning and implementation considerations can be solved by including formal reasoning in the programming process. Although it has often been stated that functional programming languages are well suited for formal reasoning, there is in practice little specific support for reasoning about functional programs. Of course, there are well-established theorem provers, such as Pvs[ONRSC99], Coq[Tea98] and Isabelle[Pau01], but they are set up for *theoreticians*. They do not support the full semantics of functional languages and can only be used if the program is translated first, making them difficult to use for a *programmer*. To make formal reasoning available for functional programmers, recently Sparkle[dMvEP01] was developed.

Sparkle is a semi-automatic theorem prover that can be used to reason about any program written in the functional language Clean[vEP98]. It supports all functional concepts and has a semantics based on lazy graph rewriting. Using Sparkle programmers can easily state and prove properties of parts of programs. This *on-the-fly* proving can only be accomplished if reasoning requires little effort and time. This is already achieved by Sparkle for smaller programs, mainly due to the possibility to reason on source code level and the support for automatic proving. It is the intention that SPARKLE will be further integrated in the language CLEAN and its IDE. SPARKLE can be downloaded at `http://www.cs.kun.nl/~clean` and at `http://www.cs.kun.nl/sparkle`.

It is the experience gained in the SPARKLE project that gave rise to the authors' opinion that more theoretical and pragmatical background was needed for *formal reasoning* with *explicit strictness* in a lazy language.

This paper deals with semantical aspects concerning strictness in the context of lazy functional languages in general and of the languages HASKELL[Hud00] and CLEAN in particular. Programming examples will be written in CLEAN.

In section 2 the need for mixed lazy/strict semantics is motivated. Section 3 first introduces the required semantical definitions. Then, in this section the required properties, such as correctness and computational adequacy are shown to be true.

An example of a proof using these mixed semantics can be found in section 4. Finally, sections 5 and 6 discuss related work and give concluding remarks.

2 Mixed lazy and strict reasoning

2.1 Explicit strictness

Although it is seldomly mentioned in papers and presentations, explicit strictness is present in almost every lazy language (and in almost every program) that is used in real-world examples.
In these 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 have arguments which are declared strict due to strictness analysis or due to the programmers annotations),
- for *enforcing the evaluation order* in interfacing with the outside world.

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 (force and seq: HASKELL),
- special inside implementations (monads: CLEAN, 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 it is essential in order to make their applications satisfy the efficiency requirements. For reasoning about these programs, however, they tend to forget strictness all together. This can lead to unexpected non-termination when programs are changed by hand or automatically transformed relying on such reasoning. For reasoning with strictness, their is only little theory and little practical guidance available so far.

Definition 1 (Mathematical Strictness). *A function f is mathematically strict in an argument x if the result of f applied to x is undefined if x is undefined.*

Usually, this is denoted as follows:

$$f\bot = \bot$$

The identity function and the equality function that are commonly found in functional languages, are usually mathematically strict in (both) arguments.

Definition 2 (Operational Strictness). *A function f is operationally strict in an argument x if the argument is always reduced to weak head normal form before the function application is evaluated and the result of the function is undefined if the argument is undefined.*

If a function is mathematically strict in its argument then that function can be made operationally strict without changing its semantics.

Definition 3 (Notational Strictness). *A function f is notationally strict in an argument x if the argument is somehow explicitly annotated as such.*

In Clean strictness is denotated by adding an ! before the argument in the type of the function (or data structure).

With taking strictness into account arguments of a function f that are notationally strict, will not only be treated are interpreted operationally strict by the compiler, but also mathematically strict by the semantics. Consequently, **when a mixed semantics is used notational strictness implies both operational and mathematical strictness!**

It is often recommended to use notational strictness in mixed semantics only when in lazy semantics mathematical strictness holds (because the meaning of a program is not affected then). However, this recommendation is in many cases (interfacing, strict data structures, efficiency) not sensible at all since for these cases it is simply the intention to change the meaning of the program from lazy to strict.

2.2 Some reasoning examples using explicit strictness

When a strict semantics is used, many "intuitive" properties suddenly turn out to be untrue. For instance, the following property which describes a relation between the mathematical = an the operational `==` does not hold:

$$\forall_{a \in \texttt{Eq}} \forall_{x \in a} \forall_{y \in a} [x = y \Leftrightarrow (x == y) = True]$$

If both x and y are undefined, then the equality $x = y$ holds but the expression x `==` y is neither $False$ nor $True$ but undefined. The reason is that the function `==` is a predefined operationally strict function.

If you would indicate strictness in the type information by adding a !-annotation before the strict argument (as is done in Clean)then the type of `==` is specified as follows:

```
(==) :: !a !a -> Bool
```

A property that does hold is the following:

$$\forall_{a\in\texttt{Eq}}\forall_{x\in a}\forall_{y\in a}[x \neq \bot \wedge y \neq \bot \Rightarrow (x = y \Leftrightarrow x == y)]$$

Many other properties like e.g.

$$\forall_{a\in\texttt{Eq}}\forall_{x\in a}\forall_{y\in a}[x == y \Leftrightarrow y == x]$$

do hold unconditionally.

Note that the function `==` may not be mathematically strict on all objects. Suppose we are comparing strings (represented as unboxed arrays of characters). If the first character of the first argument is different from the first character of the second argument then the result could mathematically be determined regardless whether the other characters are defined or not.

An indication of the need for additional strictness conditions can be obtained from the standard library of SPARKLE: of the 115 supplied theorems no fewer than 19 have one or more strictness conditions.

A statement that is not present in this library is the following:

$$\forall_{a\in\texttt{Eq}}\forall_{x\in a}\forall_{xs\in[a]}\forall_{p\in a\rightarrow\texttt{Bool}}[\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = \texttt{isMember}\ x\ xs\ \&\&\ p\ x]^1$$

The CLEAN definitions that this property refers to, are listed below:

```
isMember :: a ![a] -> Bool | Eq a
isMember x [y:ys]    = y == x || isMember x ys
isMember x []        = False


filter :: (a -> Bool) ![a] -> [a]
filter p [x:xs]
    | p x           = [x:filter p xs]
    | otherwise     = filter p xs
filter p []         = []
```

The definitions above make use of several CLEAN-specific notations and features:

- "`| Eq a`" denotes class restriction (HASKELL: `(Eq a) =>` ...);
- "`[` ··· `:` ··· `]`" for lists (HASKELL: `:`, an infix constructor);
- "`!`" indicates notational strictness, so `isMember` x $\bot = \bot$ and `filter` x $\bot = \bot$;
- The functions "`==`" and "`||`" are defined in the standard library (`StdEnv`) of CLEAN and are notationally strict ("`==`" in both arguments, "`||`" only in the first argument).

Note that due to laziness `isMember` $\bot$ `[ ]` = `False`. So, `isMember` $\bot$ `[ ]` $\neq \bot$. Similarly, due to laziness it holds that `True ||` $\bot$ = `True` but due to notational strictness $\bot$ `|| True` = $\bot$.

In the rest of this paper, we will use the following abbreviation:

^{1} Note that for predicates the "= True" is not explicitly written anymore.

$$P(x, xs, p) := \texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = \texttt{isMember}\ x\ xs\ \&\&\ p\ x$$

Intuitively, many lazy functional programmers will consider $\forall x \forall xs \forall p [P(x, xs, p)]$ to be valid without restrictions. There are several situations, however where it fails:

1. $x = \bot \wedge xs \neq [\,] \wedge \forall_y : \ p\ y \ = \ False$
   Then:
   - $\texttt{filter}\ p\ xs = [\,]$
   - $\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = False$
   - $\texttt{isMember}\ x\ xs = \bot$
   - $p\ x = False$

   So, $False \neq \bot\ \&\&\ False$.
2. $x \neq \bot \wedge xs = [\bot, x] \wedge \forall_y : \ p\ y \ = \ False$
   Then:
   - $\texttt{filter}\ p\ xs = [\,]$
   - $\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = False$
   - $\texttt{isMember}\ x\ xs = \bot$
   - $p\ x = False$

   So, $False \neq \bot\ \&\&\ False$
3. $xs = [\,] \wedge \forall_y : \ p\ y \ = \ \bot$
   Then:
   - $\texttt{filter}\ p\ xs = [\,]$
   - $\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = False$
   - $\texttt{isMember}\ x\ xs = False$
   - $p\ x = \bot$

   So, $False \neq False\ \&\&\ \bot$.

In fact, several conditions are required to ensure that P holds:

$$x \neq \bot \wedge \mathit{IsFiniteAndFullyNonBottom}(xs) \wedge \forall_y : \mathit{Total}(p, y) \rightarrow P(x, xs, p)$$

This last statement introduces two special conditions *IsFiniteAndFullyNon-Bottom* and *Total*. It is, however, not easy to formalize these conditions. One needs to define a special class of functions `eval` that return `True` for completely defined arguments (i.e. reduced completely to *normal form*) and that are undefined otherwise. This can be done on CLEAN using notational strictness as follows (the definitions below are present in an extension of `StdEnv`: `StdSparkle`):

```
// Sparkle
class eval a :: !a -> Bool

// similar instances are available for other types
instance eval Int where
    eval :: !Int -> Bool
    eval x = True

instance eval [a] | eval a
where
    eval :: ![a] -> Bool | eval a
    eval []     = True
    eval [x:xs] = eval x && eval xs
```

Using this eval function, the last property with the predicates *IsFiniteAndFullyNonBottm* and *Total* can be expressed as follows:

$$eval\ x \wedge eval\ xs \wedge \forall_y :\ eval\ y \rightarrow eval\ (p\ y) \rightarrow P(x, xs, p)$$

Note that the condition $\forall_y : eval\ y \rightarrow eval\ (p\ y)$ can be weakened: it only needs to hold for all y in the list xs. this can be expressed using the following auxiliary function:

```
evalFilter :: (a -> Bool) ![a] -> Bool
evalFilter p []     = True
evalFilter p [x:xs] = eval (p x) && evalFilter p xs
```

The complete statement can now be expressed in (and, of course, also proved by) Sparkle as follows:

$$eval\ x \wedge \texttt{evalFilter}\ p\ xs \rightarrow P(x, xs, p)$$

By expressing properties about auxiliary CLEAN functions it is possible to write quite expressive and elegant statements. Another example of such a useful function is given below (it expresses finiteness of an argument list).

```
finite :: ![a] -> Bool
finite [x:xs] = finite xs
finite []     = True
```

2.3 Extensionality

The property of extensionality is often considered to be universal.

Unfortunately, there is a (rather obscure) example of a function for which the property of extensionality does not hold. This example does not make use of strictness and it is therefore valid both for lazy and mixed semantics.

```
// example of invalid extensionality
H ::  a -> b
H x = H x

F :: a
F = F
```

With the definitions above $\texttt{F}\ x = \texttt{H}\ x$ for all x since the meaning of both is undefined. Surprisingly, the property $\texttt{F} = \texttt{H}$ does not hold, since `H` has a weak head normal form (and is thus defined) while `F` is undefined. It is therefore not safe to replace `F` by `H` in programs.

The problem can be corrected by strengthening the property of extensionality as follows:

Definition 4 (Extensionality).

$$(f = \perp \Leftrightarrow g = \perp) \Rightarrow [\ \forall x\ f\ x = g\ x \Rightarrow f = g\ ]$$

This extra condition is needed in lazy semantics as well as in mixed semantics. So, in fact *referential transparency is conditional!*

2.4 Reducing the workload to one single construct

As seen in section 2.1, there are several ways to express strictness in a functional program. Because the concept used is the same, it is possible to translate all these alternatives to one single, universal strictness construct.

This universal construct is the strict let, which is a non-recursive strict variant of a normal let. The difference is that the strict let is not-recursive and it enforces the stored expressions to evaluate to weak head normal form before all evaluation continues with the let expression. It is denoted in CLEAN by `#!`.

Below we list typical examples of translations to the strict let. It is not our intention to completely describe the most efficient translation that a compiler would use. We just want to show that these different kinds of strictness can all be translated to a single construct.

First consider a typical example concerning a strictness annotation in a function type `F :: a !b -> c`. The idea is simply to add a `#!` for each strict argument in each application of `F`.

```
//
// expressing function strictness using a strict let construct in Clean
//

// F type definition with strict annotation
F :: a !b -> c

Start = F x y

// is replaced by

Start
#! y_strict = y
= F x y_strict
```

In general $E(\mathtt{F}\ e_1\ e_2)$ is replaced by $E(let!\ y_{strict} = e_2\ in\ \mathtt{F}\ e_1\ y_{strict})$[2].

Then, consider a typical example of a (partially) strict data type definition:

```
//
// expressing data type strictness using a strict let construct in Clean
//

// type definition with strict annotation: tail strict lists
:: TailStrictList a = TCCons a !(TailStictList a)
                    | TCNil


Start = TCCons a as

// is replaced by
```

[2] For a curried application of F replace $E(\mathtt{F})$ by $E(\lambda x.\lambda y.let!\ y_{strict} = e_2\ in\ \mathtt{F}\ e_1\ y_{strict})$

```
Start
#! as_strict = as
= TCCons a as
```

Again, we add a `#!` for each strict argument in each application of `F`.

In general $E(\texttt{TCCons}\ e_1\ e_2)$ is replaced by $E(let!\ y_{strict} = e_2\ in\ \texttt{TCCons}\ e_1\ y_{strict})$[3]. So, the general transformation for data constructors is quite similar to the one for functions.

The required syntactical transformations can all be easily formalized. So, semantically only the extension with a rule for a non-recursive strict let (denoted in the semantics as let!) is needed in order to express all these different kinds of explicit strictness.

3 Natural semantics with explicit strictness

We first recall the basic semantical rules of Launchbury's natural semantics [Lau93].

$$\Gamma : \lambda\ x.e \Downarrow \Gamma : \lambda\ x.e \qquad \textit{Lambda}$$

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

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}}\qquad \textit{Variable}$$

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

A rule for reducing strict lets must be added to the system.This rule is quite similar to the rule for a normal let, but it adds a condition to enforce the evaluation of the expression to be shared:

$$\frac{\Gamma : e_1 \Downarrow \Theta : e_1' \qquad (\Gamma, x_1 \mapsto e_1) : e \Downarrow \Delta : z}{\Gamma : let!\ x_1\ =\ e_1\ in\ e\ \Downarrow \Delta : z}\ \textit{StrictLet}\ [4]$$

[3] For a curried application of `TCCons` replace $E(\texttt{TCCons})$ by $E(\lambda x.\lambda y.let!\ y_{strict} = e_2\ in\ \texttt{TCCons}\ e_1\ y_{strict})$

[4] Clearly, it would also have been possible to define the *StrictLet* rule writing $x_1 \mapsto e_1'$ instead of $x_1 \mapsto e_1$ since this is closer to how reduction is actually performed. But in this way, the theory is more close to [Lau93].

Note, that the environment need not be changed for the evaluation of e_1 since the strict let is not recursive.

¿From this definition it is intuitively clear 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.

If we would replace all let!'s by standard let's, the weak head normal forms would not change. However, if we would replace all let!'s by let! in an expression, then the weak head normal form would be either the same or it would be undefined.

These properties will be proven in the next section.

3.1 Proving Normalization, Relation to Denotational Semantics and Computational Adequacy

[most proofs in this section will be filled in later.....]

We first extend the meaning function of [Lau93] with the meaning of the new let! construct.

As in [Lau93] we have a function domain following Abramsky and Ong [Abr90], [AO93], and we use Fn and $\downarrow_{Fn}$ as lifting and projection functions. An environment is a function from variables to values where the domain of values is some appropriate domain, also containing functions on values and a least element $\perp$. We also use the special semantic environment function $\{\!\{\ \}\!\}\rho$. It resolves the possible recursion and is defined as : $\{\!\{x_1 \mapsto e_1 \cdots x_n \mapsto e_n\}\!\}\rho = \mu\rho'.\rho \cup (x_1 \mapsto [\![e_1]\!]_{\rho'} \cdots x_n \mapsto [\![e_n]\!]_{\rho'})$. Furthermore we use the same ordering on environments expressing that larger environments bind more variables but have the same values on the same variables: $\rho \leq \rho'$ means $\forall x.[\rho(x) \neq \perp \Rightarrow \rho(x) = \rho'(x)]$.

Definition 5 (Meaning Function).

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

In extension to [Lau93] we defined above a meaning for the *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.

Before establishing the required properties, we would first like to study the correspondence between the meaning function defined here and the meaning function defined in [Lau93].

Definition 6 (Replacement of *let*! by *let* 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:*

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

Definition 7 (Replacement of *let*! by *let* for environments). *The function $^{-!}$ is defined on environments such that $\Gamma^{-!}$ is the environment Γ in which in every binding every expression eis replaced by the corresponding expression $e^{-!}$:*

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

Note that in the definition above the empty environment is indicated by { }.

Below, we will indicate the meaning function of [Lau93] (which is exactly the same as our meaning function with the exception of the rule for *let*!) by $[\![]\!]^{lazy}$. The next theorem establishes a close relation between the semantics with *let*! and the semantics without *let*. In fact, the only difference is that more terms are assigned the meaning bottom. Consequently, every term that has meaning non-bottom in the mixed semantics will also have meaning non-bottom in the lazy semantics.

Theorem 1 (Compare Meanings). *The meaning of expressions with let! is the same as Launchbury's meaning for expressions with let, with the exception that more expressions get the meaning $\perp$.*

$$([\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq [\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0}) \Rightarrow [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} = \perp$$

Proof. To be filled in later....

Note that in the definition above the initial semantic environment is indicated by ρ_0.

As a direct consequence of theorem 1 the following holds:

$$[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0} \neq \perp \Rightarrow (\ [\![e^{-!}]\!]^{lazy}_{\{\!\{\Gamma^{-!}\}\!\}\rho_0} = [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho_0})$$

Similarly to $[\![]\!]^{lazy}$, we will indicate the reduction semantics of [Lau93] with $\Downarrow^{lazy}$. Again a close relationship between reduction with *let*! and reduction without *let*! can be established. For all cases where e reduces to z in the mixed semantics, $e^{-!}$ also reduces to $z^{-!}$ in the lazy semantics and consequently the lazy meaning of $e^{-!}$ is non-bottom. This can even be strengthened: if $e^{-!}$ reduces to $z^{-!}$ in the lazy semantics, then either e reduces to z in the mixed semantics or the meaning of e is bottom.

Theorem 2 (Compare Reduction). *The reduction semantics of expressions with let! is the same as Launchbury's reduction for the expressions with let, with the exception that less expressions have a weak head normal form in mixed semantics.*

$$\begin{array}{ll} \Gamma : e \Downarrow \Delta : z & \Rightarrow \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Delta^{-!} : z^{-!} \\ \Gamma^{-!} : e^{-!} \Downarrow^{lazy} \Delta^{-!} : z^{-!} & \Rightarrow \Gamma : e \Downarrow \Delta : z \vee [\![e]\!]_{\rho} = \bot \end{array}$$

Proof. To be filled in later....

In establishing standard semantical properties we will follow the structure of Launchbury's paper [Lau93]. We show that each of the theorems for the natural semantics also holds for the extension with explicit strictness.

Theorem 3 (Distinct Names). *If $\Gamma : e \Downarrow \Delta : z$ is distinctly named, then every heap/term pair occurring in the proof of the reduction is also distinctly named.*

Proof. We only have to consider the StrictLet rule, which is trivial since no renaming takes place there. The proof for the other cases is unchanged with respect to [Lau93].

Our correctness theorem must differ slightly from [Lau93] in order to make the induction work for the *let*! case. The problem is the recursive $[\![e_1]\!]_{\rho} = \bot$ condition in the definition of the meaning function. It requires us to prove ($\Gamma : e \Downarrow \Delta : z$ implies $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} \neq \bot$) and ($\Gamma : e \Downarrow \Delta : z$ implies $[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho}$) at the same time.

Theorem 4 (Correctness). *If $\Gamma : e \Downarrow \Delta : z$ then for all environments ρ,*

$$[\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} \neq \bot \wedge [\![e]\!]_{\{\!\{\Gamma\}\!\}\rho} = [\![z]\!]_{\{\!\{\Delta\}\!\}\rho} \;\wedge\; \{\!\{\Gamma\}\!\}\rho \leq \{\!\{\Delta\}\!\}\rho$$

Proof. Induction on the structure of e, to be filled in later.

Theorem 5 (Computational Adequacy).

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

Proof.

$\Leftarrow$*:* Follows immediately from theorem 4.

$\Rightarrow$*: proof sketch* The $\Rightarrow$ part requires somewhat more effort.
Using theorem 1 we find

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

Then, we know from [Lau93] that $\exists \Delta, z \;.\; \Gamma : e \Downarrow \Delta : z$. If we take the derivation that shows that e reduces to z, then this will also be a proof in our extended semantics.

4 Example proofs using the mixed semantics

4.1 An example that distinguishes between Ω and $\lambda x.\Omega$

A semantically interesting aspect of explicit strictness is that it allows the programmer to distinguish between $\lambda x.\Omega$ and Ω.

The standard lazy semantics [Lau93] makes it possible to yield these values as different results. However, in that semantics it is not possible to write a function `F` that *produces a different result* depending on which one is given as an argument. We say that two terms "produce a different result" if either a different basic value (like `1` or `True`) can be produced or one term does not terminate and the other produces a basic value.

So, in lazy natural semantics these two different values belong to a single equivalence class of which the members cannot be distinguished by the programmer.

With mixed semantics a definition for such a function `F` is certainly possible. Below a Clean definition for such an `F` is given. 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.

```
H :: a -> b       // H is the typed equivalent of (Lambda x.Omega)
H x = H x         // and H 1 is the typed equivalent of Omega

F :: a -> Int
F x
#! y = x
= K 42 y

K :: a b -> a
K x y = x

Start :: Int
Start = F H        // --> reduces to 42
// Start = F (H 1) // --> bottom, infinite reduction
```

4.2 Proving the example with mixed semantics

To illustrate the way proofs of reduction can be made using the mixed semantics, we show the reduction proofs of this example below.

First we have to transform the definitions slightly in order to fit the logical framework. We define (using without loss of generality Ω as equivalent to $H1$ in order to shorten the proofs):

$$\begin{aligned}
H &\equiv \lambda x.\Omega\\
H1 &\equiv \Omega\\
\Omega &\equiv (\lambda x.xx)(\lambda x.xx)\\
K &\equiv \lambda a.\lambda b.a\\
F &\equiv \lambda x.(let!\ y = x\ in\ (K\ 42\ y))
\end{aligned}$$

We will prove two properties:

$$\exists \Delta . \{\} : F H \Downarrow \Delta : 42 \tag{1}$$
$$\forall \rho . [\![F(H 1)]\!]_{\rho} = \bot \tag{2}$$

To work with numerals we need an extra rule for dealing with them:

$$\Gamma : \ n \Downarrow \Gamma : \ n \qquad \mathit{Numerals}$$

We write down proofs similar to [Lau93] with sub-derivations contained within square brackets.

$$\left[\begin{array}{l} \Gamma : e \\ \left[\begin{array}{l} \\ \text{a sub-proof} \\ \\ \end{array}\right. \\ \left[\begin{array}{l} \\ \text{another sub-proof} \\ \\ \end{array}\right. \\ \Delta : z \end{array}\right.$$

The proof of the first property is given below:

$$\begin{array}{l}
\{\ \} : F\ H \\
\{\ \} : F\ (\lambda x.\Omega) \\
\{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ (\lambda x.\Omega) \\
\quad \{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ (\lambda x.\Omega) \\
\quad \{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ (\lambda x.\Omega) \\
\quad \{\ \} : (let!\ y = x\ in\ K\ 42\ y)\ [x/\lambda x.\Omega] \\
\quad \{\ \} : let!\ y = \lambda x.\Omega\ in\ K\ 42\ y \\
\quad\quad \{\ \} : \lambda x.\Omega \\
\quad\quad \{\ \} : \lambda x.\Omega \\
\quad\quad \{y \mapsto \lambda x.\Omega\} : K\ 42\ y \\
\quad\quad \{y \mapsto \lambda x.\Omega\} : (\lambda a.\lambda b.a)\ 42\ y \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : \lambda a.\lambda b.a \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : \lambda a.\lambda b.a \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : (\lambda b.a)\ [a/42] \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : \lambda b.42 \\
\quad\quad \{y \mapsto \lambda x.\Omega\} : (\lambda b.42)\ y \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : \lambda b.42 \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : \lambda b.42 \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : 42\ [b/y] \\
\quad\quad\quad \{y \mapsto \lambda x.\Omega\} : 42 \\
\quad\quad \{y \mapsto \lambda x.\Omega\} : 42 \\
\quad \{y \mapsto \lambda x.\Omega\} : 42 \\
\{y \mapsto \lambda x.\Omega\} : 42
\end{array}$$

For the proof of the second property theorem 5 is used. It then suffices to prove that it is impossible to construct a derivation for $F(H1) \Downarrow e$ for any e. This is shown by:

$$\left[\begin{array}{l}
\{\ \} : F\ (H1)\\
\{\ \} : F\ \Omega\\
\{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ \Omega\\
\left[\begin{array}{l}
\{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ \Omega\\
\{\ \} : (\lambda x.\ let!\ y = x\ in\ K\ 42\ y)\ \Omega
\end{array}\right.\\
\left[\begin{array}{l}
\{\ \} : (let!\ y = x\ in\ K\ 42\ y)\ [x/\Omega]\\
\{\ \} : let!\ y = \Omega\ in\ K\ 42\ y\\
\left[\begin{array}{l}
\{\ \} : \Omega\\
\{\ \} : (\lambda x.x\ x)(\lambda x.x\ x)\\
\left[\begin{array}{l}
\{\ \} : \lambda x.x\ x\\
\{\ \} : \lambda x.x\ x\\
\left[\begin{array}{l}
\{\ \} : x\ x[x/\lambda x.x\ x]\\
\{\ \} : (\lambda x.x\ x)(\lambda x.x\ x)\\
\vdots
\end{array}\right.\\
\{\ \} : \ldots
\end{array}\right.\\
\{\ \} : \ldots
\end{array}\right.\\
\{\ \} : \ldots
\end{array}\right.\\
\{\ \} : \ldots
\end{array}\right.$$

There is no finite derivation tree for this reduction, because building this tree inevitably leads to an infinite sequence of subtrees.

5 Related work

This work extends the work of Launchbury [Lau93] with explicit laziness. As shown in section 4.2, our extension makes it possible to write programs that distinguish between terms that are different, but indistinguishable, in Launchbury's semantics.

A formal semantics that is similar to Launchbury's, has been defined independently by Barendsen and Smetsers [BS99]. They address strictness but only consider it in the context of a typing system to derive mathematical strictness. So, no mixed semantics is given. It might be worth-while to establish a formal correspondence between Launchbury's and Barendsen-Smetsers semantics such that results can be transferred back and forth.

The Sparkle project [dMvEP01] aims to further integrate programming and formal reasoning. A proof assistant[dMvE99] for Clean has been developed and work is in progress to fully describe the underlying semantical issues (including single-step semantics with strictness in a fully formal semantics for the complete language).

Another project that aims to integrate programming, properties and validation is a project of the Pacific Software Research Center in Oregon: the Programatica project (`http://www.cse.ogi.edu/PacSoft/projects/programatica`). Not much information is available about this project (the papers section on the site only makes 3 out of 8 papers available). The aim of the project seems to be much broader than just functional programming and verification. A wide range of validation techniques for programs written in different languages is intended to be supported. For functional languages they use a logic (P-logic) based on a modal μ-calculus (in which also undefinedness can be expressed). The precise relation between these semantics and the semantics of [Lau93] is unknown.

6 Conclusions

We have discussed reasoning about programs with explicit strictness. We have introduced the use of auxiliary functions on the programming level in order to facilitate the reasoning on the logic level.

We have shown that it is possible and worthwhile to extend the natural lazy semantics of [Lau93] with a construct for explicit strictness.

The resulting derivation system is shown to be correct and computationally adequate. The mixed semantics system is a proper extension of Launchbury's natural semantics. With our mixed semantics it is possible to write expressions that distinguish between terms that have different natural semantics but cannot be distinguished by a term within that semantics.

We hope to have shown that strictness is not just for functional hacking but that it is also possible to reason properly and formally about programs that use strictness explicitly.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[AO93] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, (105):159–267, 1993.

[BS99] Erik Barendsen and Sjaak Smeters. *Graph Rewriting Aspects of Functional Programming*, chapter 2, pages 63–102. In [HEe99], 1999.

[dMvE99] Maarten de Mol and Marko van Eekelen. A proof tool dedicated to clean - the first prototype. In *Proceedings of Applications of Graph Transformations with Industrial Relevance 1999*, volume 1779 of *Lecture Notes in Computer Science*, pages 271–278. Springer Verlag, 1999. ISBN 3-540-67658-9.

[dMvEP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In Thomas Arts and Markus 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.

[HEe99] H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation.* world Scientific, 1999.

[Hud00] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, New York, 2000.

[Lau93] John 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.

[ONRSC99] S. Owre, N.Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference (version 2.3)*, 1999. http://pvs.csl.sri.com/manuals.html.

[Pau01] L. C. Paulson. *The Isabelle Reference Manual.* University of Cambridge, 2001. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html.

[Tea98] The Coq Development Team. *The Coq Proof Assistant Reference Manual (version 7.0).* Inria, 1998. http://pauillac.inria.fr/coq/doc/main.html.

[vEP98] Marko van Eekelen and Rinus Plasmeijer. Concurrent CLEAN language report (version 1.3). Technical Report CSI-R9816, Computing Science Institute Nijmegen, June 1998. http://www.cs.kun.nl/~clean/Manuals/manuals.html.