Proof Tool Support for Explicit Strictness

Marko van Eekelen and Maarten de Mol

marko@cs.ru.nl, maartenm@cs.ru.nl
Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands.

Abstract. In programs written in lazy functional languages such as for example Clean and Haskell, the programmer can choose freely whether particular subexpressions will be evaluated lazily (the default) or strictly (must be specified explicitly). It is widely known that this choice affects program behavior, resource consumption and semantics in several ways. However, not much experience is available about the impact on logical program properties and formal reasoning.
This paper aims to give a better understanding of the concept of explicit strictness. The impact of explicit strictness on formal reasoning will be investigated. It will be shown that this impact is bigger than expected and that tool support is needed for formal reasoning in the context of explicit strictness. We introduce the various ways in which strictness specific support is offered by the proof assistant Sparkle.

1 Introduction

Lazy functional programming languages, such as for example Clean and Haskell, are excellent for developing readable and reliable software. One of their key features is lazy evaluation, which makes it possible to adopt a natural, almost mathematical, programming style. The downside of lazy evaluation, however, is lack of control; it becomes very difficult to predict when subexpressions will be evaluated, which makes resource management a non-trivial task.

This issue has been addressed by the introduction of *explicit strictness*, with which a functional programmer can enforce the evaluation of a subexpression by hand. Adding explicit strictness can indeed change the resource consumption of programs significantly, and it is therefore used a lot in practice. Moreover, explicit strictness can easily be incorporated in the semantics of functional languages, and is therefore theoretically sound as well.

Not all is well, however. In this paper, we will show that the addition (or removal) of strictness to programs may also give rise to many unexpected (and undesirable) effects. Of course, some effects are already widely known, such as for example the possible introduction of non-termination. However, more seriously and not so widely known, is that strictness often breaks program properties. For example, $\forall_{f,g}\forall_l[\texttt{map}\ (f\ o\ g)\ l = \texttt{map}\ f\ (\texttt{map}\ g\ l))]$, does **not** hold for element-strict lists. Furthermore, the addition of strictness also often breaks proofs, as it falsifies proof rules that are based on reduction.

We will demonstrate how to deal with these semantical effects with Sparkle, which is the proof assistant that is dedicated to Clean. Sparkle has been introduced in [6]. In this paper we will present the specific strictness support that it offers. This support has not been addressed in any earlier publication. As far we know, Sparkle is at present the only proof assistant with specific support for reasoning in the context of explicit strictness.

This paper is structured as follows. First, in Section 2 the concept of explicit strictness is introduced, both informally and formally. Also, its effects on program semantics and program transformations are discussed. Then, in Section 3 the effects of explicit strictness on program properties and reasoning will be examined. The kinds of support that Sparkle offers for this purpose will be introduced in Section 4. This support deals with dedicated reasoning steps and with expressing definedness conditions that have to be introduced due to strictness. Finally, Sections 5 and 6 discuss related work and conclusions.

2 The Concept of Explicit Strictness

Although it is seldom mentioned in publications, explicit strictness is present in almost every real-world lazy program. Explicit strictness is used for:

- improving the *efficiency of data structures* (e.g. strict lists),
- improving the *efficiency of evaluation* (e.g. functions that are made strict in arguments that always have to be evaluated),
- *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 explicitly enforce 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).

Implementers of real-world applications make it their job to know about explicit strictness, because without it essential parts of their programs would not work properly. The compiler generates code that takes strictness annotations into account by changing the order of evaluation. It is often thought that the only effects of changes in evaluation order can be on the termination properties of the program as a whole and on the program's resource consumption (with respect to space or time). Therefore, strictness is usually considered an implementation issue only.

However, in the following subsections we will show that explicit strictness is far from an implementation issue only. In Section 2.1 it is illustrated that strictness has a fundamental influence on program semantics, because explicit strictness is not just an option but a must. A surprising example of how radical this influence can be, is given in Section 2.2. Finally, to deal with that influence, formal semantics are extended with strictness in Section 2.3.

2.1 When Strictness is not an option but a must

With explicit strictness, performing an evaluation is not anymore just an option. Instead each explicit strictness annotation constitutes an evaluation obligation that has to be fulfilled before proceeding further. We will illustrate the consequences of this changed evaluation with the following example.

Consider 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`.

Another consequence is that the definition of `f` cannot just be substituted in all its occurrences. Instead 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.

The addition of an exclamation mark by a programmer is therefore more than just a harmless annotation. It also has an effect on the logical properties of functions. 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. An example of such a far-reaching consequence is given in the next subsection.

2.2 A Dramatic Case of the Influence of Explicit Strictness

The Clean compiler uses term graph rewriting semantics [3] to incorporate pattern matching, sharing and cycles. With term graph rewriting semantics, on right-hand sides of definitions those parts that are not *connected* to the root cannot have any influence on the outcome. These definitions are thrown away in a very early stage of the compilation process. Consequently, possible syntactic and semantic errors in such disconnected definitions may not be spotted by the compiler. This can be annoying but it is consistent with the term graph rewriting semantics. When strictness comes into the picture, however, this early connectedness program transformation of the compiler is no longer semantically valid.

This is illustrated by the following example. Take the following Clean programs with definition `K x y = x`:

```
Start                   Start
  #! y = undef            #! y = undef
  = K 42 y                = 42
```

The programs use the `#!`-notation of Clean which denotes a strict let. The strict let will be formally defined in Section 2.3. It forces `y` to be evaluated before the result of `Start` is computed. In Haskell the same effect can be achieved using a `seq`.

For the left program, due to the `#! y` must be evaluated first. So, the result of the program is: "Error: undefined!".

For the right program one would expect the same result. But, the result is *different* since the compiler removes unconnected nodes before any analysis is done, transforming the right program into `Start = 42`. So, the result of the right program is 42. This makes the right program a *wrong* program and the left program the *right* program. Clearly, this is an unwanted situation.

Due to the combination of connectedness and explicit strictness Clean programs do not always have the subject reduction property anymore. The meaning is not always preserved during reduction and it is not always sound anymore to substitute a definition. Clean programs are no longer guaranteed to be referentially transparent. Of course, this situation is acknowledged as a bug in the compiler for several years now. The consequences of removing this bug, however, are so drastic for the structure of the compiler that at this point in time this bug still remains to be present.

It may be a relief to the reader that Sparkle's mixed lazy/strict semantics are not based on connectedness.

2.3 Incorporating Explicit Strictness in Formal Semantics

The semantics of lazy functional languages have been described elegantly in practice in various ways: both operationally and denotationally, in terms of a term-graph rewrite system, in [12]; or just operationally, in terms of a graph rewrite-system, in [14]. All these semantics are well established, are widely known and accepted in the functional language community, and have been used for various kinds of theoretical purposes.

The basic forms of all these semantics, however, are limited to lazy expressions in which no explicit strictness is allowed to occur. If one wants to include strictness, an upgrade is required, because the introduction of strictness in an expression has an effect on its meaning that cannot be described in terms of existing concepts. In other words, strictness has to be accounted for on the semantic level as well. Fortunately, this is very easy to accomplish.

As a starting point we will use the operational semantics of Launchbury[12]. We extend this to a *mixed lazy/strict semantics*, which is able to cope with laziness as well as with strictness.

We will choose to extend expressions with the *strict let*, which is the basic primitive for denoting strictness in Clean. The strict let is a variation of the normal let, which only allows the actual sharing to take place after the expression to be shared has first successfully been reduced to weak head normal form. Moreover, it only allows a single non-recursive expression to be shared at a time. Adding the strict let to the set of expressions leads to:

$$\begin{aligned} e \in Exp ::= {} & \lambda\ x.\ e \\ \mid {} & e\ x \\ \mid {} & x \\ \mid {} & let\ \ x_1 = e_1\ \cdots\ x_n =\ e_n\ in\ e \\ \mid {} & let!\ x = e_1\ in\ e \end{aligned}$$

Due to its similarity with the normal let, the strict let is a convenient primitive that can be added to the semantical level with minimal effort. Naturally, all forms of explicit strictness can easily be expressed in terms of the strict let. This also goes for the basic Haskell primitive, *seq*:

for all expressions e_1, e_2 and fresh variables x,
seq $e_1\ e_2$ is equivalent to $let!\ x = e_1\ in\ e_2$.

Launchbury describes both an operational and a denotational semantics, which both have to be updated to cope with the strict let. Here, we treat the extension of the operational semantics only. This semantics is given by means of a multi-step term-graph rewrite system which has to be extended with a rule for the strict let. The new rule is much like the rule for the normal let, but also demands the reduction of the shared expression to weak head normal form as an additional precondition:

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

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

(for the technical details of this definition: see [12])

The addition of this single StrictLet rule is sufficient to incorporate the concept of explicit strictness in a formal semantics.

Our extension is equivalent to the one that is introduced in [2] for dealing formally with parallelism. In [2] *seq* is used as the basic primitive to denote explicit strictness. Using the equivalence of *seq* and *let!* sketched above, the proofs of soundness and computational adequacy can be applied to our mixed semantics as well.

3 Reasoning in the Context of Strictness

In the previous sections, a general introduction to the concept of explicit strictness has been provided and its, more or less obvious, effects on programs and semantics have been discussed. In this section, the effect of strictness on *reasoning* will be described. We will show that adding or removing strictness requires program properties to be reformulated. As a consequence, the proofs of the reformulated properties may have to be redone from scratch. In addition, certain proof rules may no longer be applicable and have to be replaced as well.

The effects of strictness on reasoning are not so commonly known, mainly because programming and reasoning are usually separate activities that are not carried out by the same person. With this paper, we strive to show that the effects of strictness on reasoning are quite profound and should not be ignored.

3.1 Strictness and Logical Properties

A logical (equational) property about a program is constructed by means of logical operators ($\forall, \exists, \wedge, \vee, \rightarrow, \neg$) out of basic equations of the form $E_1[x_1 \ldots x_n] = E_2[x_1 \ldots x_n]$, where $x_1 \ldots x_n$ are the variables that have been introduced by the quantors. The equations in a property can be divided into a number of conditions that precede a single obligatory conclusion. A property with conclusion $E_1 = E_2$ denotes that E_1 may safely be replaced by E_2 in all contexts, if properly instantiated and if all conditions are satisfied.

Semantically, two expressions can safely be replaced by each other if either: (1) they both compute the exact same value; or (2) they both do not compute any value at all. Note that an expression that does not terminate, or terminates erroneously, may not be replaced by an expression that successfully computes a value. This is desirable, as end-users would not be happy if terminating programs would be replaced by erroneous ones.

If explicit strictness is added to or removed from a program, the value that it computes on success is not affected, but the conditions under which it terminates are. Unfortunately, if the termination conditions of an expression E_1 are changed, but the termination conditions of E_2 stay the same, then a previously valid equation $E_1 = E_2$ will become invalid, because the replacement of E_1 by E_2 is no longer allowed.

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, either by the addition or, quite surprisingly, by the *removal* of termination conditions. This is illustrated briefly by the following two examples:

Example of the addition of a condition:

$\forall_{f,g}\forall_{xs}[\texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)]$

Affected by strictness:

This property is valid for lazy lists, but invalid for element-strict lists.

Invalid in the strict case because:

Suppose $xs = [12]$, $g\ 12 = \bot$ and $f\ (g\ 12) = 7$.
Then $\texttt{map}\ (f\ \texttt{o}\ g)\ xs = [7]$, 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:

$\forall_{f,g,xs}[\forall_{x \in xs}[g\ x \neq \bot] \rightarrow \texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)]$.

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 and can therefore be removed safely in the spine-strict case.

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:

$\forall_{xs}[\texttt{reverse}\ (\texttt{reverse}\ xs) = xs]$

In Section 4.3 it will be shown how mathematical conditions such as *finite xs* and $\forall_x[g\ x \neq \bot]$ can be expressed within the Sparkle framework.

In principle, all invalidated properties can be fixed this way. The termination conditions to be added can be obtained by carefully considering the consequences of components of quantified variables to be undefined. Such an analysis is far from easy, however, and it is easy to forget certain conditions. On paper, this may lead to incorrect proofs; when using a proof assistant, this makes it impossible to prove the property at all.

An automatic analysis to obtain termination conditions would be helpful. This does not seem too far-fetched. An idea is to extend the GAST-system (see [11]) for this purpose. With GAST, it is possible to automatically generate valid values for the quantified variables and test the property on these values. However, GAST currently is not able to cope with undefinedness.

3.2 Strictness and Formal Reasoning

Formal reasoning is the process in which formal proofs are constructed for logical program properties. These proofs are constructed by the repeated application of proof steps. Each proof step can be regarded as a function from a single property to a list of new properties. The conjunction of the produced properties must be logically stronger, and hopefully also easier to prove, than the input property.

In the previous section, it has been shown that the addition or removal of strictness to programs often requires a reformulation of the associated logical properties. This is not the only cumbersome effect of strictness on reasoning, however. A second problem, namely, is that strictness changes the behavior of reduction, and consequently also of proof steps that make use of reduction. This in turn may cause existing proofs to become invalid.

A proof step that makes use of reduction is based on the observation that if e_1 reduces to e_2, then e_1 is also semantically equal to e_2, and therefore e_1 may safely be replaced with e_2 within a logical property to be proved. It is clear that this relation is changed by the introduction of strictness. It is not intuitively clear where this change is problematic for the actual proof process; after all, in the case of program evaluation the change in reduction order was rather harmless.

The hidden reason is the availability of *logical expression variables* within propositions. Such a variable denotes an 'open position', to be replaced with a concrete expression later. It is introduced and bound by means of a (existential or universal) quantor. When reduction is forced, due to explicit strictness, to reduce such a variable to weak head normal form, the following problem occurs:

> Suppose that e is an expression in which the variable x occurs lazily.
> Suppose that e reduces to e'.
> Suppose that within e, x is now marked as explicitly strict.
> Then, the strict version cannot be reduced at all, because the required preparatory reduction of x to weak head normal fails.

In other words: the introduction of explicit strictness causes a previously valid reduction to become invalid. This in turn causes proof steps that depend on it to become invalid. That in turn causes the proof as a whole to become invalid. This effect is illustrated in the following basic example:

Property: $\forall_x[\texttt{id}\ x = x]$.
Proof: Introduce x. Reduce $(\texttt{id}\ x)$ to x. Use reflexivity. QED.
Validity: This proof is only valid if the first argument of `id` is not explicitly marked as explicitly strict. If it is, the expansion of $(\texttt{id}\ x)$ first requires x to be reduced to weak head normal form. Since this is not possible, $(\texttt{id}\ x)$ cannot be reduced at all, and the proof sketched above becomes invalid.

This effect actually occurs quite frequently, which is quite a nuisance. It causes many previously valid proof steps to become invalid, and therefore requires the proofs themselves to be revised. Fortunately, this revision is often easily realized. A general solution, which usually suffices, is to distinguish explicitly between $x = \perp$ and $x \neq \perp$. In the first case, the whole expression reduces to $\perp$. In the second case, it is statically known that x has a weak head normal form, and reduction is therefore allowed to continue in the same way as in the lazy case.

Nevertheless, the introduction of explicit strictness makes reasoning more difficult. To deal with this problem, the proof assistant Sparkle offers specific support to deal with explicit strictness. The following section is devoted to explaining this support.

4 Tool Support for Explicit Strictness in Sparkle

Sparkle [6] is Clean's dedicated proof-assistant. Apart from its location of origin Sparkle is used rather intensively in Budapest (Object Abstraction [16]) and Dublin (I/O models [7]). Sparkle works directly on a desugared version of Clean, called Core-Clean. Within Sparkle allows properties of functions to be expressed using *first-order* propositional logic; however, predicates are not allowed. Sparkle offers the usual operators and quantors with the restriction that quantification is only allowed over typed expressions and propositions.

Basic units:	*True*, *False*, $\perp$, $e_1 = e_2$, x
Operators:	$\neg$, $\wedge$, $\vee$, $\rightarrow$,$\leftrightarrow$
Quantors:	$\forall$, $\exists$

Sparkle is aimed towards making proving possible for the programmer. It contains lots of features to lower the threshold to start with proving theorems about programs:

– it can be called from within the Clean Integrated Development Environment;
– it can load a complete Clean project including all the modules of the project;
– the proof environment is highly interactive, displaying lots of information in lots of different windows;
– the proof tactics are dedicated to the programming language.

Sparkle's reduction semantics are based on term graph rewriting. Sparkle has a total semantics. The constant expression $\perp$ is used to represent the "undefined" value. Both non-terminating reductions and erroneous reductions are equal to $\perp$. For example: `hd` [ ] reduces to $\perp$ on Sparkle's semantic level. Error values propagate stepwise to the outermost level. For example: (`hd` [ ]) + 7 reduces to $\perp$ + 7 reduces to $\perp$.

Sparkle's semantics of equality are based on reduction in a manner which is independent of the reduction strategy. The equality copes with infinite reductions and equalities between infinite structures using the concept of an observation of an expression. The *observation* of an expression is obtained by replacing all its redexes by $\perp$. What remains is the fully evaluated part. Two expressions e_1 and e_2 are equal if: (1) for all reducts r_1 of e_1, there exists a reduct r_2 of e_2 such that the observation of r_1 is smaller than the observation of r_2; and (2) also the analogue property holds for all reducts of e_2. The observational ordering is such that an expression r_1 is smaller than r_2 if r_2 can be obtained by substituting subexpressions for $\perp$'s in r_1.

Being dedicated to the use of a lazy programming language, Sparkle generates on the one hand the required definedness conditions for extensionality ($f = \perp \leftrightarrow g = \perp$), induction ($P(\perp)$) and case-distinction ($P(\perp)$ as well). On the other hand Sparkle also offers specific support for reasoning with definedness conditions in the context of explicit strictness. To our knowledge, Sparkle is currently the only proof assistant that fully supports explicit strictness in the context of a lazy functional programming language. This support consists of three components:

1. a smart reduction proof step: the ‘Reduce’ tactic;
2. a specific ‘Definedness’ tactic; and
3. using an ‘eval’ function to denote definedness conditions.

These three kinds of support are explained in detail in the following sections.

4.1 The ‘Reduce’ Tactic of Sparkle

One of the proof steps (or *tactics*, as they are usually called in the context of mechanized proof assistants) that is made available by Sparkle is ‘Reduce’. This tactic applies reduction within the current logical property to be proved.

Sparkle operates on a basic functional language with a reduction mechanism similar to the one given in Section 2.3. The reduction tactic of Sparkle does *not necessarily* have to correspond completely to the formal reduction relation of this language; instead, it suffices that it is *sound*, meaning that it may only transform e_1 to e_2 if $e_1 = e_2$ formally holds. Of course, the tactic does have to be *based on* reduction, because it must look like normal reduction to the end-user.

This degree of freedom is used by Sparkle to offer specific support for the reduction of explicitly strict subexpressions that contain logical expression variables. The aim of this support is to hide the cumbersome effects of strictness to the user, allowing the same proof style and the same proof rules to be used both for the lazy and for the strict case.

The support offered by Sparkle manifests itself in the following customized behavior when reduction encounters explicit strictness:

- First, reduction is recursively applied to the strict subexpression as usual.
- If this results in either $\perp$ or a weak head normal form, then reduction continues as usual.
- Suppose that, due to logical expression variables, the recursive reduction cannot be completed and instead results in some expression E that is neither $\perp$ nor a weak head normal form.
- Then, and this is new, reduction is allowed to continue anyway, if either:
  - the validity of the statement $\neg(E = \perp)$ can be derived statically on the basis of hypotheses that have been introduced earlier in the proof; or
  - the subexpression that has explicitly been marked as strict is statically determined to be mathematically strict as well.

  Note that in both cases the expansion of the reduction rule is semantically sound. In the first case, $\neg(E = \perp)$ implies that E has a weak head normal form, even though it is not known at this point what it looks like. In the second case, it has been determined that $E = \perp$ will lead to an undefined result anyway, which means that continuing reduction is safe as well.

If either of the two additional conditions applies, then it seems to the user as if reduction has the same effect in the strict case as in the lazy case. In other words: by silently checking for additional conditions, Sparkle can sometimes hide the cumbersome effects of explicit strictness altogether.

To illustrate the additional power of the reduction mechanism, consider the following two basic examples:

Example of continuation of reduction:

Suppose that datatype `(Tree a)` is defined as follows:

```
:: Tree a = Leaf | Edge !a !(Tree a) !(Tree a)
```

Suppose that the function `treeDepth` has the following signature:

```
treeDepth :: !(Tree a) -> Int
```

Suppose that the logical expression variable x (of type `Tree Int`) and the hypothesis $\neg(x = \bot)$ have both been introduced earlier in the proof.

Then, Sparkle allows the function application `treeDepth (Edge 7 Leaf x)` to be expanded, because by means of recursive analysis Sparkle is able to determine that `Edge 7 Leaf x` is unequal to $\bot$.

Example of increased stability of proofs:

Suppose that the identify function is defined as follows:

```
id :: !a -> a
id x = x
```

Sparkle determines statically that the first argument of `id` is mathematically strict, and therefore allows all applications of `id` to be expanded, regardless of the form of its (reduced) argument.

Therefore, the proof that was given in Section 3.2 which was shown to be invalid with a standard reduce tactic, in fact becomes *valid* when the powerful strictness specific 'Reduce' tactic of Sparkle is used.

4.2 The 'Definedness' Tactic of Sparkle

Definedness conditions on variables and expressions occur frequently in proofs. They are introduced by various tactics that take explicit strictness into account, such as 'Induction', 'Case' and 'Assume'. These conditions usually appear in parts of the proof that are not in the main line of reasoning. Therefore, one wishes to get rid of them as soon as possible with as less effort as possible.

Unfortunately, proving definedness conditions often involves several small reasoning steps as is illustrated by the following example:

Example of proving a definedness condition:

$\forall_{x,y}[\neg(x = \bot) \rightarrow y = \texttt{StrictCons}\ 7\ x \rightarrow \neg(y = \bot)]$.

Proof without the Definedness tactic:

Introduce x, y. Assume H1: $x \neq \bot$. Assume H2: $y = \texttt{StrictCons}\ 7\ x$. Assume by Contradiction H3: $y = \bot$. Rewrite H3 in H2, obtaining H4: $\texttt{StrictCons}\ 7\ x = \bot$. Use Injectivity on H4 to Assume H5: $x = \bot$. Contradiction between H1 and H5. QED.

In Sparkle the 'Definedness' tactic is introduced to remove the burden of all such small proofs from the user. This tactic deduces as much definedness information as possible from the hypotheses that have been introduced. It then attempts to prove any goal by deriving a contradiction from the deduced definedness information. More precisely, the definedness analysis is performed as follows:

– Terminology: an expression e is *defined* if it is statically known to be unequal to $\bot$; likewise, e is *undefined* if it is statically known to be equal to $\bot$.

- For each expression E, let $\mathit{AsDefinedAs}(E)$ denote the set V such that:
  - If all elements of V are defined, then E must be defined as well.
  - If one element of V is undefined, then E must be undefined as well.

  The function $\mathit{AsDefinedAs}$ can easily be defined by recursion on the structure of expressions, making use of strictness information where it is available.
- Initially, let $U = \{\bot\}$ (set of undefineds) and $D = \varnothing$ (set of defineds).
- Examine all hypotheses that have been introduced earlier in the proof. For each hypothesis H, perform the following actions:
  - If H is $E_1 = E_2$, and $\mathit{AsDefinedAs}(E_2) \subseteq D$, then add E_1 to D.
  - If H is $E_1 = E_2$, and $\exists_{e \in U}[e \in \mathit{AsDefinedAs}(E_2)]$, then add E_1 to U.
  - If H is $\neg(E_1 = E_2)$, and $\exists_{e \in U}[e \in \mathit{AsDefinedAs}(E_2)]$, then add E_1 to D.
  - All three of the above, but then in vice versa.
- Repeat the previous step until D and U are both stable.

The results of this analysis are used by the tactic 'Definedness'. This tactic first transforms a goal of the form $[H_1 \ldots H_n] \rightarrow P$ to $[H_1 \ldots H_n, \neg P1] \rightarrow \mathit{False}$, and then determines the sets D and U. If it finds any overlap between D and U, it proves the goal immediately. This tactic actually performs two tasks:

- It immediately proves any goal with a contradictory set of hypotheses (with respect to definedness and undefinedness).
- It also immediately proves any statement of the form $E = \bot$ or $\neg(E = \bot)$, if its validity is statically implied by the hypotheses. This is due to the fact that the negation of the statement is treated internally as an added hypothesis.

The special tactic 'Definedness' is quite powerful and very useful in practice. It can be used to automatically get rid of almost all kinds of valid definedness conditions that have been stated in order to keep reduction going in strict contexts. The proof of the example can now be simplified to the use of one single tactic:

Example of proving a definedness condition:
$\forall_{x,y}[\neg(x = \bot) \rightarrow y = \texttt{StrictCons}\ 7\ x \rightarrow \neg(y = \bot)]$.
Proof with the Definedness tactic:
Definedness. QED.

4.3 Using an 'eval' Function to Denote Definedness Conditions

In many cases, it may seem impossible to express definedness conditions just using the first-order logic of Sparkle. For instance, spine evaluation of data-structures is very hard to express. However, the possibility to define functions in the higher-order programming language and the possibility to use these functions as predicates gives unexpected expressive power. The higher-order of the programming language can be combined with the Sparkle's first order logic.

On the programming level we define a function `eval`. The purpose of this function is to fully reduce its argument and return `True` afterwards. Such an 'eval' function is usually used to express evaluation strategies in the context of parallelism [4, 17]. We use `eval` for expressing definedness conditions.

In the standard program library of Sparkle (`StdSparkle`), the function `eval` is defined by means of overloading. The instance on characters is defined by:

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

instance eval Char
where    eval :: !Char -> Bool
         eval x = True
```

Now, in a logical property, (`eval` x) can be used as termination condition. As is usual in proof assistants, this is equivalent to (`eval` $x =$ `True`). The meaning of this condition is as follows:

- If (it is known that) x can be successfully reduced to an arbitrary character, then `eval` x will produce `True` and the condition will be satisfied, since `True` = `True` is *True*.
- If (it is known that) x cannot successfully be reduced to a character, then `eval` x does not terminate and is equal to $\bot$ on the semantic level. Therefore, the condition is not satisfied, because $\bot =$ `True` is *False*.
- Note that `eval` is defined in such a way that `eval`x *never* reduces to `False`. So, all cases are covered in the previous reasoning.

The same principle can be used for lists, making use of overloading to assume the presence of 'eval' on the element type. This leads to the following definition:

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

This instance of `eval` fully evaluates both the list itself and all its elements. It can therefore be used to express the condition that a list must be fully evaluated. Below we give a few examples of the use of `eval` in properties of functions:

- $\forall_{n}[\texttt{eval}\ n\ \rightarrow n\ <\ n = \mathit{False}]$
- $\forall_{n,xs}[\texttt{eval}\ n \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs]$
- $\forall_{p,xs}[\texttt{eval}\ (\texttt{map}\ p\ xs) \rightarrow \texttt{takeWhile}\ p\ xs\ \texttt{++}\ \texttt{dropWhile}\ p\ xs = xs]$
- $\forall_{x,p,xs}[\texttt{eval}\ x\ \rightarrow \texttt{eval}\ xs \rightarrow \texttt{eval}\ (\texttt{map}\ p\ xs) \rightarrow$
  $\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = \texttt{isMember}\ x\ xs\ \texttt{\&\&}\ p\ x]$

The conditions in the examples of Section 3.1 can be expressed using 'eval'. The property of the first example is then expressed as follows (using `isMember` instead of the mathematical $\in$):

$$\forall_{f,g,xs}[\forall_{x}[\texttt{isMember}\ x\ xs \rightarrow \texttt{eval}(g\ x)] \rightarrow \texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)]$$

To express the definedness condition of the second example of Section 3.1 we need another variant of 'eval' that does not evaluates its argument fully but that evaluates only the 'spine' of the argument. This is given below.

Expressing Spine Evaluation and List Finiteness. Spine evaluation can be expressed easily by means of an 'eval' variant. However, if already an instance for full evaluation is given, then a new function must be defined since the type class system allows only one instance per type.

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

This same function `evalSpine` also expresses finiteness of lists, as when the spine of a list is fully evaluated, the list is evidently finite.

Some valid properties that are defined using `evalSpine`:

– $\forall_{xs}[\texttt{eval (length } xs) \rightarrow \texttt{evalSpine } xs]$
– $\forall_{xs}[\texttt{evalSpine } xs \rightarrow \texttt{evalSpine (reverse } xs)]$

The second example of Section 3.1 can now be reformulated to:

$$\forall_{xs}[\texttt{evalSpine } xs \rightarrow \texttt{reverse (reverse } xs) = xs]$$

Properties of 'eval'. All instances of the class 'eval' have to share certain properties. To prove properties of *all* members of a certain type classes, the recently added tool support for general type classes can be used [10]. With this tool, the following properties of 'eval' can be stated and proven in Sparkle.

– $\forall_x[x \neq \bot \rightarrow \texttt{eval } x]$
– $\forall_x[\texttt{eval } x \neq \texttt{False})]$

5 Related Work

In [5] Danielsson and Jansson perform a case study in program verification using partial and undefined values. They assume proof rules to be valid for the programming language. They do not use a formal semantics. We expect 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 [15] 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 the semantics used in this paper, analogue revisions will be required. As is further pointed out by Sestoft [15] the rules given by Launchbury are not *fully lazy.* Full laziness can be achieved by introducing new let-bindings for every maximal free expression [8].

An equivalent extension of Launchbury's semantics can be found in [2]. In this paper, a formal semantics for Glasgow Parallel Haskell is constructed on top of the standard Launchbury's semantics. Interestingly, not only parallellism is added, but enforced strictness in terms of a seq-construct as well. Furthermore, it is formally shown that this extension is sound. However, no properties are proven that are specific for the seq, such as the relation between 'lazy' and 'strict' terms.

It is possible to translate seq's to let!s (and vice versa) and shown properties can be compared directly.

Andrew Pitts [13] 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 [9] who use a denotational approach to present some "free" theorems in the presence of Haskell's seq.

At Chalmers University of Technology for the language Haskell a proof assistant Agda [1] has been developed in the context of the CoVer project. As with Sparkle the language is translated to a core-version on which the proofs are performed. Being geared towards facilitating the 'average' functional programmer Sparkle uses dedicated tactics and proof rules based on standard proof theory. Agda uses constructive type theory on λ-terms enabling independent proof checking. However, in contrast to Sparkle, Agda has no facilities to prove properties that are related to changed strictness properties.

Another project that aims to integrate programming, properties and validation is the Programatica project (www.cse.ogi.edu/PacSoft/projects/programatica) of the Pacific Software Research Center in Oregon. 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). In the Programatica project properties are mixed with the Haskell source. So, reasoning is bound to take place on the more complex syntactical source level instead of on a simpler core-language.

6 Conclusions / Future Work

The impact of changes in strictness properties on logical program properties is shown to be quite significant. It is illustrated how program properties can be adapted to reflect these changes. Furthermore, it is explained what the influence of explicit changes in strictness is on the semantics and on the reasoning steps.

We have shown that the special combination of several techniques, that have been made available in the proof assistant Sparkle to deal with definedness aspects, is well suited to assist the programmer in constructing the required proofs. We do not know of any other proof assistant with such a combined set of techniques to help dealing with these kinds of proofs.

Future work could be to study the relation of our approach to an approach which only aims to prove partial correctness.

References

1. A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying haskell programs using constructive type theory. In D. Leijen, editor, *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 62 – 74. Tallinn, Estonia, ACM Press, 2005.

2. 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.
3. 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.
4. G. L. Burn. Evaluation transformers a model for the parallel evolution of functional languages. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 446–470, London, UK, 1987. Springer-Verlag.
5. 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*, pages 85–109.
6. 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.
7. M. Dowse, A. Butterfield, and M. van Eekelen. Ta language for reasoning about concurrent functional i/o. In T. Arts and M. Mohnen, editors, *Selected Papers from the 16th International Workshop on Implementation and Application of Functional Languages, IFL 2004*, volume 3074 of *Lecture Notes in Computer Science*, pages 177–195, Lbeck, Germany, 2004. Springer Verlag.
8. 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.
9. P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 99–110, Venice, Italy, 2004.
10. Kesteren, R. van and Eekelen, M. van and Mol, M. de. *Proof Support for General Type Classes*. Intellect, 2004. to appear.
11. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.
12. 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.
13. 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.
14. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. 1993. ISBN 0-201-41663-8.
15. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
16. M. Tejfel, Z. Horváth, and T. Kozsik. Extending the sparkle core language with object abstraction. *Acta Cybernetica*, 17(2), 2005.
17. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. P. Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.