Making "Stricterness" More Relevant

Stefan Holdermans Jurriaan Hage

Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{stefan,jur}@cs.uu.nl

Abstract

Adapting a strictness analyser to have it take into account explicit strictness annotations can be a tricky business. Straightforward extensions of analyses based on relevance typing are likely to either be unsafe or fail to pick the fruits of increases in strictness that are introduced through annotations. We propose a more involved adaptation of relevance typing, that can be used to derive strictness analyses that are both safe and effective in the presence of explicit strictness annotations. The resulting type system provides a firm foundation for implementations of type-based strictness analysers in compilers for lazy programming languages such as Haskell and Clean.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, Type structure

General Terms Languages, Theory

Keywords lazy evaluation, strictness analysis, relevance typing, explicit strictness annotations

1. Introduction

In the design of strictness analyses, lazy functional languages are typically modelled in terms of nonstrict lambda-calculi. However, such calculi fail to account for the semantic properties of constructs such as Haskell's primitive function *seq* (Peyton Jones 2003) and Clean's strictness annotations (Plasmeijer and Van Eekelen 1998), that allow programmers to selectively make their functions stricter. As a result, it is not always clear how strictness analyses and the optimisations they enable scale to real-world programming languages.

In this context, this paper makes the following contributions:

- We show that two straightforward extensions of an optimising transformation that is driven by a strictness analysis based on relevance typing for the call-by-name lambda-calculus (Section 4) to a nonstrict language with a construct for selectively making programs stricter (Section 3) yield transformations that are either unsafe or otherwise ineffective, in the sense that they fail to pick up on any increases in strictness incurred from the use of this construct (Section 5).
- Moreover, we present a more involved adaptation of the transformation that makes use of a relevance typing discipline that, in addition to demand propagation, also keeps track of *applicativeness*: i.e., which expressions in a program are guaranteed to give rise to functions that, during the evaluation of the program, receive arguments (Section 6). The transformations obtained by this approach are both safe and effective, and so our approach can serve as a basis for optimisations in compilers for real-world languages.

PEPM'10, January 18–19, 2010, Madrid, Spain.

2. Background

The praises of lazy evaluation have been sung in many voices (e.g., Hughes 1989) and its benefits (the ability to construct infinite data structures, the ability to define custom control structures, the inherent avoidance of unnecessary computations, the facilitation of increased modularisation—to name a few) are well-known. Equally well-known are its drawbacks, most notably the excessive use of memory that is frequently associated with the deferral of computations. Accurate strictness analysers have therefore proven themselves indispensable tools in the optimisation of lazy programs.

The goal of strictness analysis is to identify which functions within a nonstrict program are in fact strict, i.e., always diverge on diverging input. Changing the evaluation strategy for applications of such functions from a nonstrict strategy (call-by-name or call-by-need) into a strict strategy (call-by-value) does not change the semantics of the program and, hence, avoids the cost that is otherwise incurred by deferring the evaluation of argument expressions. Still, as strictness analysers are necessarily conservative, they, in general, do not succeed in identifying all strict functions in a program. Moreover, in practice, a lot of functions written in lazily evaluated languages turn out to be *almost* strict but not quite. Obviously, in these situations, no help can be expected from strictness analysis. As a countermeasure, modern lazy languages like Haskell and Clean offer the programmer a means to make such programs stricter: Clean offers explicit strictness annotations in type signatures and strict local definitions, while Haskell provides a primitive function $seq :: \alpha \to \beta \to \beta$, that evaluates its first argument (that is, forces it to *weak-head normal form*) and returns its second.

As an example of the use of such constructs, consider the function *const* as found in Haskell's *Prelude*,

$$\begin{aligned} &const :: \alpha \to \beta \to \alpha \\ &const\ x\ y = x, \end{aligned}$$

and the following—stricter—version, that makes essential use of the primitive *seq*:

$$\begin{aligned} &const' :: \alpha \to \beta \to \alpha \\ &const'\ x\ y = seq\ y\ x. \end{aligned}$$

Whereas *const* is strict only in its first argument (and lazy in its second), $const'$ is strict in both its arguments.

Of particular interest is that the increase in strictness that is induced by explicit annotations propagates through function application. For instance, because of its use of the *seq*-annotated function *const′*, the otherwise lazy function *force*, defined by

$$\begin{array}{l} \mathit{force} :: \alpha \to () \\ \mathit{force}\ x = \mathit{const}'\ ()\ x \end{array}$$

and always producing the empty tuple $()$, becomes strict itself as well. Of course, one would hope that such "stricterness" would be picked up by automatic strictness analysers, so that calls to functions like *const′* and *force* could be evaluated eagerly rather than lazily. However, faithfully accounting for the effects of explicit annotations is a tricky business as these come with some peculiar semantic properties.

For one thing, Haskell's built-in *seq* can be used to tell the programs $\bot$ and $\lambda x \to \bot$ apart. For example, the expression $\mathit{seq}\ \bot\ ()$ diverges whereas $\mathit{seq}\ (\lambda x \to \bot)\ ()$ produces the empty tuple. Note that this distinction cannot be made in either the call-by-name or call-by-need lambda-calculus; yet studies into programming languages typically assume one or the other of these calculi as a model for nonstrict functional languages. Then, having *seq* (or a similar construct) in a language has profound repercussions. For example, in the presence of strictness annotations, polymorphic functions generally do not possess the parametricity property (Wadler 1989) and, so, optimisations such as short-cut deforestation (Gill et al. 1993), that are justified by parametricity, are no longer sound.

Unsurprisingly, similar issues arise when adapting a strictness analysis for a mere nonstrict lambda-calculus for use in a lazy language with a construct like *seq*.

3. Preliminaries

To be able to consider the problem from a formal angle, we introduce a small expression language that can be thought of as a tiny subset of Haskell. Assuming a countable infinite set of variable symbols ranged over by x, expressions are given by

$$e \quad ::= \quad () \mid x \mid \lambda x \to e_1 \mid e_1\ e_2 \mid e_1\ \$!\ e_2 \mid \bot$$

and include the nullary tuple constructor $()$, the ever-diverging constant $\bot$, variables x, and lambda-abstractions $\lambda x \to e_1$. Furthermore, we distinguish between lazy function applications $e_1\ e_2$ and eager function applications $e_1\ \$!\ e_2$. We adopt the Barendregt convention and assume that all bound variables are named distinctly from any free variables. As always, lambda-abstractions extend as far to the right as possible. Lazy function application is left-associative and binds stronger than (right-associative) eager application.

To ease the presentation in the sequel, we have chosen to include eager function application as a primitive rather than *seq*, but this choice is by no means essential as $\$!$ and *seq* can always be defined in terms of each other: assuming *seq* as a primitive, we have

$$\begin{array}{l} (\$!) :: (\alpha \to \beta) \to \alpha \to \beta \\ f\ \$!\ x = \mathit{seq}\ x\ (f\ x), \end{array}$$

whereas with $\$!$ wired in, we can write

$$\begin{array}{l} \mathit{seq} :: \alpha \to \beta \to \beta \\ \mathit{seq}\ x\ y = (\lambda\_ \to y)\ \$!\ x. \end{array}$$

A natural semantics for the expression language is given in Figure 1 by means of rules for deriving judgements of the form $e \Downarrow w$. That is, successful evaluation of an expression e yields a weak-head normal form w, given by

$$w \quad ::= \quad () \mid \lambda x \to e_1.$$

As reflected by the rules [*e-unit*] and [*e-abs*], tuples and abstractions are already in weak-head normal form. Lazy and eager function applications are evaluated under a call-by-name and a call-by-value strategy, respectively; in the rules [*e-lapp*] and [*e-eapp*] beta-substitution is denoted by $[\cdot \mapsto \cdot]\cdot$. We do not have a rule for evaluating $\bot$ and say that evaluation of an expression e diverges if there is no weak-head normal form w such that $e \Downarrow w$.

Evaluation $\boxed{e \Downarrow w}$

$$\frac{}{() \Downarrow ()}\ [\textit{e-unit}] \qquad \frac{}{\lambda x \to e_1 \Downarrow \lambda x \to e_1}\ [\textit{e-abs}]$$

$$\frac{e_1 \Downarrow \lambda x \to e_0 \quad [x \mapsto e_2]e_0 \Downarrow w}{e_1\ e_2 \Downarrow w}\ [\textit{e-lapp}] \qquad \frac{e_1 \Downarrow \lambda x \to e_0 \quad e_2 \Downarrow w_2 \quad [x \mapsto w_2]e_0 \Downarrow w}{e_1\ \$!\ e_2 \Downarrow w}\ [\textit{e-eapp}]$$

Figure 1. Natural semantics.

Typing $\boxed{\Gamma \vdash e :: \tau}$

$$\frac{}{\Gamma \vdash () :: ()}\ [\textit{t-unit}] \qquad \frac{}{\Gamma \vdash \bot :: \tau}\ [\textit{t-bot}]$$

$$\frac{}{\Gamma_1 +\!\!+ [x \mapsto \tau] +\!\!+ \Gamma_2 \vdash x :: \tau}\ [\textit{t-var}] \qquad \frac{\Gamma +\!\!+ [x \mapsto \tau_1] \vdash e_1 :: \tau_2}{\Gamma \vdash \lambda x \to e_1 :: \tau_1 \to \tau_2}\ [\textit{t-abs}]$$

$$\frac{\Gamma \vdash e_1 :: \tau_2 \to \tau \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash e_1\ e_2 :: \tau}\ [\textit{t-lapp}] \qquad \frac{\Gamma \vdash e_1 :: \tau_2 \to \tau \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash e_1\ \$!\ e_2 :: \tau}\ [\textit{t-eapp}]$$

Figure 2. The underlying type system.

The static semantics of the language is presented in Figure 2 in terms of type-assignment rules for deriving judgements $\Gamma \vdash e :: \tau$, expressing that, in the type environment Γ, the expression e can be assigned the type τ. Here, types are given by

$$\tau \quad ::= \quad () \mid \tau_1 \to \tau_2,$$

with $()$ the empty-tuple or unit type and $\tau_1 \to \tau_2$ the type of functions that take arguments of type τ_1 to results of type τ_2. Type environments are finite maps from variables to types. We write $[\,]$ for the empty environment, $[x \mapsto \tau]$ for the singleton environment that maps x to τ, and $\Gamma_1 +\!\!+ \Gamma_2$ for the environment that is obtained by taking the union of two environments, Γ_1 and Γ_2, with disjoint domains.

The assignment rules in Figure 2 are completely standard. Note in particular that lazy applications $e_1\ e_2$ and eager applications $e_1\ \$!\ e_2$ have the same static semantics, and that $\bot$ can be assigned any type. In the sequel, we are only concerned with well-typed expressions, i.e., expressions for which, in a given environment Γ, there is at least one type τ with $\Gamma \vdash e :: \tau$. The static semantics of Figure 2 is referred to as the *underlying type system*.

4. Elementary Relevance Typing

Let us now consider a simple approach to strictness analysis that makes use of a nonstandard standard type system for keeping track of *relevance*. Similar systems have been considered by Wright (1991) and Amtoft (1993); the presentation below is largely inspired by Walker (2005).

We will use the analysis to drive program transformations that safely replace lazy function applications by eager applications. Here, "safely" means "without changing the meaning of the program". For now, we will only deal with transformations of programs that themselves do not already contain eager function applications. In Sections 5 and 6, we will then admit eager applications in source programs as well and show that to deal with these adequately can actually be quite involved.

4.1 Annotated Types

A variable x is *relevant* to an expression e if any expression bound to x is guaranteed to be evaluated whenever e is evaluated. We say that an abstraction $\lambda x \to e_1$ is a *relevant abstraction* if its formal parameter x is relevant to its body e_1.

To distinguish between expressions that result in relevant abstractions and expressions that result in abstractions that may not be relevant, we introduce annotated types, given by

$$\widehat{\tau} \quad ::= \quad () \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2.$$

That is, an annotated type $\widehat{\tau}$ is either the type $()$ of empty tuples or a function type $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ where φ ranges over annotations, for which we have

$$\varphi \quad ::= \quad \mathsf{S} \mid \mathsf{L}.$$

In our analysis, the annotation S is used to designate the types of expressions that, when successfully evaluated to weak-head normal form, produce relevant abstractions, while L is used to annotate the types of expressions that may or may not produce relevant abstractions.

Note that relevance implies strictness: if a variable x is relevant to an expression e, then e is strict in x and relevant abstractions denote strict functions. Now, while S and L can thus be thought of as mnemonics for "strict" and "lazy", they can also be taken to stand for "small" and "large" as we will impose an order on annotations that is characterised by $\mathsf{S} \sqsubset \mathsf{L}$. The induced joins and meets are given by

$$\begin{array}{lll} \mathsf{S} \sqcup \varphi = \varphi & \text{and} & \mathsf{S} \sqcap \varphi = \mathsf{S} \\ \mathsf{L} \sqcup \varphi = \mathsf{L} & & \mathsf{L} \sqcap \varphi = \varphi. \end{array}$$

4.2 Type-driven Call-by-value Transformation

Our goal will now be to transform programs by turning as many lazy applications of strict functions into eager applications. That is, if for a given lazy application $e_1\ e_2$ it can be shown that succesfully evaluating e_1 will always result in a relevant abstraction, then we optimise away the assumed overhead of nonstrict evaluation by replacing the application by its eager counterpart $e_1\ \$!\ e_2$. This optimisation is justified by the observation that from the implied strictness of the function e_1 it follows that, for the application to produce a result, evaluation of the argument e_2 is required anyway.

We define the transformation through rules for deriving judgements of the form

$$\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{\varphi},$$

expressing that, in the annotated type environment $\widehat{\Gamma}$, the source expression e, of type $\widehat{\tau}$ and annotated with φ, can be safely transformed into the target expression e'. Here, the annotation φ is used to indicate whether or not the context in which e appears guarantees its evaluation: e is said to be *demanded* if $\varphi = \mathsf{S}$.

Annotated type environments map variables x to pairs $\widehat{\tau}^{\varphi}$ consisting of an annotated type $\widehat{\tau}$ and an annotation φ. Analogously to the unannotated environments from Section 3, we write $[\,]$ for the empty environment, $[x \mapsto \widehat{\tau}^{\varphi}]$ for the singleton environment that maps x to $\widehat{\tau}^{\varphi}$, and $\widehat{\Gamma}_1 +\!\!+ \widehat{\Gamma}_2$ for the environment that is obtained by taking the union of two environments, $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$, with disjoint domains.

The rules of the transformation relation are given in Figure 3. Relevance typing constitutes a so-called *substructural typing discipline* (Walker 2005), which is reflected by a very careful treatment of type environments throughout the rules. The all-important invariant that we maintain is that any S-annotated variable in an annotated type environment $\widehat{\Gamma}$ is to appear in an S-context at least once. The rules [*r-unit*] and [*r-bot*], for instance, require that the constants $()$ and $\bot$ are transformed in an empty type environment. Both can be typed and transformed in any context; tuples always have the type $()$, while $\bot$ can be assigned any type. Variables, as per rule [*r-var*], are typed and transformed in singleton environments that agree with the types and annotations assigned.

Transformation $\quad \boxed{\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{\varphi}}$

$$\frac{}{[\,] \vdash () \triangleright () :: ()^{\varphi}}\ [\textit{r-unit}] \qquad \frac{}{[\,] \vdash \bot \triangleright \bot :: \widehat{\tau}^{\varphi}}\ [\textit{r-bot}]$$

$$\frac{}{[x \mapsto \widehat{\tau}^{\varphi}] \vdash x \triangleright x :: \widehat{\tau}^{\varphi}}\ [\textit{r-var}]$$

$$\frac{\varphi \sqsubseteq \widehat{\Gamma} \quad \widehat{\Gamma} +\!\!+ [x \mapsto \widehat{\tau}_1^{\varphi_1}] \vdash e_1 \triangleright e_1' :: \widehat{\tau}_2^{\mathsf{S}}}{\widehat{\Gamma} \vdash \lambda x \to e_1 \triangleright \lambda x \to e_1' :: (\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2)^{\varphi}}\ [\textit{r-abs}]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 \triangleright e_1' :: (\widehat{\tau}_2 \xrightarrow{\mathsf{S}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash e_2 \triangleright e_2' :: \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\ e_2 \triangleright e_1'\ \$!\ e_2' :: \widehat{\tau}^{\varphi}}\ [\textit{r-lapp}_1]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 \triangleright e_1' :: (\widehat{\tau}_2 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash e_2 \triangleright e_2' :: \widehat{\tau}_2^{\mathsf{L}}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\ e_2 \triangleright e_1'\ e_2' :: \widehat{\tau}^{\varphi}}\ [\textit{r-lapp}_2]$$

$$\frac{\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{\mathsf{L}}}{\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{\mathsf{S}}}\ [\textit{r-sub}] \qquad \frac{\widehat{\Gamma}_1 +\!\!+ \widehat{\Gamma}_2 \vdash e \triangleright e' :: \widehat{\tau}^{\varphi}}{\widehat{\Gamma}_1 +\!\!+ [x \mapsto \widehat{\tau}_0^{\mathsf{L}}] +\!\!+ \widehat{\Gamma}_2 \vdash e \triangleright e' :: \widehat{\tau}^{\varphi}}\ [\textit{r-weak}]$$

Figure 3. Relevance typing and call-by-value transformation.

Interesting is the rule for lambda-abstractions, [*r-abs*]. It states that typing and transforming an abstraction $\lambda x \to e_1$ depends on the typing and transformation of its body e_1 in a type enviroment that is extended with a binding for the formal parameter x. As far as deriving the argument and result types $\widehat{\tau}_1$ and $\widehat{\tau}_2$ is concerned, the rule is completely standard; what remains is to consider how annotations are dealt with. To determine whether the parameter x is relevant to the body e_1—and thus whether the abstraction itself is relevant—we "reset" the demand context for e_1 to S. If x can then be annotated with S as well, we conclude that $\lambda x \to e_1$ is relevant; otherwise, we classify the abstraction as possibly irrelevant. To prevent that resetting the demand for the body propagates to any variables other than the formal parameter, we require that none of the bindings in the original type environment $\widehat{\Gamma}$ carries an annotation that is smaller than the demand φ of the abstraction. Hence, the rule for abstractions includes the so-called *containment restriction* $\varphi \sqsubseteq \widehat{\Gamma}$, for which we have the following rules:

$$\frac{}{\mathsf{S} \sqsubseteq \widehat{\Gamma}} \qquad \frac{}{\mathsf{L} \sqsubseteq [\,]} \qquad \frac{}{\mathsf{L} \sqsubseteq [x \mapsto \widehat{\tau}^{\mathsf{L}}]} \qquad \frac{\mathsf{L} \sqsubseteq \widehat{\Gamma}_1 \quad \mathsf{L} \sqsubseteq \widehat{\Gamma}_2}{\mathsf{L} \sqsubseteq (\widehat{\Gamma}_1 +\!\!+ \widehat{\Gamma}_2)}\ .$$

There are two rules that deal with lazy function applications $e_1\ e_2$. The first, [*r-lapp*$_1$], is applicable whenever the function expression e_1 can be assigned an S-annotated function type and thus constitutes a strict function. Then, the argument e_2 is demanded whenever the result of the application is and, hence, the demand φ of the application propagates to e_2. Preeminently, here we seize on the opportunity and transform the lazy application $e_1\ e_2$ into the eager application $e_1'\ \$!\ e_2'$. The second rule, [*r-lapp*$_2$], deals with the application $e_1\ e_2$ of possibly nonstrict functions e_1. In that case, there are no guarantees about the demand for the argument and so e_2 receives the demand L. Both rules, of course, require that the type $\widehat{\tau}_2$ of the argument matches the argument type of the function and that the type $\widehat{\tau}$ of the application equals the result type of the function. Moreover, both rules insist that the environment in which the application is analysed corresponds to the pointwise meet $\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2$ of the environments $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$ in which, respectively, the subexpressions e_1 and e_2 are analysed:

$$
\begin{aligned}
[\,] \sqcap [\,] &= [\,] \\
[x \mapsto \widehat{\tau}^{\varphi_1}] \sqcap [x \mapsto \widehat{\tau}^{\varphi_2}] &= [x \mapsto \widehat{\tau}^{\varphi_1 \sqcap \varphi_2}] \\
(\widehat{\Gamma}_{11} +\!\!+ \widehat{\Gamma}_{12}) \sqcap (\widehat{\Gamma}_{21} +\!\!+ \widehat{\Gamma}_{22}) &= (\widehat{\Gamma}_{11} \sqcap \widehat{\Gamma}_{21}) +\!\!+ (\widehat{\Gamma}_{12} \sqcap \widehat{\Gamma}_{22}).
\end{aligned}
$$

This "context split" (Cervesato and Pfenning 2002) reflects that a variable is relevant to $e_1\ e_2$ if its relevance can be established in at least one of the subanalyses for e_1 and e_2.

The rule [*r-sub*] implements *subeffecting* (cf., e.g., Talpin and Jouvelot 1992) and, by enabling derivations to selectively "forget" about the demand of an expression, allows for more programs to be considered well-typed and hence transformable. Intuitively, it states that any conclusions that may be drawn from the assumption that an expression is not demanded are still valid if the expression is in fact demanded.

The weakening rule [*r-weak*], finally, expresses that any transformation that is derivable in a given annotated type environment $\widehat{\Gamma}$ that does not contain a binding for a variable x can also be derived in an annotated type environment that is obtained from adding an L-annotated binding for x to $\widehat{\Gamma}$.

4.3 Examples

Before we discuss how to formulate the correctness of the transformation from Figure 3, let us consider two examples of derivations in our system. First, consider the expression

$$(\lambda x \to \lambda y \to x)\ ()\ ().$$

A transformation derivation for this expression is given in Figure 4. There, the abstraction $\lambda x \to \lambda y \to x$ is assigned the relevance type $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()$, reflecting that the function it produces is strict in its first argument. Hence, the innermost application can be performed eagerly rather than lazily, resulting in the target term

$$((\lambda x \to \lambda y \to x)\,\$!\,())\ ().$$

Then, as an example of a derivation in which the use of subeffecting is crucial, consider the typing of the term

$$\lambda x \to (\lambda y \to ())\ (\lambda z \to x)$$

in Figure 5. As the abstraction $\lambda y \to ()$ results in a lazy function, the function argument $\lambda z \to x$ has to be assigned an L-annotated type. The containment restriction for $\lambda z \to x$ then prescribes that x, in the type environment for the abstraction, is also L-annotated. However, as the body of the abstraction, like the bodies of all abstractions, is to be assigned an S-annotated type, we have to somehow be able to derive the judgement $[x \mapsto ()^{\mathsf{L}}] \vdash x \triangleright x :: ()^{\mathsf{S}}$. As attested to in the upper-right corner of Figure 5, subeffecting allows us to obtain the desired judgement from the trivially fulfilled premise $[x \mapsto ()^{\mathsf{L}}] \vdash x \triangleright x :: ()^{\mathsf{L}}$.

4.4 Semantic Correctness

To be able to demonstrate that the transformations derivable in our system are indeed safe, i.e., do not change the meanings of their source programs, we define an ordering $\lesssim$ on expressions:

$$\frac{}{e \lesssim e} \qquad \frac{e_1 \lesssim e_1'}{\lambda x \to e_1 \lesssim \lambda x \to e_1'}$$

$$\frac{e_1 \lesssim e_1' \quad e_2 \lesssim e_2'}{e_1\ e_2 \lesssim e_1'\ e_2'} \qquad \frac{e_1 \lesssim e_1' \quad e_2 \lesssim e_2'}{e_1\,\$!\,e_2 \lesssim e_1'\ e_2'} \qquad \frac{e_1 \lesssim e_1' \quad e_2 \lesssim e_2'}{e_1\,\$!\,e_2 \lesssim e_1'\,\$!\,e_2'}\ .$$

That is, $e \lesssim e'$ if e can be derived from e' merely by replacing lazy applications by eager applications and thus, intuitively, e is as least as strict than e'.

The safety of the transformation is now captured by the following result:

Theorem 1 (Semantic Soundness). If $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{\varphi}$, then $e' \lesssim e$. Furthermore, if $e \Downarrow w$, then there is a w' such that $e' \Downarrow w'$ and $w' \lesssim w$. ∎

In particular, if evaluating the source program e does not diverge, then neither does evaluating the target program e'.

5. Dealing with Eager Applications

The call-by-value transformation in the previous section has a somewhat limited scope in that it does not support the transformation of expressions that include eager applications: in our relevance type system of Section 4.2 such expressions are simply considered ill-typed. As a result, the source language for the transformation does not quite model real-world languages such as Haskell and Clean. In this section, we will try to overcome this limitation by admitting eager applications in source terms. Naturally, when doing so, we want to make sure that transformation remains safe. Moreover, we would want the transformation to be effective, in the sense that it is able to pick up on the increase in strictness that is induced by eager applications and have the resulting stricterness propagate.

At first glance, the task at hand seems as simple as adding one or two appropriate rules for eager applications to the type system: below, we will describe two of the most straightforward approaches. However, as it turns out, these are either safe but ineffective (Section 5.1) or effective but unsafe (Section 5.2).

5.1 A Conservative Approach

Arguably the most simple way to extend the transformation with support for eager applications is to treat eager applications as if they were lazy applications. That is, we add two rules,

$$\frac{\widehat{\Gamma}_1 \vdash e_1 \triangleright e_1' :: (\widehat{\tau}_2 \xrightarrow{\mathsf{S}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash e_2 \triangleright e_2' :: \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\,\$!\,e_2 \triangleright e_1'\,\$!\,e_2' :: \widehat{\tau}^{\varphi}}$$

and

$$\frac{\widehat{\Gamma}_1 \vdash e_1 \triangleright e_1' :: (\widehat{\tau}_2 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash e_2 \triangleright e_2' :: \widehat{\tau}_2^{\mathsf{L}}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\,\$!\,e_2 \triangleright e_1'\,\$!\,e_2' :: \widehat{\tau}^{\varphi}}\ ,$$

that differ from the rules [*r-lapp*$_1$] and [*r-lapp*$_2$] in nothing but their mentioning of eager applications rather than lazy applications.

However, while these additions do bring eager applications in scope of the transformation and perserve its safety, they do not enable us to profit from any increase in strictness caused by the use of \$!. For example, the rules cannot derive the relevance of the following abstraction (cf. the function *force* from Section 2),

$$\lambda x \to (\lambda y \to \lambda z \to y)\ ()\,\$!\,x, \tag{1}$$

and, hence, lazy applications of this abstraction are unaffected by associated call-by-value transformations—even though it would be completely safe to have them replaced by eager applications.

5.2 A More Ambitious Attempt

To be able to have \$!-induced increases in strictness propagate, we have to record that an eager application always evaluates its argument in order to produce a result: in other words, that the argument of an eager application is demanded whenever the result of the application is. This can be easily expressed by means of a single transformation rule for eager application:

$$\frac{\widehat{\Gamma}_1 \vdash e_1 \triangleright e_1' :: (\widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash e_2 \triangleright e_2' :: \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\,\$!\,e_2 \triangleright e_1'\,\$!\,e_2' :: \widehat{\tau}^{\varphi}}\ .$$

Note how this rule resembles the rule for lazy applications of strict functions, [*r-lapp*$_1$], with the notable exception that the actual

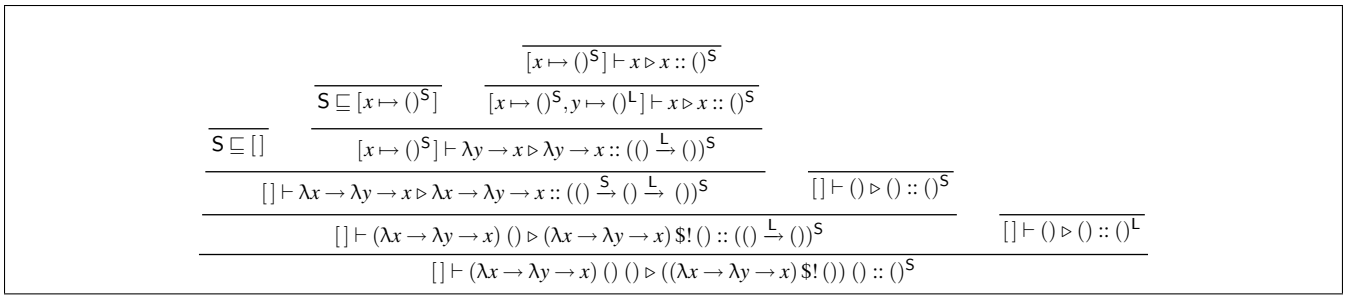

Figure 4. A derivation for $[\,] \vdash (\lambda x \to \lambda y \to x)\ ()\ ()\ \triangleright\ ((\lambda x \to \lambda y \to x)\ \$!\ ())\ ()\ ::\ ()^{\mathsf{S}}$.

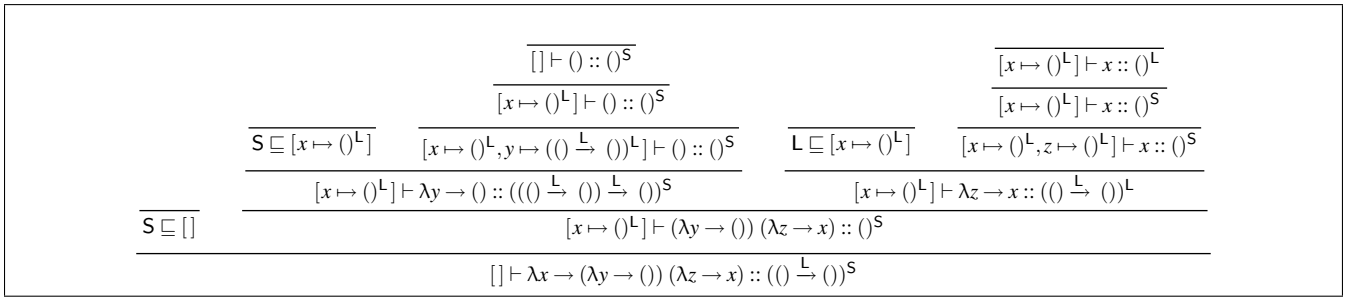

Figure 5. A derivation that involves subeffecting. *(Target terms omitted for clarity.)*

relevance φ_0 of the function expression e_1 is completely ignored here.

This new rule seems to capture the semantics of eager application much better than the rules that we considered in the previous subsection and, indeed, we are now able to derive that abstractions like expression (1) above are relevant, effectively enabling local increases in strictness to propagate. Unfortunately, and perhaps surprisingly, addition of the rule renders the transformation system unsafe! To see this, consider the abstraction

$$\lambda x \to (\lambda y \to ())\ \$!\ (\lambda z \to x) \tag{2}$$

and note that it is lazy in its argument x; in particular, that the application

$$(\lambda x \to (\lambda y \to ())\ \$!\ (\lambda z \to x))\ \bot \tag{3}$$

evaluates to the empty tuple $()$, while evaluation of the stricter

$$(\lambda x \to (\lambda y \to ())\ \$!\ (\lambda z \to x))\ \$!\ \bot \tag{4}$$

obviously diverges. Still, with the suggested rule for eager applications we can, as demonstrated in Figure 6, wrongfully derive that expression (2) is a relevant abstraction. As a result, it allows expression (3) to be transformed into expression (4)—which is clearly unsafe.

Now, with such transformations admitted, there is of course no hope for establishing a correctness result in the spirit of Theorem 1. However, rather than completely abandoning the proposed approach to dealing with eager applications, we will, in the next section, analyse what exactly prohibits the resulting transformation from being safe and show what can be done to compensate. This will lead us to an extension of the original system that is more involved than the extensions considered in the present section, but that is also both safe and effective.

6. A Refined Approach to Relevance Typing

In the previous section, we have seen that safely extending the transformation system of Section 4 with support for eager function applications in the source language does not require much effort in itself (Section 5.1); but also that our desire to have the system reflect the semantics of these eager applications apparently complicates matters considerably (Section 5.2). And, indeed, there is no such thing as a free lunch: in this section, we show how transformations that are both safe and faithful to the nature of eager applications can be realised at the cost of extending the system in more essential ways than those that we have considerd before. Interestingly, the key to success lies in establishing where the approach of Section 5.2 went wrong.

6.1 Stocktaking

Let us have a closer look at how the derivation tree for expression (2) in Figure 6 enabled us to inappropriately conclude that the given abstraction was relevant. To do so, it had to provide "evidence" that the formal parameter x of the abstraction was relevant to its body. Since the body is an application, this can be achieved by deriving that x is relevant to at least one of the two subexpressions of the application. Clearly, x cannot be relevant to the function expression $\lambda y \to ()$ as it does not even occur in it. The only possibility left is then to establish that x is relevant to the argument abstraction $\lambda z \to x$. Now, note that the body of this abstraction is, as is per [*r-abs*] the case for all abstractions, to be analysed as if it were demanded. Then, from there, because the body reads just x, we can conclude that x is at least locally relevant. As x is a free variable of $\lambda z \to x$, the containment restriction prescribes that this conclusion can only be propagated globally if the abstraction $\lambda z \to x$ is itself demanded. However, as the demand for $\lambda z \to x$ is implied by its use as an argument to a demanded eager abstraction (!), the containment restriction is met trivially and so we indeed derive that x is relevant to $(\lambda y \to ())\ \$!\ (\lambda z \to x)$.

Now—where did we go wrong? A moment's reflection reveals that, with a progressive rule for eager applications in place, the containment restriction does not serve its purpose anymore. It was there to ensure that we could not conclude that an expression bound to a free variable of an abstraction was demanded while, in fact, it was not. Still, that is exactly what happened for the free variable x of $\lambda z \to x$ in the derivation in Figure 6. So, why is the containment

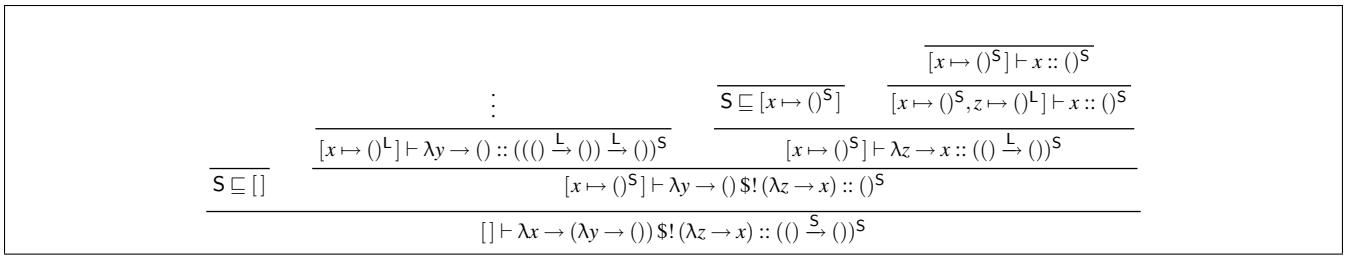

Figure 6. An unsafe derivation. *(Target terms omitted for clarity.)*

restriction adequate in the simple setting of Section 4, but not in the more realistic setting of Section 5.2?

The answer lies in the observation that, if we leave eager applications out of the source language, an abstraction can only ever be considered demanded if it can be guaranteed to appear in the function position of an application at least once during evaluation. In simpler terms: the only way to force the evaluation of a function is to apply it to an argument. Thus, if the evaluation of a function is forced, then it is applied to an argument and if it is applied to an argument, then its body is evaluated. Hence, the demands for an abstraction and its body coincide, and this invariant is exploited by the containment restriction: if an abstraction is demanded, then so are all variables that are relevant to its body.

Note that the coinciding of demands for an abstraction and its body is reflected by the inability of nonstrict lambda-calculi to tell $\bot$ and $\lambda x \to \bot$ apart. But, as we demonstrated in Section 2, with constructs like *seq* and \$!, we can actually tell $\bot$ and $\lambda x \to \bot$ apart! Indeed, with eager applications in the source language, we can force the evaluation of an abstraction without ever evaluating its body and this is what happens to $\lambda z \to x$ in the expression $(\lambda y \to ())\,\$!\,(\lambda z \to x)$.

These considerations are suggestive of replacing the containment restriction by either a stronger constraint that does not ever allow free variables to be relevant to an abstraction or a more refined constraint that takes into account why abstractions are demanded. The first of these options is easy to realise but obviously gives rise to transformation that are even less effective than the conservative transformations of Section 5.1, as it would prevent us from detecting the strictness of curried multiargument functions in any but the first of their arguments. Below, we shall therefore proceed along the path of the second option and extend our relevance typing discipline with a facility for recording which functions are guaranteed to be applied to arguments.

6.2 Type-driven Call-by-value Transformation (Revised)

In the transformation system of Section 4, we used annotations S and L to keep track of whether or not expressions were demanded by their contexts. We will now refine the transformation system and have these annotations also indicate the *applicativeness* of expressions. We say that an expression is applicative if it is guaranteed to be applied to an argument at least once. We will repeat the essential parts of the development of Section 4 for our refined system and call attention to any differences with respect to the orginal system. In the definitions that follow, changes and additions with respect to the original definitions are typeset against a shaded background.

Let us start with the annotations. Since we are using the same set of annotations $\{\mathsf{S},\mathsf{L}\}$ to express both demand and applicativeness, we commit to the convention that annotations are ranged over by the metavariable φ if they are used to indicate demand and by the metavariable ψ if they are used to indicate applicativeness:

$$\varphi, \psi \quad ::= \quad \mathsf{S} \mid \mathsf{L}.$$

When expressing applicativeness, the smaller annotation S is to be read as "guaranteed to be applied to an argument" and the larger annotation L as "may not be applied to an argument".

As far as annotated types are concerned, function types are decorated with information about the applicativeness of the arguments and results of functions:

$$\widehat{\tau} \quad ::= \quad () \mid \widehat{\tau}_1^{\,\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2^{\,\psi_2}.$$

Furthermore, annotated type environments $\widehat{\Gamma}$ now map from variables x to triples $\widehat{\tau}^{(\varphi,\psi)}$, consisting of an annotated type $\widehat{\tau}$, a demand annotation φ, and an annotation ψ that reflects the applicativeness of any expressions bound to x.

The judgements of the refined transformation relation read

$$\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)},$$

where the annotation ψ indicates the applicativeness of the transformed expression e. Note that applicativeness implies relevance: if an expression can be guaranteed to be applied to an argument, it can also be guaranteed to be evaluated. Therefore, in our refined transformation system, we maintain the invariant that whenever we have that $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$, it holds that $\varphi \sqsubseteq \psi$.

The rules for deriving jugdements of the given form are listed in Figure 7. The rule for empty tuples, [*r-unit*], expresses that tuples can never appear in contexts in which they are applied to arguments, while rule [*r-bot*] expresses that whether an occurrence of $\bot$ is applicative depends on its context. In rule [*r-var*] the applicativeness of variables is obtained from the environment.

Crucially, in the rule for abstractions, [*r-abs*], the containment restriction is dominated by applicativeness rather than demand: the body of an abstraction is demanded only if the abstraction itself is guaranteed to be applied to an argument. One of the rules for deriving containment needs to be updated as well, but only to reflect that type environments now also contain annotations for applicativeness:

$$\frac{}{\mathsf{S} \sqsubseteq \widehat{\Gamma}} \qquad \frac{}{\mathsf{L} \sqsubseteq []} \qquad \frac{}{\mathsf{L} \sqsubseteq [x \mapsto \widehat{\tau}^{(\mathsf{L},\mathsf{L})}]} \qquad \frac{\mathsf{L} \sqsubseteq \widehat{\Gamma}_1 \quad \mathsf{L} \sqsubseteq \widehat{\Gamma}_2}{\mathsf{L} \sqsubseteq (\widehat{\Gamma}_1 +\!\!+ \widehat{\Gamma}_2)}\,.$$

The refined transformation system has three rules for applications, which all make use of a revised context-split operation:

$$\begin{array}{rl} [] \sqcap [] & = [] \\ [x \mapsto \widehat{\tau}^{(\varphi_1,\psi_1)}] \sqcap [x \mapsto \widehat{\tau}^{(\varphi_2,\psi_2)}] & = [x \mapsto \widehat{\tau}^{(\varphi_1 \sqcap \varphi_2,\, \psi_1 \sqcap \psi_2)}] \\ (\widehat{\Gamma}_{11} +\!\!+ \widehat{\Gamma}_{12}) \sqcap (\widehat{\Gamma}_{21} +\!\!+ \widehat{\Gamma}_{22}) & = (\widehat{\Gamma}_{11} \sqcap \widehat{\Gamma}_{21}) +\!\!+ (\widehat{\Gamma}_{12} \sqcap \widehat{\Gamma}_{22}). \end{array}$$

In each of the rules [*r-lapp*$_1$], [*r-lapp*$_2$], and [*r-eapp*], the applicativeness of the function expression e_1 follows from the demand for the application. In the two rules for lazy applications, the applicativeness of the argument expression e_2 is obtained by taking the join of the demand for the application and the applicativeness of arguments of e_1. In the case for eager applications, we ignore, as we did in Section 5.2, the strictness of the function expression e_1 and propagate the demand for the application directly to the argument

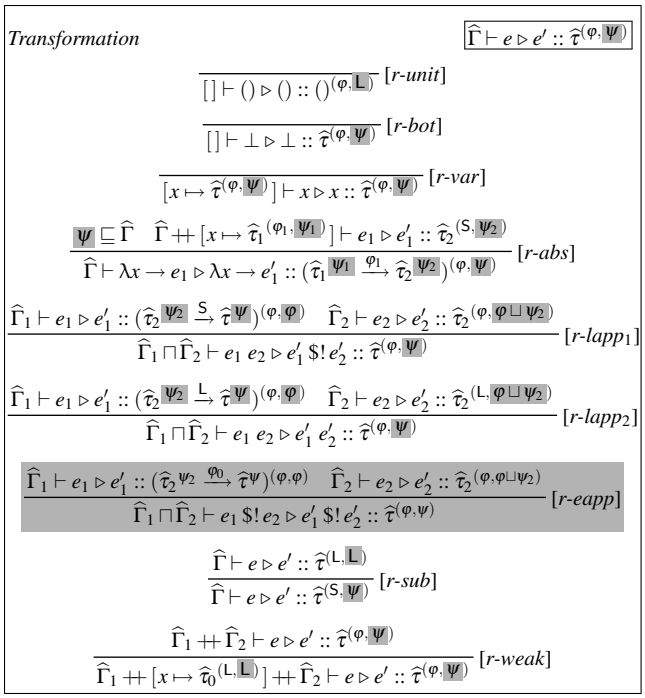

Figure 7. Relevance typing and call-by-value transformation. *(Cf. Figure 3.)*

expression e_2; furthermore, the applicativeness of e_2 is obtained by combining the demand φ for the application and the applicativeness ψ_2 of the parameter of e_1.

As expressed by rule [*r-sub*], subeffecting applies to applicativeness as well as to demand. Similarly, as far as weakening is concerned, by rule [*r-weak*], no differences arise between the annotations for applicativeness and those for demand.

6.3 Examples

As an example of how our revised transformation system is indeed faithful to the semantics of eager applications, consider the analysis of expression (1),

$$\lambda x \to (\lambda y \to \lambda z \to y)\ ()\ \$!\ x,$$

from Section 5.1, shown in Figure 8. Even though the subexpression $(\lambda y \to \lambda z \to y)\ ()$ produces a lazy function, reflected by its relevance type $()^{\mathsf{L}} \xrightarrow{\mathsf{L}} ()^{\mathsf{L}}$, the argument x of the eager application $(\lambda y \to \lambda z \to y)\ ()\ \$!\ x$ is still assigned the relevance annotation S and so the expression as a whole can be recognised as producting a strict function, i.e., a function of relevance type $()^{\mathsf{L}} \xrightarrow{\mathsf{S}} ()^{\mathsf{L}}$.

As an illustration of how faithfulness to the semantics of eager applications does not come at the expense of an unsound analysis, reconsider expression (2),

$$\lambda x \to (\lambda y \to ())\ \$!\ (\lambda z \to x),$$

for which an unsafe derivation was given in Figure 6. As the derivation in Figure 9 shows, the inner abstraction $\lambda z \to x$ does not appear in an applicative position and, hence, the revised containment restriction for this abstraction now forces all bindings in its type environment to be annotated with L exclusively. In particular, the variable x can no longer be considered relevant to the body of the outer abstraction $\lambda x \to (\lambda y \to ())\ \$!\ (\lambda z \to x)$ and so we can no longer derive that the function produced by this abstraction is strict.

6.4 Properties

The transformations that arise from our revised system—and that are thus faithful to the semantics of eager applications—can be shown to preserve the meaning of their source programs in a manner similar to that of Section 4:

Theorem 2 (Semantic Soundness, cf. Theorem 1). If $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$, then $e' \lesssim e$. Furthermore, if $e \Downarrow w$, then there is a w' such that $e' \Downarrow w'$ and $w' \lesssim w$. ∎

Moreover, in contrast to the system from Section 4, call-by-value transformation is now applicable to all well-typed terms. To demonstrate this, let us write $\lfloor \widehat{\tau} \rfloor$ for the underlying type that is obtained by removing all annotations from the annotated type $\widehat{\tau}$ and $\lfloor \widehat{\Gamma} \rfloor$ for the underlying type environment that is obtained by removing all annotations from the annotated type environment $\widehat{\Gamma}$. Then, we have

Theorem 3 (Conservative Extension).

1. If $\Gamma \vdash e :: \tau$, then there exist $\widehat{\Gamma}$, e', $\widehat{\tau}$, φ, and ψ with $\lfloor \widehat{\Gamma} \rfloor = \Gamma$ and $\lfloor \widehat{\tau} \rfloor = \tau$, such that $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$.
2. If $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$, then $\lfloor \Gamma \rfloor \vdash e :: \lfloor \widehat{\tau} \rfloor$ and $\lfloor \widehat{\Gamma} \rfloor \vdash e' :: \lfloor \widehat{\tau} \rfloor$. ∎

Finally, as functions that are demanded without ever being applied to arguments do not appear in real-world functional programs very often, the refined transformation system is expected, in practice, to be as effective as the naïve system that was proposed in Section 5.2.

7. Algorithmic Relevance Typing

The main challenge in the design of an algorithm for our transformation system is to deal with the nondeterministic context-split operation and the non-syntax-directed typing rules for subeffecting and weakening. To address this challenge, we restructure the transformation rules from Figure 7 to obtain a syntax-directed system that is equivalent to the transformation of Section 6 and from which an algorithm can be read off.

The syntax-directed system is presented in Figure 10, as a set of rules for deriving judgements of the form

$$\widehat{\Gamma}\ ;\ \varphi\ ;\ \psi \vdash e \triangleright e' :: \widehat{\tau}\ ;\ \widehat{\Gamma}'.$$

Here, $\widehat{\Gamma}$ is an input environment, φ an annotation that expresses the demand for the expression under analysis e, and ψ an annotation that indicates whether or not e is guaranteed to be applied to an argument. Then, e' is a semantics-preserving transformation of e, $\widehat{\tau}$ an annotated type assignable to e, and $\widehat{\Gamma}'$ an output environment. The idea is that $\widehat{\Gamma}'$ is an updated version of $\widehat{\Gamma}$, to which information on the relevance and applicativeness of variables in $\widehat{\Gamma}$ that is collected throughout the analysis of e is added.

The rules [*s-unit*] and [*s-bot*] for $()$ and $\bot$, respectively, are straightforward. In particular, these rules do not update the input environment $\widehat{\Gamma}$.

The rule for variables, [*s-var*], takes into account the context, given by φ and ψ, in which a variable x appears and updates the input environment $\widehat{\Gamma}$ accordingly by taking the meet of φ and ψ with the annotations φ_0 and ψ_0 for x from $\widehat{\Gamma}$. That way, if x appears in a relevant (or applicative) position in its containing expression, it is indeed recorded as relevant (or applicative) in the output environment.

In the rule for lambda-abstractions, [*s-abs*], we extend the input environment $\widehat{\Gamma}$ with a binding for the formal parameter x. This binding is pessimistically annotated with the pair (L,L) before it is passed on as part of the input enviroment for the body e_1 of the abstraction. As analysis of e_1 may update the binding, we inspect the

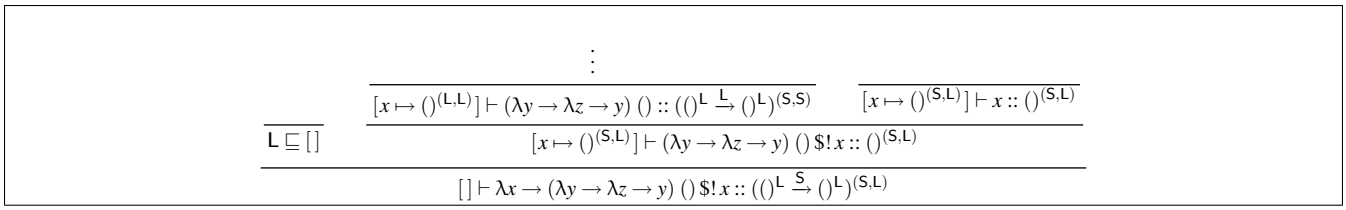

Figure 8. A derivation in which "stricterness" is propagated. *(Target terms omitted for clarity.)*

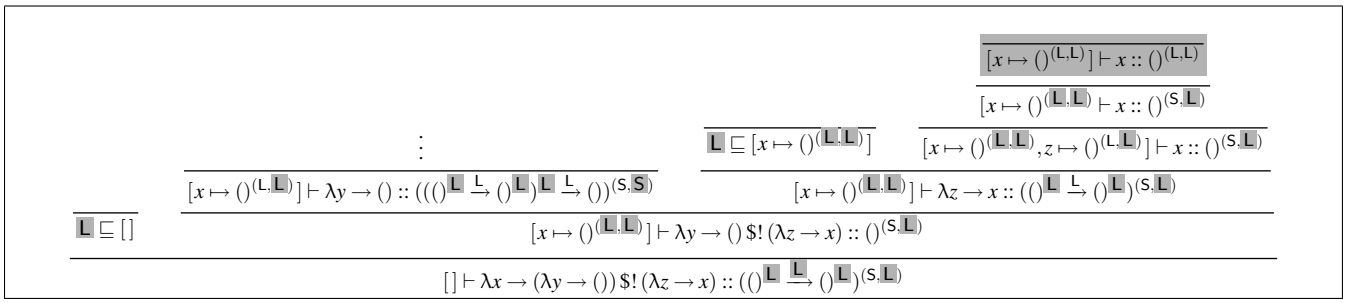

Figure 9. A deriviation that makes essential use of applicativeness. *(Cf. Figure 6; target terms omitted for clarity.)*

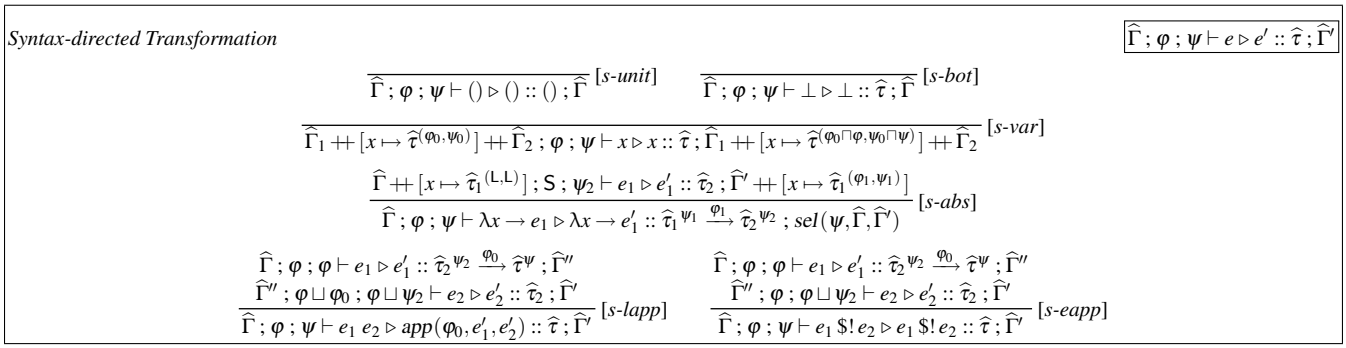

Figure 10. Syntax-directed relevance typing and call-by-value transformation.

output enviroment for e_1 to retrieve the relevance φ_1 of the abstraction and the applicativeness ψ_1 of the parameter. The containment restriction is implemented by means of a metaoperation *sel*,

$$\begin{aligned} \mathit{sel}(\mathsf{S},\widehat{\Gamma},\widehat{\Gamma}') &= \widehat{\Gamma}' \\ \mathit{sel}(\mathsf{L},\widehat{\Gamma},\widehat{\Gamma}') &= \widehat{\Gamma}, \end{aligned}$$

that, depending on the applicativeness ψ of the abstraction, determines whether or not updates to the input environment $\widehat{\Gamma}$ are propagated.

The rule [*s-lapp*] for lazy applications, $e_1\ e_2$, forwards its input environment $\widehat{\Gamma}$ to the transformation of the function expression e_1. The environment $\widehat{\Gamma}''$ that is produced as an output of this transformation is, in turn, used as the input environment for the argument e_2. The relevance and applicativeness of the argument depend on both the relevance φ of the application as a whole and the strictness φ_0 and applicativeness ψ_2 of the function. The strictness φ_0 of the function is also used by the metaoperation *app*,

$$\begin{aligned} \mathit{app}(\mathsf{S},e_1,e_2) &= e_1\,\$!\,e_2 \\ \mathit{app}(\mathsf{L},e_1,e_2) &= e_1\ e_2, \end{aligned}$$

that determines whether or not the application is transformed into an eager application. The rule [*s-eapp*] for eager applications $e_1\,\$!\ e_2$ is identical, expect, of course, that the strictness of e_1 has no influence on the relevance of e_2 and the eagerness of the application as a whole is always maintained.

The equivalence of the syntax-directed system and the system from Section 6 follows from the soundness and completeness of the syntax-directed transformation with respect to its non-syntax-directed counterpart:

Theorem 4 (Syntactic Soundness). If $\widehat{\Gamma}\ ;\ \varphi\ ;\ \psi \vdash e \triangleright e' :: \widehat{\tau}\ ;\ \widehat{\Gamma}'$, then $\widehat{\Gamma}' \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$. ∎

Theorem 5 (Syntactic Completeness). If $\widehat{\Gamma} \vdash e \triangleright e' :: \widehat{\tau}^{(\varphi,\psi)}$, then there exist $\widehat{\Gamma}'$ and e'', such that $\widehat{\Gamma}\ ;\ \varphi\ ;\ \psi \vdash e \triangleright e'' :: \widehat{\tau}\ ;\ \widehat{\Gamma}'$ with $e'' \lesssim e'$. ∎

8. Related Work

While strictness analysis has been around for over two decades, the problem of adapting strictness analysers to deal with *seq* and the like is ignored by most authors. A notable exception is Schmidt-Schauß (2008), who considers the semantics of *seq* in a safety proof for the strictness analysis of Nöcker (1993), which is based on abstract reduction and implemented in the Clean compiler. As far as we are aware, we are the first to consider the problem of extending a relevance-based strictness analysis to deal with both lazy and eager application.

An excellent introduction to substructural type systems is given by Walker (2005). Examples of the application of substructural typing to program analyses other than strictness analysis include work on uniqueness analysis (De Vries 2008) and sharing analysis (Hage et al. 2007). The context-split operation has been attributed to Cervesato and Pfenning (2002) and shows up in recent formulations of uniqueness analysis (De Vries 2008; Hage and Holdermans 2008).

Looking through a Curry-Howard lens, relevance type systems are connected to relevance logics (Anderson et al. 1975). Relevance has been put to use as an approximation of strictness by several authors, most prominently Wright (1991), Baker-Finch (1992), and Amtoft (1993). Indeed, relevance implies strictness, but they are not equivalent: for instance, $\lambda x \to \bot$ is strict but not relevant. Furthermore, while strictness is an extensional property of the functions defined by a program, relevance is an intensional property. Operationally, the focus of relevance type systems is on identifying the *needed redexes* (Barendregt et al. 1987) in a program. Now, whereas conventional relevance typing disciplines are foremost concerned with terms that appear in the argument position of needed beta-redexes, our refined system also aims at predicting which terms are guaranteed to appear in function position.

Type-based approaches of strictness analysis that keep track of totality rather than relevance are given by Kuo and Mishra (1989), Jensen (1991, 1998), Benton (1992), Solberg Gasser et al. (1998), Glynn et al. (2001), and Coppo et al. (2002). Of these, only the formulations of Glynn et al. and Coppo et al. support distinguishing between diverging functions and functions that produce diverging results. Hence, for the others, when applied to languages with explicit strictness annotations, the drawbacks reported on in this paper arise. A general discomfort of totality-based approaches is that, in comparison to relevance-based approaches, it is harder to read off the strictness of functions from their assigned types; on the other hand, totality-based formulations are arguably easier to grasp. As, in this paper, we specifically deal with type-based strictness analysis based on relevance typing, our refined approach does not directly apply to totality-based systems. Similar observations can be made for more traditional strictness analyses that are expressed as either abstract interpretations (Mycroft 1980; Burn et al. 1986; Wadler 1987) or projection analyses (Hughes 1988; Davis and Wadler 1990; Hinze 1995).

An elaborate exposition of the rôle played by *seq* in the design of Haskell is given by Hudak et al. (2007). In the process, they point out an interesting point in the design space: that is, having the (indirect) use of *seq* be reflected in the types assignable to polymorphic functions in order to recover the parametricity property. Seidel and Voigtländer (2009), however, demonstrate that the concrete method described by Hudak et al. is flawed and not adequate for regaining parametricity. They then propose an alternative type-based approach that does actually allow for parametricity to be recovered. This approach, which thus serves a somewhat different purpose than ours, is dual to the one taken in the present paper, in the sense that, rather than in establishing that a term is never used as an argument to *seq*, we are interested in demonstrating that a particular function is used at least once in a context in which it is not an argument to *seq*.

A precise account of the impact of *seq* on the so-called free theorems derivable from polymorphic types and some program transformations that are based on these theorems is given by Johann and Voigtländer (2006). Van Eekelen and De Mol (2006) discuss how properties derived for lazy programs can be adapted when these programs are decorated with explicit strictness annotations and show how the techniques involved can be incorporated in a proof assistant.

9. Conclusions and Further Work

We have demonstrated how a relevance type system for a completely lazy language can be adapted to a language with a construct for selectively making programs stricter. We have argued that it is not trivial to keep such an adaptation sound and, at the same time, have it satisfactorily account for the propagation of programmer-induced increases in strictness. In our approach, safe and effective analyses are derived in the context of an extended relevance type system that not only keeps track of demand propagation, but also of the so-called applicativeness of expressions.

While our system can be used as a basis for the design and implementation of type-based strictness analyses for modern lazy functional languages such as Haskell and Clean, it is not readily applicable to such languages yet as it fails to account for some essential features, such as type polymorphism and algebraic data types. Extending our type system to deal with such features remains future work, as is subjecting the analysis to techniques—such as subtyping and annotation polymorphism (Nielson and Nielson 1999)—that make it truly applicable to programming in the large. We stress, however, that these extensions are completely orthogonal to the issues focussed on in the present paper and fairly straightforward to implement.

Another direction for future work follows from the observation that, in our system, applicativeness implies relevance: if an expression of function type is guaranteed to be applied to an argument, it is also guaranteed to be evaluated. This suggests that all properties of interest can be captured in terms of elements of a single ternary lattice rather than in terms of the squared binary lattice that we implicitly used in the present paper. It would be interesting to see whether a *seq*-aware strictness analysis can then be elegantly formulated as an abstract interpretation in such a ternary abstract domain.

Acknowledgments

An earlier version of this paper was presented at the 2009 Symposium on Trends in Functional Programming under the title "Spreading the Joy". The work described in this paper was supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406). The authors wish to express their gratitude to the anonymous reviewers for their helpful comments and useful suggestions.

References

Torben Amtoft. Minimal thunkification. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Static Analysis, Third International Workshop, WSA '93, Padova, Italy, September 22–24, 1993, Proceedings*, volume 724 of *Lecture Notes of Computer Science*, pages 218–229. Springer-Verlag, 1993.

Alan Ross Anderson, Nuel D. Belnap Jr., and J. Michael Dunn. *Entailment: The Logic of Relevance and Necessity*, volume 1. Princeton University Press, Princeton, New Jersey, 1975.

Clement A. Baker-Finch. Relevant logic and strictness analysis. In Michel Billaud, Pierre Castéran, Marc-Michel Corsini, Kaninda Musumbu, and Antoine Rauzy, editors, *Actes WSA'92 Workshop on Static Analysis (Bordeaux), 23–25 September 1992, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings*, volume 81–82 of *Series Bigre*, pages 221–228. Atelier Irisa, 1992.

Henk P. Barendregt, Richard Kennaway, Jan Willem Klop, and M. Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75(3):191–231, 1987.

Nick Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, University of Cambridge, 1992.

Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. The theory of strictness analysis for higher order functions. In Harald Ganzinger and

Neil D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17–19, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 42–62. Springer-Verlag, 1986.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002.

Mario Coppo, Ferruccio Damiani, and Paola Giannini. Strictness analysis, totality, and non-standard-type inference. *Theoretical Computer Science*, 272(1–2):69–112, 2002.

Kei Davis and Philip Wadler. Backwards strictness analysis: Proved and improved. In Kei Davis and John Hughes, editors, *Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21–23 August 1989, Fraserburgh, Scotland, UK*, Workshops in Computing, pages 12–30. Springer-Verlag, 1990.

Marko van Eekelen and Maarten de Mol. Proof tool support for explicit strictness. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19–21, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 37–54. Springer-Verlag, 2006.

Andy Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *FPCA '93 Conference on Functional Programming and Computer Architecture. Copenhagen, Denmark, 9–11 June 1993*, pages 223–232. ACM Press, 1993.

Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. Effective strictness analysis with HORN constraints. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16–18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 73–92. Springer-Verlag, 2001.

Jurriaan Hage and Stefan Holdermans. Heap recycling for lazy languages. In John Hatcliff, Robert Glück, and Oege de Moor, editors, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'08, San Francisco, California, USA, January 7–8, 2008*, pages 189–197. ACM Press, 2008.

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007*, pages 235–246. ACM Press, 2007.

Ralf Hinze. *Projection-based Strictness Analysis: Theoretical and Practical Aspects*. PhD thesis, Bonn University, 1995.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9–10 June 2007*, pages 1–55. ACM Press, 2007.

John Hughes. Backwards analysis of functional programs. In Anders Björner, Neil D. Jones, and Andrei P. Ershov, editors, *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18–24 Oct., 1987*, pages 187–208. North-Holland, 1988.

John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

Thomas P. Jensen. Strictness analysis in logical form. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26–30, 1991, Proceedings*, pages 352–366. Springer-Verlag, 1991.

Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 209–221. ACM Press, 1998.

Patricia Johann and Janis Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.

Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *FPCA '89, Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, England, 11–13 September 1989*, pages 260–272. ACM Press, 1989.

Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming, Proceedings of the Fourth 'Colloque International sur la Programmation', Paris, France, 22–24 April 1980*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag, 1980.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, 1999.

Eric Nöcker. Strictness analysis using abstract reduction. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture. Copenhagen, Denmark, 9–11 June 1993*, pages 255–265. ACM Press, 1993.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean language report—version 1.3. Technical Report CSI-R9816, University of Nijmegen, 1998.

Manfred Schmidt-Schauß. Safety of Nöcker's strictness analysis. *Journal of Functional Programming*, 18(4):503–551, 2008.

Daniel Seidel and Janis Voigtländer. Taming selective strictness. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *INFORMATIK 2009 – Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28. September – 2. Oktober, in Lübeck*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, 2009.

Kirsten Lackner Solberg Gasser, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. *Science of Computer Programming*, 31(1):113–145, 1998.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

Edsko de Vries. *Making Uniqueness Typing Less Unique*. PhD thesis, Trinity College Dublin, 2008.

Edsko de Vries, Rinus Plasmeijer, and David Abrahamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007, Freiburg, Germany, September 2007, Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer-Verlag, 2008.

Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood, Chichester, 1987.

Philip Wadler. Theorems for free! In *FPCA '89 Conference on Functional Programming and Computer Architecture. Imperial College, London, England, 11–13 September 1989*, pages 347–359. ACM Press, 1989.

David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.

David A. Wright. A new technique for strictness analysis. In Samson Abramsky and Tom Maibaum, editors, *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8–12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 235–258. Springer-Verlag, 1991.