Higher-Order Symb Comput (2010) 23:315–335
DOI 10.1007/s10990-011-9079-7

Making "stricterness" more relevant

Stefan Holdermans · Jurriaan Hage

Published online: 24 November 2011

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.

Keywords Lazy evaluation · Strictness analysis · Relevance typing · Explicit strictness annotations · Functional languages · Type and effect systems

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* [31] and Clean's strictness annotations [32], 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 article makes the following contributions:

S. Holdermans
Vector Fabrics, Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
e-mail: stefan@vectorfabrics.com

J. Hage (✉)
Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: jur@cs.uu.nl

- We present an optimising transformation for a call-by-name lambda-calculus. The transformation is driven by a strictness analysis based on relevance typing (Sect. 4). We show how two straightforward extensions for dealing with a construct for selective strictness (Sect. 3) yield transformations that are either unsafe or otherwise ineffective in the sense that they fail to pick up on any increases in strictness that are incurred from the use of this construct (Sect. 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 (Sect. 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.

2 Background

The praises of lazy evaluation [20] have been sung in many voices 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 avoids the cost that is otherwise incurred by deferring the evaluation of argument expressions. Because strictness is generally an undecidable property, strictness analysers need to be conservative and cannot be expected to successfully identify all strict functions in a program. Moreover, in practice, many functions written in lazily evaluated languages turn out to be *almost* strict. For example, a function that consumes a given argument in all but one of the arms of a case analysis on another argument. 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 binary function *seq*, that evaluates its first argument (that is, forces it to *weak head normal form*) and returns its second. Furthermore, both languages allow for strictness annotations to occur in datatype declarations.

As an example of the use of such constructs, consider the definition of a function that returns its first argument, while ignoring its second argument, in some lazily evaluated functional language,

$$\lambda x.\lambda y.x,$$

and the following—stricter—version, that is defined in terms of a function *seq* with semantics as described above:

$$\lambda x.\lambda y.\mathit{seq}\ y\ x.$$

Whereas the former is strict only in its first argument (and lazy in its second), the latter is strict in both its arguments.

Of particular interest is that the increase in strictness induced by primitive operations like *seq* also propagates through function application. For instance, because of its use of the stricter version of the function that ignores its second argument, the otherwise lazy function

$$\lambda z.(\lambda x.\lambda y.\mathit{seq}\ y\ x)\ \mathsf{true}\ z,$$

that always produces the Boolean constant true, is now 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 $\lambda z.(\lambda x.\lambda y.\mathit{seq}\ y\ x)\ \mathsf{true}\ z$ could be evaluated eagerly rather than lazily. However, faithfully accounting for the effects of this so-called *selective strictness* is a tricky business as it comes with some peculiar semantic properties.

For one thing, selective strictness can be used to tell the programs $\bot$ and $\lambda x.\bot$ apart. As a simple example, the term $\mathit{seq}\ \bot\ \mathsf{true}$ diverges whereas $\mathit{seq}\ (\lambda x.\bot)\ \mathsf{true}$ produces the constant true. Note that this distinction cannot be made in either the "pure" call-by-name or call-by-need lambda-calculus;[1] 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 [40] and, so, optimisations such as short-cut deforestation [12] that are justified by parametricity are no longer sound.

Unsurprisingly, similar issues arise when adapting a strictness analysis for a mere non-strict lambda-calculus for use in a lazy language with constructs for selective strictness.

3 Preliminaries

To be able to consider the problem from a formal angle, we introduce a small, implicitly typed, nonstrict expression language. Assuming a countable infinite set of variable symbols ranged over by x,

$$x \in \mathbf{Var} \quad \text{variables},$$

its terms,

$$t \in \mathbf{Term} \quad \text{terms},$$

are given by

$$\begin{aligned} t ::= {} & \mathsf{false} \mid \mathsf{true} \mid 0 \mid x \mid \lambda x.t_1 \mid t_1\ t_2 \mid t_1 \bullet t_2 \mid \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \\ & \mid \mathsf{succ}\ t_1 \mid \mathsf{pred}\ t_1 \mid \mathsf{iszero}\ t_1 \mid \mathsf{error}. \end{aligned}$$

That is, terms include the Boolean constants false and true, the numeral 0, variables x, and lambda-abstractions $\lambda x.t_1$. Furthermore, we distinguish between nonstrict function applications $t_1\ t_2$ and strict function applications $t_1 \bullet t_2$. Conditionals are written as $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$; $\mathsf{succ}\ t_1$ and $\mathsf{pred}\ t_1$ denote, respectively, the successor and predecessor operations on natural numbers, while $\mathsf{iszero}\ t_1$ tests whether or not a given natural-number computation results in a zero result. Finally, error denotes a failing computation.

[1] From *within* these calculi, that is: one cannot write a function that behaves differently for $\bot$ and $\lambda x.\bot$.

Evaluation $\boxed{t \longrightarrow w}$

$$\frac{}{\mathsf{false} \longrightarrow \mathsf{false}}\,[\textit{e-false}] \qquad \frac{}{\mathsf{true} \longrightarrow \mathsf{true}}\,[\textit{e-true}] \qquad \frac{}{0 \longrightarrow 0}\,[\textit{e-zero}]$$

$$\frac{}{\lambda x.\, t_1 \longrightarrow \lambda x.\, t_1}\,[\textit{e-abs}]$$

$$\frac{t_1 \longrightarrow \lambda x.\, t_0 \quad [x \mapsto t_2]t_0 \longrightarrow w}{t_1\ t_2 \longrightarrow w}\,[\textit{e-app}] \qquad \frac{t_1 \longrightarrow \lambda x.\, t_0 \quad t_2 \longrightarrow w_2 \quad [x \mapsto w_2]t_0 \longrightarrow w}{t_1 \bullet t_2 \longrightarrow w}\,[\textit{e-sapp}]$$

$$\frac{t_1 \longrightarrow \mathsf{true} \quad t_2 \longrightarrow w}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow w}\,[\textit{e-if-true}] \qquad \frac{t_1 \longrightarrow \mathsf{false} \quad t_3 \longrightarrow w}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow w}\,[\textit{e-if-false}]$$

$$\frac{}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ t_1}\,[\textit{e-succ}] \qquad \frac{t_1 \longrightarrow 0}{\mathsf{pred}\ t_1 \longrightarrow 0}\,[\textit{e-pred-zero}]$$

$$\frac{t_1 \longrightarrow \mathsf{succ}\ t_0 \quad t_0 \longrightarrow w}{\mathsf{pred}\ t_1 \longrightarrow w}\,[\textit{e-pred-succ}]$$

$$\frac{t_1 \longrightarrow 0}{\mathsf{iszero}\ t_1 \longrightarrow \mathsf{true}}\,[\textit{e-iszero-zero}] \qquad \frac{t_1 \longrightarrow \mathsf{succ}\ t_0}{\mathsf{iszero}\ t_1 \longrightarrow \mathsf{false}}\,[\textit{e-iszero-succ}]$$

Fig. 1 Natural semantics

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, while function application is left-associative.

To ease the presentation in the sequel, strict function application is included instead of a primitive operation $\mathsf{seq}\ t_1\ t_2$. Note that this choice is by no means essential as—under a call-by-need semantics at least—strict function application and *seq* can always be defined in terms of each other. That is, given $\mathsf{seq}\ t_1\ t_2$ as part of the language, strict function application is given by

$$\lambda f.\, \lambda x.\, \mathsf{seq}\, x\ (f\ x),$$

whereas in the term language defined above, *seq* can be defined as

$$\lambda x.\, \lambda y.\, ((\lambda z.\, y) \bullet x).$$

A natural semantics [24] for the term language is given in Fig. 1 by means of rules for deriving judgements of the form $t \longrightarrow w$. That is, successful evaluation of a term t yields a weak head normal form w,

$$w \in \mathbf{Whnf} \quad \text{weak head normal forms},$$

given by

$$w ::= \mathsf{false} \mid \mathsf{true} \mid 0 \mid \lambda x.\, t_1.$$

As reflected by the rules [*e-false*], [*e-true*], [*e-zero*], [*e-abs*], and [*e-succ*], Boolean constants, the numeral 0, abstractions, applications of the successor operator are already in weak head normal form. Nonstrict and strict function applications are evaluated under a call-by-name and a call-by-value strategy, respectively; in the rules [*e-app*] and [*e-sapp*] beta-substitution is denoted by $[\cdot \mapsto \cdot]\,\cdot$. For conditionals $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$, rules [*e-if-true*] and [*e-if-false*] evaluate the condition t_1 and proceed by evaluating one of the branches t_2 and t_3. Taking the predecessor of a computation that results in 0, yields 0 (rule schema [*e-pred-zero*]), while taking the predecessor of a computation that produces a result of the form $\mathsf{succ}\ t_0$ for some t_0 amounts to evaluating t_0 (rule schema [*e-pred-succ*]). Rules [*e-iszero-zero*] and [*e-iszero-succ*] denote that a zero-equality test $\mathsf{iszero}\ t_1$ for a term t_1 results in true if t_1 evaluates to 0 and false if t_1 evaluates to $\mathsf{succ}\ t_0$ for some t_0.

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

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ [\textit{t-false}] \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ [\textit{t-true}] \qquad \frac{}{\Gamma \vdash 0 : \mathsf{nat}}\ [\textit{t-zero}]$$

$$\frac{}{\Gamma_1[x \mapsto \tau] \vdash x : \tau}\ [\textit{t-var}] \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Gamma \vdash \lambda x.\, t_1 : \tau_1 \to \tau_2}\ [\textit{t-abs}]$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}\ [\textit{t-app}] \qquad \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 \bullet t_2 : \tau}\ [\textit{t-app}]$$

$$\frac{\Gamma \vdash t_1 : \mathsf{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \tau}\ [\textit{t-if}]$$

$$\frac{\Gamma \vdash t_1 : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}\ t_1 : \mathsf{nat}}\ [\textit{t-succ}] \qquad \frac{\Gamma \vdash t_1 : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\ t_1 : \mathsf{nat}}\ [\textit{t-pred}] \qquad \frac{\Gamma \vdash t_1 : \mathsf{nat}}{\Gamma \vdash \mathsf{iszero}\ t_1 : \mathsf{bool}}\ [\textit{t-iszero}]$$

$$\frac{}{\Gamma \vdash \mathsf{error} : \tau}\ [\textit{t-error}]$$

Fig. 2 The underlying type system

We say that evaluation of a term t fails if there is no weak head normal form w such that $t \longrightarrow w$. Note that we do not have rules for evaluating error; hence, evaluation of error always fails.

A type system for our language is presented in Fig. 2 as a set of type-assignment rules for deriving judgements $\Gamma \vdash t : \tau$, expressing that, in the type environment Γ, the term t can be assigned the type τ. Here, types,

$$\tau \in \mathbf{Type} \quad \text{types},$$

are given by

$$\tau ::= \mathsf{bool} \mid \mathsf{nat} \mid \tau_1 \to \tau_2,$$

with bool the type of Booleans, nat the type of natural numbers, 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:

$$\Gamma \in \mathbf{TEnv} = \mathbf{Var} \to_{\text{fin}} \mathbf{Type} \quad \text{type environments.}$$

We write $[\,]$ for the empty map, $[x \mapsto \tau]$ for the singleton map that binds x to τ, and $\Gamma_1[x \mapsto \tau]$ for the map that is obtained by extending Γ_1 with a binding from x to τ.

The assignment rules in Fig. 2 are completely standard. Note in particular that nonstrict applications $t_1\ t_2$ and strict applications $t_1 \bullet t_2$ have the same static semantics, and that error can be assigned any type. In the sequel, we are only concerned with well-typed terms, i.e., terms for which, in a given environment Γ, there is at least one type τ with $\Gamma \vdash e : \tau$. The type system of Fig. 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 type system for keeping track of *relevance*. Similar systems have been considered by Wright [42] and Amtoft [1]; the presentation below is largely inspired by Walker [41].

We will use the analysis to drive program transformations that safely replace nonstrict function applications by strict 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 strict function applications. In Sect. 5 and Sect. 6, we will then admit strict applications in source programs as well and show that dealing with these in an adequate manner is actually quite involved.

4.1 Annotated types

A variable x is *relevant* to a term t if any term bound to x is guaranteed to be evaluated whenever t is evaluated. We say that an abstraction $\lambda x.t_1$ is a *relevant abstraction* if its formal parameter x is relevant to its body t_1.

To distinguish between expressions that evaluate to relevant abstractions and expressions that evaluate to abstractions that may not be relevant, we introduce annotated types,

$$\widehat{\tau} \in \widehat{\mathbf{Type}} \quad \text{annotated types,}$$

given by

$$\widehat{\tau} ::= \mathsf{bool} \mid \mathsf{nat} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2.$$

That is, an annotated type $\widehat{\tau}$ is either one of the base types bool and nat or else a function type $\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$ where φ ranges over annotations,

$$\varphi \in \mathbf{Ann} \quad \text{annotations,}$$

for which we have

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

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

As an illustration of how to assign annotated types, consider the terms $\lambda x.x$ and $\lambda x.\mathsf{true}$. The former term denotes a relevant abstraction, whereas the latter does not. Hence, for any annotated type $\widehat{\tau}$, the type $\widehat{\tau} \xrightarrow{\mathsf{S}} \widehat{\tau}$ would be a valid type for $\lambda x.x$, while $\lambda x.\mathsf{true}$ should always receive a type of the form $\widehat{\tau} \xrightarrow{\mathsf{L}} \mathsf{bool}$.

Note that relevance implies strictness: if a variable x is relevant to a term t, then t is strict in x and, so, relevant abstractions denote strict functions.[2] 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 a partial order $(\mathbf{Ann}, \sqsubseteq)$ on annotations that is characterised by $\mathsf{S} \sqsubseteq \mathsf{L}$. Hence, $(\mathbf{Ann}, \sqsubseteq)$ is a complete lattice with least element S and greatest element L, and joins and meets given by

$$\mathsf{S} \sqcup \varphi = \varphi,$$
$$\mathsf{L} \sqcup \varphi = \mathsf{L}$$

and

$$\mathsf{S} \sqcap \varphi = \mathsf{S},$$
$$\mathsf{L} \sqcap \varphi = \varphi.$$

[2]However, relevance and strictness are not equivalent: for instance, $\lambda x.\mathsf{error}$ is strict but not relevant.

4.2 Type-driven call-by-value transformation

Our goal will now be to transform programs by turning as many nonstrict applications of strict functions into strict applications as possible. That is, if for a given nonstrict application $t_1\ t_2$ it can be shown that successfully evaluating t_1 will always result in a relevant abstraction, then we optimise away the assumed overhead of nonstrict evaluation by replacing the application by its strict counterpart $t_1 \bullet t_2$. This optimisation is justified by the observation that from the implied relevance of the function t_1 it follows that, for the application to produce a result, evaluation of the argument t_2 is required anyway.

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

$$\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{\varphi},$$

expressing that, in the annotated type environment $\widehat{\Gamma}$, the source term t, of type $\widehat{\tau}$ and annotated with φ, can be safely transformed into the target term t'. Here, the annotation φ is used to indicate whether or not the context in which t appears guarantees its evaluation: t is said to be *demanded* if $\varphi = \mathsf{S}$. When analysing the body of an abstraction, we will set its annotation to S and observe whether the modelled demand propagates to its parameter: if it does, the abstraction is relevant; if it does not, the abstraction may be irrelevant.

Annotated type environments,

$$\widehat{\Gamma} \in \mathbf{TEnv} = \mathbf{Var} \to_{\mathrm{fin}} \widehat{\mathbf{Type}} \times \mathbf{Ann} \quad \text{annotated type environments},$$

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 Sect. 3, we write $[\,]$ for the empty environment, $[x \mapsto (\widehat{\tau}, \varphi)]$ for the singleton environment that maps x to $(\widehat{\tau}, \varphi)$, and $\widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi)]$ for the environment that is obtained by extending $\widehat{\Gamma}_1$ with a binding from x to $(\widehat{\tau}, \varphi)$.

The rules of the transformation relation are given in Fig. 3.

Relevance typing constitutes a so-called *substructural typing discipline* [41], which is reflected by a 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-false*], [*r-true*], [*r-zero*], and [*r-error*], for instance, require that any constant terms are transformed in an empty type environment. All constants can be typed and transformed in any context; false and true always have the type bool, 0 always has the type nat, and error can be assigned any type. Variables, as per rule schema [*r-var*], are typed and transformed in singleton environments that agree with the types and annotations assigned.

Interesting is the rule schema for lambda-abstractions, [*r-abs*]. It states that typing and transforming an abstraction $\lambda x.\,t_1$ depends on the typing and transformation of its body t_1 in a type environment 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 t_1—and thus whether the abstraction itself is relevant—we “reset” the demand context for t_1 to S. If x can then be annotated with S as well, we conclude that $\lambda x.\,t_1$ is relevant; otherwise, we classify the abstraction as possibly irrelevant. To ensure that resetting the demand for the body does not propagate 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 a so-called *containment restriction* $\varphi \blacktriangleright \widehat{\Gamma}$, the rules for

Transformation $\boxed{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{\varphi}}$

$$\frac{}{[\,] \vdash \mathsf{false} \rhd \mathsf{false} : \mathsf{bool}^{\varphi}}\;[\textit{r-false}] \qquad \frac{}{[\,] \vdash \mathsf{true} \rhd \mathsf{true} : \mathsf{bool}^{\varphi}}\;[\textit{r-true}]$$

$$\frac{}{[\,] \vdash 0 \rhd 0 : \mathsf{nat}^{\varphi}}\;[\textit{r-zero}] \qquad \frac{}{[x \mapsto (\widehat{\tau}, \varphi)] \vdash x \rhd x : \widehat{\tau}^{\varphi}}\;[\textit{r-var}]$$

$$\frac{\varphi \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto (\widehat{\tau}_1, \varphi_1)] \vdash t_1 \rhd t_1' : \widehat{\tau}_2^{\mathsf{S}}}{\widehat{\Gamma} \vdash \lambda x.\, t_1 \rhd \lambda x.\, t_1' : (\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2)^{\varphi}}\;[\textit{r-abs}]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\mathsf{S}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1\, t_2 \rhd t_1' \bullet t_2' : \widehat{\tau}^{\varphi}}\;[\textit{r-app}_1]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\mathsf{L}}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1\, t_2 \rhd t_1'\, t_2' : \widehat{\tau}^{\varphi}}\;[\textit{r-app}_2]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : \mathsf{bool}^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}^{\mathsf{L}} \quad \widehat{\Gamma}_2 \vdash t_3 \rhd t_3' : \widehat{\tau}^{\mathsf{L}}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rhd \mathsf{if}\ t_1'\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3' : \widehat{\tau}^{\varphi}}\;[\textit{r-if}]$$

$$\frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \mathsf{nat}^{\mathsf{L}}}{\widehat{\Gamma} \vdash \mathsf{succ}\ t_1 \rhd \mathsf{succ}\ t_1' : \mathsf{nat}^{\varphi}}\;[\textit{r-succ}] \qquad \frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \mathsf{nat}^{\varphi}}{\widehat{\Gamma} \vdash \mathsf{pred}\ t_1 \rhd \mathsf{pred}\ t_1' : \mathsf{nat}^{\varphi}}\;[\textit{r-pred}]$$

$$\frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \mathsf{nat}^{\varphi}}{\widehat{\Gamma} \vdash \mathsf{iszero}\ t_1 \rhd \mathsf{iszero}\ t_1' : \mathsf{bool}^{\varphi}}\;[\textit{r-iszero}]$$

$$\frac{}{[\,] \vdash \mathsf{error} \rhd \mathsf{error} : \widehat{\tau}^{\varphi}}\;[\textit{r-error}] \qquad \frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{\mathsf{L}}}{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{\mathsf{S}}}\;[\textit{r-sub}] \qquad \frac{\widehat{\Gamma}_1 \vdash t \rhd t' : \widehat{\tau}^{\varphi}}{\widehat{\Gamma}_1[x \mapsto (\widehat{\tau}_0, \mathsf{L})] \vdash t \rhd t' : \widehat{\tau}^{\varphi}}\;[\textit{r-weak}]$$

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

Containment $\boxed{\varphi \blacktriangleright \widehat{\Gamma}}$

$$\frac{}{\varphi \blacktriangleright [\,]}\;[\textit{c-nil}] \qquad \frac{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi_0)]}\;[\textit{c-cons-s}] \qquad \frac{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathsf{L})]}\;[\textit{c-cons-l}]$$

Fig. 4 Containment

which are given in Fig. 4. Note in particular that $\mathsf{L} \blacktriangleright \widehat{\Gamma}$ if and only if all bindings in $\widehat{\Gamma}$ carry the annotation L.

There are two rule schemata that deal with nonstrict function applications $t_1\ t_2$. Recall that we are not dealing with strict function applications yet. The first, [*r-app*$_1$], is applicable whenever the function term t_1 can be assigned an S-annotated function type and thus constitutes a strict function. Then, the argument t_2 is demanded whenever the result of the application is and, hence, the demand φ of the application propagates to t_2. Importantly, here we seize on the opportunity and transform the nonstrict application $t_1\ t_2$ into the strict application $t_1' \bullet t_2'$. The second schema, [*r-app*$_2$], deals with the application $t_1\ t_2$ of a possibly nonstrict function t_1. In that case, there are no guarantees about the demand for the argument and so t_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 $\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2$ of the environments $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$ in which, respectively, the subterms t_1 and t_2 are analysed. That is, $\cdot \diamond \cdot$ is a partial operation on annotated type environments with:

$$\begin{aligned} [\,] \diamond [\,] \qquad\qquad\qquad\qquad &= [\,], \\ \widehat{\Gamma}_{11}[x \mapsto (\widehat{\tau}, \varphi_1)] \diamond \widehat{\Gamma}_{21}[x \mapsto (\widehat{\tau}, \varphi_2)] &= (\widehat{\Gamma}_{11} \diamond \widehat{\Gamma}_{21})[x \mapsto (\widehat{\tau}, \varphi_1 \sqcap \varphi_2)]. \end{aligned}$$

This "context split" [8] reflects that a variable is relevant to $t_1\ t_2$ if its relevance can be established in at least one of the subanalyses for t_1 and t_2. The same operation is used in the rule schema for conditionals, [*r-if*], in which the type environment is split in two parts: one for the guard and one for the conditional branches. As only one of them will be evaluated, both branches are analysed in an L-context.

The rules [*r-succ*], [*r-pred*], and [*r-iszero*] are straightforward. Operands t_1 in $\mathsf{succ}\ t_1$ receive L-contexts to reflect that weak head normal form has been reached when a successor operation is produced.

The rule [*r-sub*] implements *subeffecting* [36] 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 a term is not demanded are still valid if the term is in fact demanded.

Finally, the weakening rule [*r-weak*], expresses that any transformation that is derivable in a given annotated type environment $\widehat{\Gamma}_1$ that does not contain a binding for a variable x can also be derived in an annotated type environment that is obtained by adding an L-annotated binding for x to $\widehat{\Gamma}_1$.

4.3 Examples

Let us now consider two examples of transformations in our system. First, consider the term

$$(\lambda x.\lambda y.\,x)\ \mathsf{true}\ \mathsf{false}$$

and its transformation into

$$((\lambda x.\,\lambda y.x) \bullet \mathsf{true})\ \mathsf{false},$$

justified by the following derivation of

$$[\,] \vdash \lambda x.\,\lambda y.\,x \rhd \lambda x.\,\lambda y.\,x : (\mathsf{bool} \xrightarrow{\mathsf{S}} \mathsf{bool} \xrightarrow{\mathsf{L}} \mathsf{bool})^{\mathsf{S}}$$

in our system (the target terms are omitted for clarity):

$$\frac{\dfrac{}{\mathsf{S} \blacktriangleright [\,]} \qquad \dfrac{\dfrac{}{\mathsf{S} \blacktriangleright [x \mapsto (\mathsf{bool}, \mathsf{S})]} \qquad \dfrac{\dfrac{}{[x \mapsto (\mathsf{bool}, \mathsf{S})] \vdash x : \mathsf{bool}^{\mathsf{S}}}}{[x \mapsto (\mathsf{bool}, \mathsf{S})][y \mapsto (\mathsf{bool}, \mathsf{L})] \vdash x : \mathsf{bool}^{\mathsf{S}}}}{[x \mapsto (\mathsf{bool}, \mathsf{S})] \vdash \lambda y.\,x : (\mathsf{bool} \xrightarrow{\mathsf{L}} \mathsf{bool})^{\mathsf{S}}}}{[\,] \vdash \lambda x.\,\lambda y.\,x : (\mathsf{bool} \xrightarrow{\mathsf{S}} \mathsf{bool} \xrightarrow{\mathsf{L}} \mathsf{bool})^{\mathsf{S}}}\ .$$

Here, the assigned typed $(\mathsf{bool} \xrightarrow{\mathsf{S}} \mathsf{bool} \xrightarrow{\mathsf{L}} \mathsf{bool})$ reflects that the function that is produced by $\lambda x.\,\lambda y.\,x$ is strict in its first argument. Hence, the innermost application in

$$(\lambda x.\,\lambda y.\,x)\ \mathsf{true}\ \mathsf{false}$$

can be transformed into a strict application.

As an example of a case in which the use of subeffecting is crucial, consider the analysis of the term

$$\lambda x.\,(\lambda y.\,\mathsf{true})\;(\lambda z.\,x).$$

As the abstraction $\lambda y.\,\mathsf{true}$ results in a lazy function, the function argument $\lambda z.\,x$ must be analysed in an L-context. The containment restriction for $\lambda z.\,x$ then prescribes that x, in the type environment for the abstraction, has to be L-annotated. However, because the body of the abstraction must be analysed in an S-context, we have to somehow be able to derive the judgement

$$[x \mapsto (\widehat{\tau}, \mathsf{L})] \vdash x \rhd x : \widehat{\tau}^{\mathsf{S}}$$

for some annotated type $\widehat{\tau}$. Here, subeffecting allows us to obtain the desired judgement from the trivially fulfilled premise

$$[x \mapsto (\widehat{\tau}, \mathsf{L})] \vdash x \rhd x : \widehat{\tau}^{\mathsf{L}}.$$

5 Dealing with strict 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 strict applications: in our relevance type system of Sect. 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 strict applications in source terms. Naturally, when doing so, we want to make sure that transformation remains safe. Moreover, we 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 strict 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 strict 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 (Sect. 5.1) or effective but unsafe (Sect. 5.2).

5.1 A conservative approach

Arguably the simplest way to extend the transformation with support for strict applications is to treat strict applications as if they were nonstrict applications. That is, we add two rules,

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\mathsf{S}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 \bullet t_2 \rhd t_1' \bullet t_2' : \widehat{\tau}^{\varphi}}$$

and

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\mathsf{L}}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 \bullet t_2 \rhd t_1' \bullet t_2' : \widehat{\tau}^{\varphi}},$$

that are identical to the rules [*r-app*$_1$] and [*r-app*$_2$] except that they deal with strict applications rather than nonstrict applications.

However, while these additions do bring strict 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 strict application. For example, the rules cannot derive the relevance of the following abstraction (cf. Sect. 2),

$$\lambda z.\,(((\lambda x.\,\lambda y.\,x)\ \mathsf{true}) \bullet z), \tag{1}$$

and, hence, nonstrict applications of this abstraction are unaffected by associated call-by-value transformations—even though it would be completely safe to replace them by strict applications.

5.2 A more ambitious attempt

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

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau})^{\varphi} \qquad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 \bullet t_2 \rhd t_1' \bullet t_2' : \widehat{\tau}^{\varphi}}.$$

Note how this rule resembles the rule for nonstrict applications of strict functions, [*r-app*$_1$], with the notable exception that the actual relevance φ_0 of the function expression t_1 is completely ignored here.

This new rule seems to capture the semantics of strict 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 completely breaks the transformation system! To see this, consider the abstraction

$$\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)) \tag{2}$$

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

$$(\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)))\ \mathsf{error} \tag{3}$$

evaluates to the numeral 0, while evaluation of the stricter

$$(\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x))) \bullet \mathsf{error} \tag{4}$$

obviously fails. Still, with the suggested rule for strict applications we can derive that $\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x))$ has type $\widehat{\tau} \xrightarrow{\mathsf{S}} \mathsf{nat}$ for any annotated type $\widehat{\tau}$:

$$\dfrac{\dfrac{}{\mathsf{S} \blacktriangleright [\,]} \qquad \dfrac{\dfrac{\vdots}{[x \mapsto (\widehat{\tau}, \mathsf{L})] \vdash \lambda y.\,0 : ((\widehat{\tau}_0 \xrightarrow{\mathsf{L}} \widehat{\tau}) \xrightarrow{\mathsf{L}} \mathsf{nat})^{\mathsf{S}}} \qquad \dfrac{\dfrac{\mathsf{S} \blacktriangleright [\,]}{\mathsf{S} \blacktriangleright [x \mapsto (\widehat{\tau}, \mathsf{S})]} \qquad \dfrac{\dfrac{}{[x \mapsto (\widehat{\tau}, \mathsf{S})] \vdash x : \widehat{\tau}^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})][z \mapsto (\widehat{\tau}_0, \mathsf{L})] \vdash x : \widehat{\tau}^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})] \vdash \lambda z.\,x : (\widehat{\tau}_0 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})] \vdash (\lambda y.\,0) \bullet (\lambda z.\,x) : \mathsf{nat}^{\mathsf{S}}}}{[\,] \vdash \lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)) : (\widehat{\tau} \xrightarrow{\mathsf{S}} \mathsf{nat})^{\mathsf{S}}}.$$

That is, we can—wrongfully—derive that term (2) is a relevant abstraction. As a result, it allows term (3) to be transformed into term (4)—which is clearly unsafe.

Now, with such transformations admitted, the proposed rule for strict applications is of course not fit for inclusion in our system. However, rather than completely abandoning the proposed approach to dealing with strict applications, in the next section we will analyse exactly what 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 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 Sect. 4 with support for eager function applications in the source language does not require much effort in itself (Sect. 5.1); but also that our desire to have the system reflect the semantics of these eager applications apparently complicates matters considerably (Sect. 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 considered before. As it turns out, the key to success lies in establishing where the approach of Sect. 5.2 went wrong.

6.1 Stocktaking

Let us have a closer look at how the derivation tree for term (2) in Sect. 5.2,

$$\dfrac{\dfrac{}{\mathsf{S} \blacktriangleright [\,]} \qquad \dfrac{\dfrac{\vdots}{[x \mapsto (\widehat{\tau}, \mathsf{L})\,] \vdash \lambda y.\,0 : ((\widehat{\tau}_0 \xrightarrow{\mathsf{L}} \widehat{\tau}) \xrightarrow{\mathsf{L}} \mathsf{nat})^{\mathsf{S}}} \qquad \dfrac{\dfrac{\mathsf{S} \blacktriangleright [\,]}{\mathsf{S} \blacktriangleright [x \mapsto (\widehat{\tau}, \mathsf{S})\,]} \quad \dfrac{\dfrac{}{[x \mapsto (\widehat{\tau}, \mathsf{S})\,] \vdash x : \widehat{\tau}^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})\,][z \mapsto (\widehat{\tau}_0, \mathsf{L})\,] \vdash x : \widehat{\tau}^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})\,] \vdash \lambda z.\,x : (\widehat{\tau}_0 \xrightarrow{\mathsf{L}} \widehat{\tau})^{\mathsf{S}}}}{[x \mapsto (\widehat{\tau}, \mathsf{S})\,] \vdash (\lambda y.\,0) \bullet (\lambda z.\,x) : \mathsf{nat}^{\mathsf{S}}}}{[\,] \vdash \lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)) : (\widehat{\tau} \xrightarrow{\mathsf{S}} \mathsf{nat})^{\mathsf{S}}},$$

enabled us to inappropriately conclude that the given abstraction was relevant. To do so, it had to derive that the formal parameter x of the abstraction was relevant to its body. Since the body is an application, this is achieved by deriving that x is relevant to at least one of the two subterms of the application. Clearly, x cannot be relevant to the function term $\lambda y.\,0$ as it does not even occur in it. The only possibility left is then to establish that x is relevant to the argument abstraction $\lambda z.\,x$. Recall that we always analyse the body of an abstraction as if it were demanded. Then, as x occurs in the body, we can conclude that x is at least locally relevant. As x is a free variable of $\lambda z.\,x$, the containment restriction prescribes that this conclusion can only be propagated globally if the abstraction $\lambda z.\,x$ is itself demanded. However, as the demand for $\lambda z.\,x$ is implied by its use as an argument in a demanded strict application, the containment restriction is met trivially and so we indeed derive that x is relevant to $(\lambda y.\,0) \bullet (\lambda z.\,x)$.

Now—where did we go wrong? A moment's reflection reveals that, with a progressive rule for strict applications in place, the containment restriction does not serve its purpose anymore. It was there to ensure that we could not conclude that a term 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 $\lambda z.\,x$ in the derivation in Sect. 5.2. So, why is the containment restriction adequate in the language of Sect. 4, but not in the language of Sect. 5.2?

The answer lies in the observation that, if we leave strict 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 distinguish between $\bot$ and $\lambda x.\,\bot$. However, as we demonstrated in Sect. 2, with constructs like *seq* this is no longer the case. Indeed, with strict applications in the source language, we can force the evaluation of an abstraction without ever evaluating its body and this is what happens to $\lambda z.\,x$ in the expression $(\lambda y.\,0) \bullet (\lambda z.\,x)$. In our relevance type system this is reflected as follows: for the original language, parameter types are permitted to be annotated with S only if the corresponding parameters are relevant to the bodies of their functions. However, with the suggested rule for strict applications, S-annotations can cross over to the environments of abstractions even if they are irrelevant to these abstractions.

These considerations suggest replacing the containment restriction by either a stronger constraint that does not 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 gives rise to transformations that are even less effective than the conservative transformations of Sect. 5.1, as it would prevent us from detecting the strictness of curried functions in any but the last of their arguments. For example, to type $\lambda x.\,\lambda y.\,x$, we first have to analyse the inner abstraction $\lambda y.\,x$ in which x occurs free. If we can no longer assign x an S-annotation there, then we cannot derive that it is relevant to the outer abstraction. 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 Sect. 4, we used annotations S and L to keep track of whether or not terms were demanded by their contexts. We will now refine the transformation system and have these annotations also indicate the *applicativeness* of terms. We say that a term 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 Sect. 4 for our refined system and call attention to any differences with respect to the original system.

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:

$$\varphi, \psi \in \mathbf{Ann} \quad \text{annotations},$$
$$\varphi, \psi ::= \mathsf{S} \mid \mathsf{L}.$$

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 before, we let $\widehat{\tau}$ range over annotated types,

$$\widehat{\tau} \in \widehat{\mathbf{Type}} \quad \text{annotated types.}$$

Function types are now decorated with information about the applicativeness of the arguments and results of functions:

$$\widehat{\tau} ::= \mathsf{bool} \mid \mathsf{nat} \mid \widehat{\tau}_1^{\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2^{\psi_2}.$$

Furthermore, annotated type environments 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:

$$\widehat{\Gamma} \in \widehat{\mathbf{TEnv}} = \mathbf{Var} \to_{\text{fin}} \widehat{\mathbf{Type}} \times \mathbf{Ann} \times \mathbf{Ann} \quad \text{annotated type environments.}$$

The judgements of the refined transformation relation read

$$\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi, \psi)},$$

where the annotation ψ indicates the applicativeness of the term t. 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 $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi, \psi)}$, it holds that $\varphi \sqsubseteq \psi$.

The rules for deriving judgements of the given form are listed in Fig. 5. The rules for constants—[*r-false*], [*r-true*], and [*r-zero*]—express that constants can never appear in contexts in which they are applied to arguments. 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: terms bound to the free variables that are relevant to the body of an abstraction are only guaranteed to be demanded if the abstraction itself is guaranteed to be applied. The rules for deriving containment need to be updated as well to reflect that type environments now also contain annotations for applicativeness: see Fig. 6.

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

$$\begin{aligned} [\,] \diamond [\,] &= [\,], \\ \widehat{\Gamma}_{11}[x \mapsto (\widehat{\tau}, \varphi_1, \psi_1)] \diamond \widehat{\Gamma}_{21}[x \mapsto (\widehat{\tau}, \varphi_2, \psi_2)] &= (\widehat{\Gamma}_{11} \diamond \widehat{\Gamma}_{21})[x \mapsto (\widehat{\tau}, \varphi_1 \sqcap \varphi_2, \psi_1 \sqcap \psi_2)]. \end{aligned}$$

In each of the rules [*r-app*$_1$], [*r-app*$_2$], and [*r-sapp*], the applicativeness of the function term t_1 follows from the demand for the application. In the two rules for nonstrict applications, the applicativeness of the argument term t_2 is obtained by taking the join of the demand for the application and the applicativeness of arguments of t_1. That is, arguments are guaranteed to be used applicatively if the application is guaranteed to be performed and the function is guaranteed to use its argument applicatively. In the case for strict applications, we ignore, as we did in Sect. 5.2, the strictness of the function expression t_1 and propagate the demand for the application directly to the argument term t_2; furthermore, the applicativeness of t_2 is obtained by combining the demand φ for the application and the applicativeness ψ_2 of the parameter of t_1.

Transformation $\boxed{\widehat{\Gamma} \vdash t \triangleright t' : \widehat{\tau}^{(\varphi,\psi)}}$

$$\frac{}{[\,] \vdash \mathsf{false} \triangleright \mathsf{false} : \mathsf{bool}^{(\varphi,\mathsf{L})}}\ [\textit{r-false}] \qquad \frac{}{[\,] \vdash \mathsf{true} \triangleright \mathsf{true} : \mathsf{bool}^{(\varphi,\mathsf{L})}}\ [\textit{r-true}]$$

$$\frac{}{[\,] \vdash 0 \triangleright 0 : \mathsf{nat}^{(\varphi,\mathsf{L})}}\ [\textit{r-zero}] \qquad \frac{}{[x \mapsto (\widehat{\tau},\varphi,\psi)] \vdash x \triangleright x : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-var}]$$

$$\frac{\psi \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto (\widehat{\tau}_1,\varphi_1,\psi_1)] \vdash t_1 \triangleright t_1' : \widehat{\tau}_2^{(\mathsf{S},\psi_2)}}{\widehat{\Gamma} \vdash \lambda x.\, t_1 \triangleright \lambda x.\, t_1' : (\widehat{\tau}_1^{\psi_1} \xrightarrow{\varphi_1} \widehat{\tau}_2^{\psi_2})^{(\varphi,\psi)}}\ [\textit{r-abs}]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \triangleright t_1' : (\widehat{\tau}_2^{\psi_2} \xrightarrow{\mathsf{S}} \widehat{\tau}^{\psi})^{(\varphi,\varphi)} \quad \widehat{\Gamma}_2 \vdash t_2 \triangleright t_2' : \widehat{\tau}_2^{(\varphi,\varphi\sqcup\psi_2)}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1\ t_2 \triangleright t_1' \bullet t_2' : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-app}_1]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \triangleright t_1' : (\widehat{\tau}_2^{\psi_2} \xrightarrow{\mathsf{L}} \widehat{\tau}^{\psi})^{(\varphi,\varphi)} \quad \widehat{\Gamma}_2 \vdash t_2 \triangleright t_2' : \widehat{\tau}_2^{(\mathsf{L},\varphi\sqcup\psi_2)}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1\ t_2 \triangleright t_1'\ t_2' : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-app}_2]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \triangleright t_1' : (\widehat{\tau}_2^{\psi_2} \xrightarrow{\varphi_0} \widehat{\tau}^{\psi})^{(\varphi,\varphi)} \quad \widehat{\Gamma}_2 \vdash t_2 \triangleright t_2' : \widehat{\tau}_2^{(\varphi,\varphi\sqcup\psi_2)}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 \bullet t_2 \triangleright t_1' \bullet t_2' : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-sapp}]$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \triangleright t_1' : \mathsf{bool}^{(\varphi,\mathsf{L})} \quad \widehat{\Gamma}_2 \vdash t_2 \triangleright t_2' : \widehat{\tau}^{(\mathsf{L},\psi)} \quad \widehat{\Gamma}_2 \vdash t_3 \triangleright t_3' : \widehat{\tau}^{(\mathsf{L},\psi)}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \triangleright \mathsf{if}\ t_1'\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3' : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-if}]$$

$$\frac{\widehat{\Gamma} \vdash t_1 \triangleright t_1' : \mathsf{nat}^{(\mathsf{L},\mathsf{L})}}{\widehat{\Gamma} \vdash \mathsf{succ}\ t_1 \triangleright \mathsf{succ}\ t_1' : \mathsf{nat}^{(\varphi,\mathsf{L})}}\ [\textit{r-succ}] \qquad \frac{\widehat{\Gamma} \vdash t_1 \triangleright t_1' : \mathsf{nat}^{(\varphi,\mathsf{L})}}{\widehat{\Gamma} \vdash \mathsf{pred}\ t_1 \triangleright \mathsf{pred}\ t_1' : \mathsf{nat}^{(\varphi,\mathsf{L})}}\ [\textit{r-pred}]$$

$$\frac{\widehat{\Gamma} \vdash t_1 \triangleright t_1' : \mathsf{nat}^{(\varphi,\mathsf{L})}}{\widehat{\Gamma} \vdash \mathsf{iszero}\ t_1 \triangleright \mathsf{iszero}\ t_1' : \mathsf{bool}^{(\varphi,\mathsf{L})}}\ [\textit{r-iszero}]$$

$$\frac{}{[\,] \vdash \mathsf{error} \triangleright \mathsf{error} : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-error}] \qquad \frac{\widehat{\Gamma} \vdash t \triangleright t' : \widehat{\tau}^{(\mathsf{L},\mathsf{L})}}{\widehat{\Gamma} \vdash t \triangleright t' : \widehat{\tau}^{(\mathsf{S},\psi)}}\ [\textit{r-sub}]$$

$$\frac{\widehat{\Gamma}_1 \vdash t \triangleright t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma}_1[x \mapsto (\widehat{\tau}_0,\mathsf{L},\mathsf{L})] \vdash t \triangleright t' : \widehat{\tau}^{(\varphi,\psi)}}\ [\textit{r-weak}]$$

Fig. 5 Relevance typing and call-by-value transformation. (Cf. Fig. 3)

Containment $\boxed{\psi \blacktriangleright \widehat{\Gamma}}$

$$\frac{}{\psi \blacktriangleright [\,]}\ [\textit{c-nil}] \qquad \frac{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau},\varphi_0,\psi_0)]}\ [\textit{c-cons-s}] \qquad \frac{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau},\mathsf{L},\mathsf{L})]}\ [\textit{c-cons-l}]$$

Fig. 6 Containment. (Cf. Fig. 4)

Adapting the rules for terms of the forms $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$, $\mathsf{succ}\ t_1$, $\mathsf{pred}\ t_1$, and $\mathsf{iszero}\ t_1$ is straightforward. Note that Booleans and natural numbers are never to occur in function position and hence can never be used applicatively. Rule [*r-error*] expresses that whether an occurrence of error is applicative depends on its context.

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 strict applications, consider the analysis of term (1),

$$\lambda z.\,((\lambda x.\,\lambda y.\,x)\ \mathsf{true} \bullet z),$$

from Sect. 5.1:

$$\dfrac{\dfrac{}{\mathsf{L} \blacktriangleright [\,]} \qquad \dfrac{\dfrac{\vdots}{[z \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash (\lambda x.\,\lambda y.\,x)\ \mathsf{true} : (\widehat{\tau}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{bool}^{\mathsf{L}})^{(\mathsf{S},\mathsf{S})}} \qquad \dfrac{}{[z \mapsto (\widehat{\tau}, \mathsf{S}, \mathsf{L})] \vdash z : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}}}{[z \mapsto (\widehat{\tau}, \mathsf{S}, \mathsf{L})] \vdash ((\lambda x.\,\lambda y.\,x)\ \mathsf{true}) \bullet z : \mathsf{bool}^{(\mathsf{S},\mathsf{L})}}}{[\,] \vdash \lambda z.\,((\lambda x.\,\lambda y.\,x)\ \mathsf{true} \bullet z) : (\widehat{\tau}^{\mathsf{L}} \xrightarrow{\mathsf{S}} \mathsf{bool}^{\mathsf{L}})^{(\mathsf{S},\mathsf{L})}}.$$

Even though the subterm $(\lambda x.\,\lambda y.\,x)\ \mathsf{true}$ produces a lazy function, reflected by its relevance type $\widehat{\tau}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{bool}^{\mathsf{L}}$, the argument z of the strict application $((\lambda x.\,\lambda y.\,x)\ \mathsf{true}) \bullet z$ is still assigned the relevance annotation S and so the term as a whole can be recognised as producing a strict function, i.e., a function of relevance type $\widehat{\tau}^{\mathsf{L}} \xrightarrow{\mathsf{S}} \mathsf{bool}^{\mathsf{L}}$.

To illustrate that our refined system handles strict applications correctly, reconsider term (2),

$$\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)),$$

for which an unsafe derivation was given in Sect. 5.2. As the following derivation shows, the inner abstraction $\lambda z.\,x$ 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. That is, writing $\widehat{\tau}_{\lambda y.0}$ as an abbreviation for the annotated type

$$(\widehat{\tau}_0^{\mathsf{L}} \xrightarrow{\mathsf{L}} \widehat{\tau}^{\mathsf{L}})^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{nat}^{\mathsf{L}},$$

we have

$$\dfrac{\dfrac{}{\mathsf{L} \blacktriangleright [\,]} \qquad \dfrac{\dfrac{\vdots}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash \lambda y.\,0 : \widehat{\tau}_{\lambda y.0}^{(\mathsf{S},\mathsf{S})}} \qquad \dfrac{\dfrac{\mathsf{L} \blacktriangleright [\,]}{\mathsf{L} \blacktriangleright [x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})]} \qquad \dfrac{\dfrac{\dfrac{}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash x : \widehat{\tau}^{(\mathsf{L},\mathsf{L})}}}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash x : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}}}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})][z \mapsto (\widehat{\tau}_0, \mathsf{L}, \mathsf{L})] \vdash x : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}}}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash \lambda z.\,x : (\widehat{\tau}_0^{\mathsf{L}} \xrightarrow{\mathsf{L}} \widehat{\tau}^{\mathsf{L}})^{(\mathsf{S},\mathsf{L})}}}{[x \mapsto (\widehat{\tau}, \mathsf{L}, \mathsf{L})] \vdash (\lambda y.\,0) \bullet (\lambda z.\,x) : \mathsf{nat}^{(\mathsf{S},\mathsf{L})}}}{[\,] \vdash \lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x)) : (\widehat{\tau}^{\mathsf{L}} \xrightarrow{\mathsf{L}} \mathsf{nat}^{\mathsf{L}})^{(\mathsf{S},\mathsf{L})}}.$$

In particular, the variable x can now no longer be considered relevant to the body of the outer abstraction $\lambda x.\,((\lambda y.\,0) \bullet (\lambda z.\,x))$ and so we can no longer derive that the function produced by this abstraction is strict.

6.4 Properties

The transformations that arise from the revised system can be shown to preserve the behaviour of programs. To this end, let us write $\lfloor t \rfloor$ for the term that is obtained by turning all strict applications in t into nonstrict applications.

Theorem 1 (Correctness) *If $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}$, then* (1) $\lfloor t \rfloor = \lfloor t' \rfloor$, *and* (2) *if $t \longrightarrow w$ for some weak head normal form w, then there exists a weak head normal form w', such that $t' \longrightarrow w'$.*

Proof (Sketch.) The first part follows from a trivial structural induction on a derivation of $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$.

For the second part, we first show that in our system well-typed terms can only go wrong if during evaluation the constant error shows up in head position. The property is then demonstrated by induction on a derivation of $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$. The only interesting case is the one for [*r-app*$_1$] for which we need as an auxiliary result that $\widehat{\Gamma} \vdash \lambda x.\, t_0 \rhd t' : (\widehat{\tau}_1^{\psi_1} \xrightarrow{\mathsf{S}} \widehat{\tau}_2^{\psi_2})^{(\varphi,\psi)}$ implies that during the evaluation of $[x \mapsto t_2]t_0$ the term t_2 is guaranteed to show up in head position.

The required result follows from a lemma for $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}$ that states that, for every x, $\widehat{\tau}_0$, φ_0, and ψ_0 with $\widehat{\Gamma}(x) = (\widehat{\tau}_0, \varphi_0, \psi_0)$, $\varphi_0 = \mathsf{S}$ implies that terms substituted for x in t show up in head position during evaluation, while $\psi_0 = \mathsf{S}$ implies that such terms show up in function position. □

To show that our transformation is indeed applicable to all terms that are well-typed in the underlying type system, including those that contain strict applications, 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 $\widehat{\Gamma}$.

Theorem 2 (Conservative Extension)

1. *If $\Gamma \vdash t : \tau$, then there exist $\widehat{\Gamma}$, t', $\widehat{\tau}$, φ, and ψ with $\lfloor \widehat{\Gamma} \rfloor = \Gamma$ and $\lfloor \widehat{\tau} \rfloor = \tau$ such that $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$.*
2. *If $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$, then $\lfloor \widehat{\Gamma} \rfloor \vdash t : \lfloor \widehat{\tau} \rfloor$ and $\lfloor \widehat{\Gamma} \rfloor \vdash t' : \lfloor \widehat{\tau} \rfloor$.*

Proof Trivial structural inductions on derivations of $\Gamma \vdash t : \tau$ and $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$, respectively. For the first part, we simply annotate all terms and bindings with L. □

7 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 the article by Schmidt-Schauß et al. [33], who consider the semantics of *seq* in a safety proof for the strictness analysis of Nöcker [29], 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 [41]. Examples of the application of substructural typing to program analyses other than strictness analysis include work on uniqueness analysis [37] and sharing analysis [15]. The context-split operation has been attributed to Cervesato and Pfenning [8] and shows up in recent formulations of uniqueness analysis [14, 38].

Looking through a Curry-Howard lens, relevance type systems are connected to relevance logics [2, 3, 30]. Relevance has been put to use as an approximation of strictness by several authors, most prominently Wright [42], Baker-Finch [4], and Amtoft [1]. 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* [5] 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 to strictness analysis that keep track of totality rather than relevance are given by Kuo and Mishra [25], Jensen [21, 22], Benton [6], Solberg Gasser [35], Glynn et al. [13], and Coppo et al. [9]. Of these, only the formulations of Glynn et al. and Coppo et al. can distinguish between diverging functions and functions that produce diverging results. Hence, for the others, when applied to languages with explicit strictness annotations, the drawbacks reported 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. However, as this paper specifically deals 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 [7, 27, 39] or projection analyses [10, 16, 19].

An elaborate exposition of the rôle played by *seq* in the design of Haskell is given by Hudak et al. [18]. In the process, they point out an interesting point in the design space: 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 [34], 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 [23]. Van Eekelen and De Mol [11] 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.

8 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 yet applicable to such languages 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 [28]—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. The authors are currently planning to have the described approach implemented in a full-scale Haskell compiler; the resulting implementation may then serve as a basis for benchmarks and quantitative comparisons with other realisations of strictness analysis.

While the correctness property that was stated in Sect. 6 captures the main property of a strictness analyser, i.e., that changing nonstrict applications into strict applications does not worsen the termination behaviour of the program under analysis, a stronger property is desirable. We would like to make explicit that our transformation preserves the semantics of terms. That is, if $\widehat{\Gamma} \vdash t \triangleright t' : \widehat{\tau}^{(\mathsf{S},\mathsf{L})}$, then t and t' are contextually equivalent [26].

Another direction for future work follows from the observation that 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.

Acknowledgements This work was generously supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and initiated while the first author was employed at Utrecht University.

Some early results were presented under the title "Spreading the Joy" on June 3rd, 2009 at the 2009 Symposium on Trends in Functional Programming in Komárno, Slovakia. The present article is a revised version of a paper [17] that was presented on January 19th, 2010 at the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation in Madrid, Spain. The authors wish to express their gratitude to the anonymous reviewers of TFP 2009 and PEPM 2010, and the reviewers and editors of the current volume for their insightful comments and helpful suggestions.

That S and L may serve as mnemonics not only for "strict" and "lazy", but also for "small" and "large" was pointed out to us by Jeroen Fokker.

References

1. Amtoft, T.: Minimal thunkification. In: Cousot, P., Falaschi, M., Filé, G., Rauzy, A. (eds.) Static Analysis, Third International Workshop, WSA'93, Proceedings, Padova, Italy, 22–24 September 1993. Lecture Notes of Computer Science, vol. 724, pp. 218–229. Springer, Berlin (1993)
2. Anderson, A.R., Belnap, N.D. Jr.: Entailment: The Logic of Relevance and Necessity, vol. 1. Princeton University Press, Princeton (1975)
3. Anderson, A.R., Belnap, N.D. Jr., Dunn, J.M.: Entailment: The Logic of Relevance and Necessity, vol. 2. Princeton University Press, Princeton (1992)
4. Baker-Finch, C.A.: Relevant logic and strictness analysis. In: Billaud, M., Castéran, P., Corsini, M.-M., Musumbu, K., Rauzy, A. (eds.) Actes WSA'92 Workshop on Static Analysis (Bordeaux), Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings, 23–25 September 1992. Series Bigre, vols. 81–82, pp. 221–228. Atelier Irisa, Rennes (1992)
5. Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: Needed reduction and spine strategies for the lambda calculus. Inf. Comput. **75**(3), 191–231 (1987)

6. Benton, N.: Strictness analysis of lazy functional programs. Ph.D. thesis, University of Cambridge (1992)
7. Burn, G.L., Hankin, C., Abramsky, S.: The theory of strictness analysis for higher order functions. In: Ganzinger, H., Jones, N.D. (eds.) Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, 17–19 October 1985. Lecture Notes in Computer Science, vol. 217, pp. 42–62. Springer, Berlin (1986)
8. Cervesato, I., Pfenning, F.: A linear logical framework. Inf. Comput. **179**(1), 19–75 (2002)
9. Coppo, M., Damiani, F., Giannini, P.: Strictness analysis, totality, and non-standard-type inference. Theor. Comput. Sci. **272**(1–2), 69–112 (2002)
10. Davis, K., Wadler, P.: Backwards strictness analysis: Proved and improved. In: Davis, K., Hughes, J. (eds.) Functional Programming, Proceedings of the 1989 Glasgow Workshop, Workshops in Computing, 21–23 August 1989, Fraserburgh, Scotland, UK, pp. 12–30. Springer, Berlin (1990)
11. van Eekelen, M., de Mol, M.: Proof tool support for explicit strictness. In: Butterfield, A., Grelck, C., Huch, F. (eds.) Implementation and Application of Functional Languages, 17th International Workshop, Revised Selected Papers, IFL 2005, Dublin, Ireland, September 19–21. Lecture Notes in Computer Science, vol. 4015, pp. 37–54. Springer, Berlin (2006)
12. Gill, A., Launchbury, J., Jones, S.P.: A short cut to deforestation. In: FPCA'93 Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, 9–11 June 1993, pp. 223–232. ACM Press, New York (1993)
13. Glynn, K., Stuckey, P.J., Sulzmann, M.: Effective strictness analysis with HORN constraints. In: Cousot, P. (ed.) Static Analysis, 8th International Symposium, SAS 2001, Proceedings, Paris, France, 16–18 July 2001. Lecture Notes in Computer Science, vol. 2126, pp. 73–92. Springer, Berlin (2001)
14. Hage, J., Holdermans, S.: Heap recycling for lazy languages. In: Hatcliff, J., Glück, R., de Moor, O. (eds.) Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'08, San Francisco, California, USA, 7–8 January 2008, pp. 189–197. ACM Press, New York (2008)
15. Hage, J., Holdermans, S., Middelkoop, A.: A generic usage analysis with subeffect qualifiers. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, 1–3 October 2007, pp. 235–246. ACM Press, New York (2007)
16. Hinze, R.: Projection-based strictness analysis: theoretical and practical aspects. Ph.D. thesis, Bonn University (1995)
17. Holdermans, S., Hage, J.: Making "stricterness" more relevant. In: Gallagher, J.P., Voigtländer, J. (eds.) Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, 18–19 January 2010, pp. 121–130. ACM Press, New York (2010)
18. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A history of Haskell: Being lazy with class. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference, HOPL-III, San Diego, California, USA, 9–10 June 2007, pp. 1–55. ACM Press, New York (2007)
19. Hughes, J.: Backwards analysis of functional programs. In: Björner, A., Jones, N.D., Ershov, A.P. (eds.) Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18–24 Oct. 1987, pp. 187–208. North-Holland, Amsterdam (1988)
20. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989)
21. Jensen, T.P.: Strictness analysis in logical form. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Proceedings, Cambridge, MA, USA, 26–30 August 1991, pp. 352–366. Springer, Berlin (1991)
22. Jensen, T.P.: Inference of polymorphic and conditional strictness properties. In: POPL'98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 19–21 January 1998, San Diego, CA, USA, pp. 209–221. ACM Press, New York (1998)
23. Johann, P., Voigtländer, J.: The impact of *seq* on free theorems-based program transformations. Fundam. Inform. **69**(1–2), 63–102 (2006)
24. Kahn, G.: Natural semantics. In: Brandenburg, F.-J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings, Passau, Germany, 19–21 February 1987. Lecture Notes in Computer Science, vol. 247, pp. 22–39 (1987)
25. Kuo, T.-M., Mishra, P.: 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, pp. 260–272. ACM Press, New York (1989)
26. Morris, J.: Lambda-calculus models of programming languages. Ph.D. thesis, Massachusetts Institute of Technology (1968)
27. Mycroft, A.: The theory and practice of transforming call-by-need into call-by-value. In: Robinet, B. (ed.) International Symposium on Programming, Proceedings of the Fourth 'Colloque International sur la

Programmation', Paris, France, 22–24 April 1980. Lecture Notes in Computer Science, vol. 83, pp. 269–281. Springer, Berlin (1980)

28. Nielson, F., Nielson, H.R.: Type and effect systems. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design, Recent Insight and Advances (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel). Lecture Notes in Computer Science, vol. 1710, pp. 114–136. Springer, Berlin (1999)
29. Nöcker, E.: Strictness analysis using abstract reduction. In: FPCA'93 Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 9–11 June 1993, pp. 255–265. ACM Press, New York (1993)
30. Orlov, I.E.: Ischislenie sovmestimosti predlozhenii. Mat. Sb. **35**(3–4), 263–286 (1928)
31. Jones, S.P. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
32. Plasmeijer, R., van Eekelen, M.: Concurrent Clean language report—version 1.3. Technical Report CSI-R9816, University of Nijmegen (1998)
33. Schmidt-Schauß, M., Sabel, D., Schütz, M.: Safety of Nöcker's strictness analysis. J. Funct. Program. **18**(4), 503–551 (2008)
34. Seidel, D., Voigtländer, J.: Refined typing to localize the impact of forced strictness on free theorems. Acta Inform. **48**(3), 191–211 (2011)
35. Gasser, K.L.S., Nielson, H.R., Nielson, F.: Strictness and totality analysis. Sci. Comput. Program. **31**(1), 113–145 (1998)
36. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. J. Funct. Program. **2**(3), 245–271 (1992)
37. de Vries, E.: Making uniqueness typing less unique. Ph.D. thesis, Trinity College Dublin (2008)
38. de Vries, E., Plasmeijer, R., Abrahamson, D.: Uniqueness typing simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) Implementation and Application of Functional Languages, 19th International Symposium, Revised Selected Papers, IFL 2007, Freiburg, Germany, September 2007. Lecture Notes in Computer Science, vol. 5083, pp. 201–218. Springer, Berlin (2008)
39. Wadler, P.: Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In: Abramsky, S., Hankin, C. (eds.) Abstract Interpretation of Declarative Languages, pp. 266–275. Ellis Horwood, Chichester (1987)
40. Wadler, P.: Theorems for free! In: FPCA'89 Conference on Functional Programming and Computer Architecture, Imperial College, London, England, 11–13 September 1989, pp. 347–359. ACM Press, New York (1989)
41. Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages. The MIT Press, Cambridge (2005)
42. Wright, D.A.: A new technique for strictness analysis. In: Abramsky, S., Maibaum, T. (eds.) TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development. Vol. 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD), Brighton, UK, 8–12 April 1991. Lecture Notes in Computer Science, vol. 494, pp. 235–258. Springer, Berlin (1991)