



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Making “Stricterness” More Relevant

Stefan Holdermans

(Joint work with Jurriaan Hage)

Dept. of Information and Computing Sciences, Utrecht University

Centrumgebouw Noord, room B109

Padualaan 14, 3584 CH Utrecht

Phone: (030) 253 32 61

E-mail: stefan@cs.uu.nl

Web pages: <http://people.cs.uu.nl/stefan/>

Software Technology Colloquium

January 28, 2010

About this work



- ▶ Presented January 19th, 2010 at the the **Workshop on Partial Evaluation and Program Manipulation** (PEPM 2010) in Madrid, Spain.
- ▶ 45 papers submitted, 18 accepted (40%).

Other accepted papers from Utrecht:

- ▶ Corrective Hints for Type Incorrect Generic Java Programs (Nabil el Boustani and Jurriaan Hage);
- ▶ Optimizing Generics Is Easy! (José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh).

This paper available from <http://people.cs.uu.nl/stefan/pubs/holdermans10making.html>.





Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Making “Stricterness” More Relevant

Stefan Holdermans

(Joint work with Jurriaan Hage)

Dept. of Information and Computing Sciences, Utrecht University

Centrumgebouw Noord, room B109

Padualaan 14, 3584 CH Utrecht

Phone: (030) 253 32 61

E-mail: stefan@cs.uu.nl

Web pages: <http://people.cs.uu.nl/stefan/>

Software Technology Colloquium

January 28, 2010


What is “strictness”?

Making Haskell programs more strict by using the built-in function *seq*,

$$\text{seq} :: \alpha \rightarrow \beta \rightarrow \beta$$

which forces the evaluation of its first argument:

$$\text{seq } x \ y = \begin{cases} \perp & \text{if } x = \perp, \\ y & \text{otherwise} \end{cases}$$

 Evaluation: reducing a term to weak-head normal form.



Example: a stricter const

$const, const' :: \alpha \rightarrow \beta \rightarrow \alpha$

$const\ x\ y = x$ -- strict in x, lazy in y

$const'\ x\ y = seq\ y\ x$ -- strict in x and y

Interactive session:

```
Main> const π (error ";Ayuda!")
```

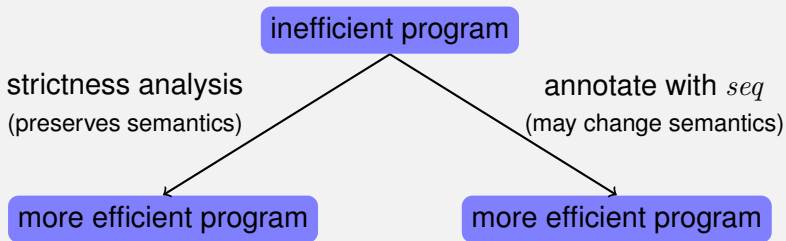
```
3.141592653589793
```

```
Main> const' π (error ";Ayuda!")
```

```
*** Exception: ;Ayuda!
```



Why do we have seq?



Stricterness propagates

Stricterness propagates through function application:

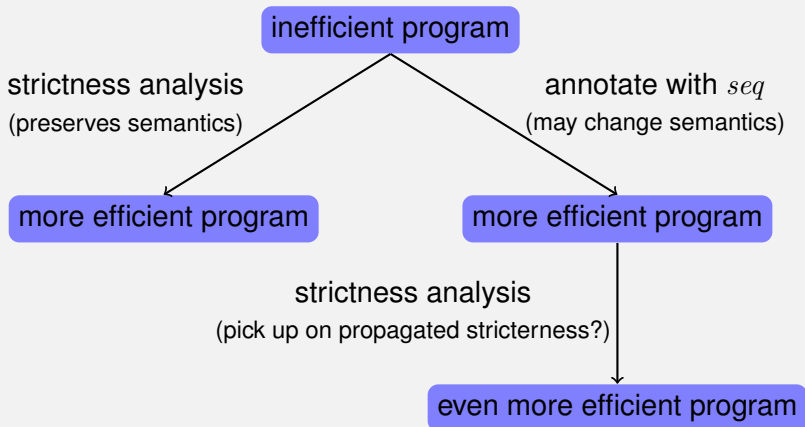
$$\begin{aligned} \text{force} &:: \alpha \rightarrow () \\ \text{force } x &= \text{const}' () x \end{aligned}$$

☞ *const'* is strict due to its use of *seq*, *force* is strict due to its use of *const'*.

In general: making a function stricter by means of *seq*, may cause several other functions to become stricter as well.



Taking advantage of seq?



Fun with seq

Without *seq*, we cannot tell $\lambda x \rightarrow \perp$ and \perp apart.

But with *seq*, we can:

```
Main> seq  $\perp$   $\pi$   
*** Exception:  $\perp$   
Main> seq ( $\lambda x \rightarrow \perp$ )  $\pi$   
3.141592653589793
```

- ☞ Evaluating a function: reducing it until a lambda appears at top-level.



Metaprogrammers should be seq-aware

The presence of *seq* asks for carefulness when reasoning about Haskell programs or implementing compiler optimisations: **eta-equivalence** does not hold, **parametricity** does not hold, **fold-build** fusion is invalid, ...

See Danielsson et al. (2006), Van Eekelen and De Mol (2006), Johann and Voigtländer (2006), ...


This talk: consequences for **strictness analysis** by means of **relevance typing**.



Relevance typing

- ▶ Type-based analysis for keeping track of **relevance**.
- ▶ See Wright (1991), Baker-Finch (1992), Amtoft (1993), Benton (1996), ...
- ▶ Connections with relevance logics.

Key idea: A variable x is relevant to an expression e , if any expression bound to x is **guaranteed** to be evaluated whenever e is evaluated.

 Goal: for a given expression, identify as many relevant variables as possible.



Refining function space

We use information about the relevance of variables to determine whether or functions are strict.

Information about the strictness of functions is stored in their types. We distinguish between

- ▶ strict function space, $\tau_1 \xrightarrow{S} \tau_2$, and
- ▶ (possibly) lazy function space, $\tau_1 \xrightarrow{L} \tau_2$.

☞ More appropriate: relevant function space, (possibly) irrelevant function space.



Interaction between relevance and strictness

Strictness is determined by relevance and vice versa.

- ▶ If the formal parameter x of an abstraction $\lambda x \rightarrow e$ is relevant to its body e , then $\lambda x \rightarrow e$ is strict (and, hence, gets a type of the form $\tau_1 \xrightarrow{S} \tau_2$).
- ▶ If a function expression e_1 is strict (i.e., has a type of the form $\tau_1 \xrightarrow{S} \tau_2$), then all variables that are relevant to an argument e_2 are relevant to a function application $e_1 e_2$.



Example: typing const

$$\begin{aligned} \text{const} &:: \alpha \xrightarrow{S} \beta \xrightarrow{L} \alpha \\ \text{const } x \ y &= x \end{aligned}$$

- ▶ x is relevant to x .
- ▶ y is irrelevant to x .

☞ Really: x is relevant to $\lambda y \rightarrow x$.



Call-by-value transformation

With *seq*, we can define a call-by-value application:


$$\begin{aligned} (\$!) &:: (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\ f \$! x &= \text{seq } x (f x) \end{aligned}$$

Idea: if a function expression e_1 is strict (i.e., has a type of the form $\tau_1 \xrightarrow{S} \tau_2$), replace all function applications $e_1 e_2$ by $e_1 \$! e_2$.



Example: transforming applications of const

```
Main> const (2 * 3) 5
6
Main> :cbv const (2 * 3) 5
(const $! (2 * 3)) 5
Main> (const $! (2 * 3)) 5
6
```

 Call-by-value transformation is semantics-preserving.



Relevance typing is unsound for seq

Relevance typing crucially relies on the fact that **functions are only evaluated when applied to arguments**.

With *seq*, this is no longer true:

- ▶ Functions are evaluated when applied to arguments.
- ▶ Functions are evaluated when passed to *seq*.



Relevance w.r.t. lambda-abstractions

Without *seq*:

Variables (other than x) **that are relevant to e are also relevant to $\lambda x \rightarrow e$.**

For example: x is relevant $\lambda y \rightarrow x$. (And, hence, $\lambda x \rightarrow \lambda y \rightarrow x$ is strict in x).



Refined typing for seq

We expect:

$$seq :: \alpha \xrightarrow{S} \beta \xrightarrow{S} \beta$$

Then, we have:

$$(\$!) :: (\alpha \xrightarrow{\gamma} \beta) \xrightarrow{S} \alpha \xrightarrow{S} \beta$$
$$f \$! x = seq x (f x)$$



Example: passing functions to seq

Consider:

$$f :: \alpha \xrightarrow{S} \text{Float}$$
$$f\ x = \text{seq}\ (\lambda y \rightarrow x)\ \pi$$

- ▶ x is relevant to $\lambda y \rightarrow x$.
- ▶ x is relevant to $\text{seq}\ (\lambda y \rightarrow x)\ \pi$ (**because seq is strict!**).
- ▶ f is strict in x .

But is it?



Example: passing functions to seq (cont'd)

$$f :: \alpha \xrightarrow{S} \text{Float}$$
$$f\ x = \text{seq } (\lambda y \rightarrow x) \ \pi$$

```
Main> f ⊥  
3.141592653589793  
Main> :cbv f ⊥  
f $! ⊥  
Main> f $! ⊥  
*** Exception: ⊥
```

- ☞ Call-by-value transformation is no longer semantics-preserving.



Nonsolution: a more conservative typing for seq

Considering *seq* to be lazy in its first argument:

$$seq :: \alpha \xrightarrow{L} \beta \xrightarrow{S} \beta$$

This renders relevance typing (and, hence, call-by-value transformation) sound again.

However, we are not able to take advantage of strictness due to *seq*:

$$\begin{aligned} const' &:: \alpha \xrightarrow{S} \beta \xrightarrow{L} \alpha \\ const' \ x \ y &= seq \ y \ x \\ force \ x &:: \alpha \xrightarrow{L} () \\ force \ x &= const' \ () \ x \end{aligned}$$



Solution: applicativeness

We need to distinguish between two uses of functions: being applied to arguments, being passed to *seq*.

Idea: adapt the relevance type system so that it additionally keeps track of which functions are **guaranteed** to be applied to arguments.



Relevance w.r.t. lambda-abstractions (revisited)

Without *seq*: variables (other than x) that are relevant to e are also relevant to $\lambda x \rightarrow e$.

With *seq*: variables (other than x) that are relevant to e are also relevant to $\lambda x \rightarrow e$, **if $\lambda x \rightarrow e$ is guaranteed to be used applicatively.**

For example: x is relevant to $\lambda y \rightarrow x$, only if $\lambda y \rightarrow x$ is (eventually) applied to an argument.

Hence, $\lambda x \rightarrow \lambda y \rightarrow x$ is strict in x only if $\lambda x \rightarrow \lambda y \rightarrow x$ is (eventually) fully applied.



Interaction between relevance and applicativeness

We have seen how applicativeness determines relevance.

Applicativeness itself is, in turn, dependent on relevance:

- ▶ In an application $e_1 \ e_2$, the function e_1 is guaranteed to be used applicative if the application is guaranteed to be evaluated.
- ☞ Hence, the notion of relevance needs to be lifted to **sub-terms**.
- ☞ As relevance and applicativeness are mutually dependent, implementations need to compute these properties **simultaneously**.



Example: passing functions to seq (revisited)

Consider again:

$$f :: \alpha \xrightarrow{L} \text{Float}$$
$$f\ x = \text{seq}\ (\lambda y \rightarrow x)\ \pi$$

- ▶ x is relevant to $\lambda y \rightarrow x$, **if $\lambda y \rightarrow x$ is eventually used applicatively.**
- ▶ x is relevant to $\text{seq}\ (\lambda y \rightarrow x)\ \pi$, **if $\lambda y \rightarrow x$ is eventually used applicatively.**
- ▶ But: $\lambda y \rightarrow x$ is not used applicatively in the body of f .
- ▶ Hence, we cannot derive that f is strict in x .



Taking advantage of strictness

With applicativeness:

$$\begin{aligned} \text{const}' &:: \alpha \xrightarrow{S} \beta \xrightarrow{S} \beta \quad \text{-- if const' is (eventually) fully applied} \\ \text{const}' x y &= \text{seq } y x \end{aligned}$$

(In the actual type system, the side condition is encoded in the type.)

$$\begin{aligned} \text{force} &:: \alpha \xrightarrow{S} () \\ \text{force } x &= \text{const}' () x \end{aligned}$$

(Because const' is fully applied in the body of force .)



What's in the paper?

- ▶ Formalisation of relevance typing for a language without *seq*.
- ▶ Call-by-value transformation into a language with *seq*.
- ▶ Adaptation of relevance typing for the language with *seq*.
- ▶ Adaptation of the call-by-value transformation.
- ▶ Algorithm.

- ▶ Related work.
- ▶ Future work.



In summary

- ▶ Naïve relevance typing is unsound in the presence of *seq*.
- ▶ Easy to get it sound again, but at the expense of missing out opportunities for call-by-value transformation.
- ▶ Taking into account applicativeness yields a sound transformation that does take advantage of strictness.

