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
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: `stefan@cs.uu.nl`
Web pages: `http://people.cs.uu.nl/stefan/`

PEPM 2010
January 19, 2010

What is “stricterness”?

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

$$seq :: \alpha \to \beta \to \beta$$

which forces the evaluation of its first argument:

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

☞ 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*

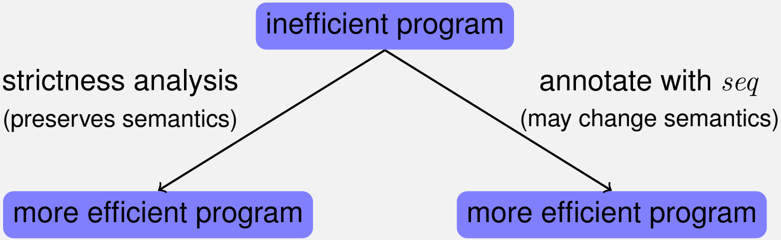
Interactive session:

```
Main>  const π (error "¡Ayuda!")
3.141592653589793
Main>  const' π (error "¡Ayuda!")
*** Exception: ¡Ayuda!
```


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Why do we have seq?



Stricterness propagates

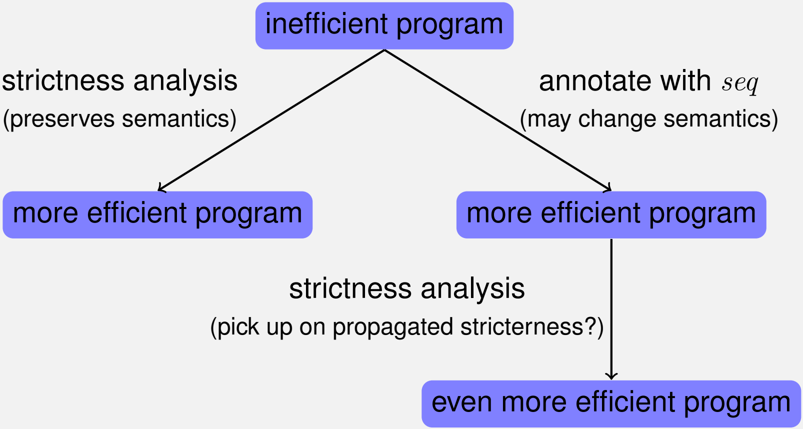
Stricterness propagates through function application:

$$force :: \alpha \to ()$$
$$force\ x = const'\ ()\ x$$

☞ $const'$ 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 $\lambda x \to \bot$ and $\bot$ apart.

But with seq, we can:

```
Main> seq ⊥ π
*** Exception: ⊥
Main> seq (λx → ⊥) π
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: consequenses 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 $\lambda x \to e$ is relevant to its body e, then $\lambda x \to e$ is strict (and, hence, gets a type of the form $\tau_1 \xrightarrow{\mathsf{S}} \tau_2$).
- If a function expression e_1 is strict (i.e., has a type of the form $\tau_1 \xrightarrow{\mathsf{S}} \tau_2$), then all variables that are relevant to an argument e_2 are relevant to a function application $e_1\ e_2$.

Example: typing const

$$const :: \alpha \xrightarrow{S} \beta \xrightarrow{L} \alpha$$
$$const\ x\ y = x$$

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

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

Call-by-value transformation

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

$$(\$!) :: (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$
$$f \ \$! \ x = seq \ x \ (f \ x)$$

Idea: if a function expression e_1 is strict (i.e., has a type of the form $\tau_1 \xrightarrow{\mathsf{S}} \tau_2$), 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** $\lambda x \to e$**.**

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

Refined typing for seq

We expect:

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

Then, we have:

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

Example: passing functions to seq

Consider:

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

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

But is it?

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Example: passing functions to seq (cont'd)

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

Main > $f\ \bot$
3.141592653589793

Main > :cbv $f\ \bot$
$f\ \$!\ \bot$

Main > $f\ \$!\ \bot$
*** Exception: $\bot$

☞ 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:

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

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

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

$$const' :: \alpha \xrightarrow{\mathsf{S}} \beta \xrightarrow{\mathsf{L}} \alpha$$
$$const'\ x\ y = seq\ y\ x$$

$$force\ x :: \alpha \xrightarrow{\mathsf{L}} ()$$
$$force\ x = const'\ ()\ x$$

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.

Example: passing functions to seq (revisited)

Consider again:

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

- x is relevant to $\lambda y \rightarrow x$, **if** $\lambda y \rightarrow x$ **is eventually used applicatively.**
- x is relevant to $\mathit{seq}\ (\lambda y \rightarrow x)\ \pi$, **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 stricterness

With applicativeness:

$$const' :: \alpha \xrightarrow{S} \beta \xrightarrow{S} \beta$$ -- *if const' is (eventually) fully applied*

$$const'\ x\ y = seq\ y\ x$$

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

$$force :: \alpha \xrightarrow{S} ()$$

$$force\ x = const'\ ()\ x$$

(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 stricterness.