Universiteit Utrecht

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

On the Rôle of Minimal Typing Derivations in Type-driven Program Transformation

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
February 25, 2010

About this work

- Paper to be presented at the Workshop on Language Descriptions, Tools, and Applications (LDTA 2010), Paphos, Cyprus, March 27–28, 2010.
- Two-day satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS 2010): “primary European forum for academic and industrial researchers working on topics relating to software science”.

Type-driven program transformation

Typically proceeds in two logical phases:

- Analysis: annotating a source program with types from a nonstandard type system capable of expressing certain properties of interest.
- Synthesis: using the annotations to drive the actual transformation into a target program.

Often establishes some form of program optimisation.

Dead-code elimination

doesn't use its 2nd argument

$$const :: \forall \alpha\ \beta.\ \alpha \to \beta \to \alpha$$
$$const\ x\ y = x$$

$$goldenRatio :: Double$$
$$goldenRatio =$$
$$const\ 1.618\ ((\lambda z \to z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2})\ 3.141)\ \bot$$

live code

dead code

☞ Transformation must be safe, i.e., semantics-preserving.

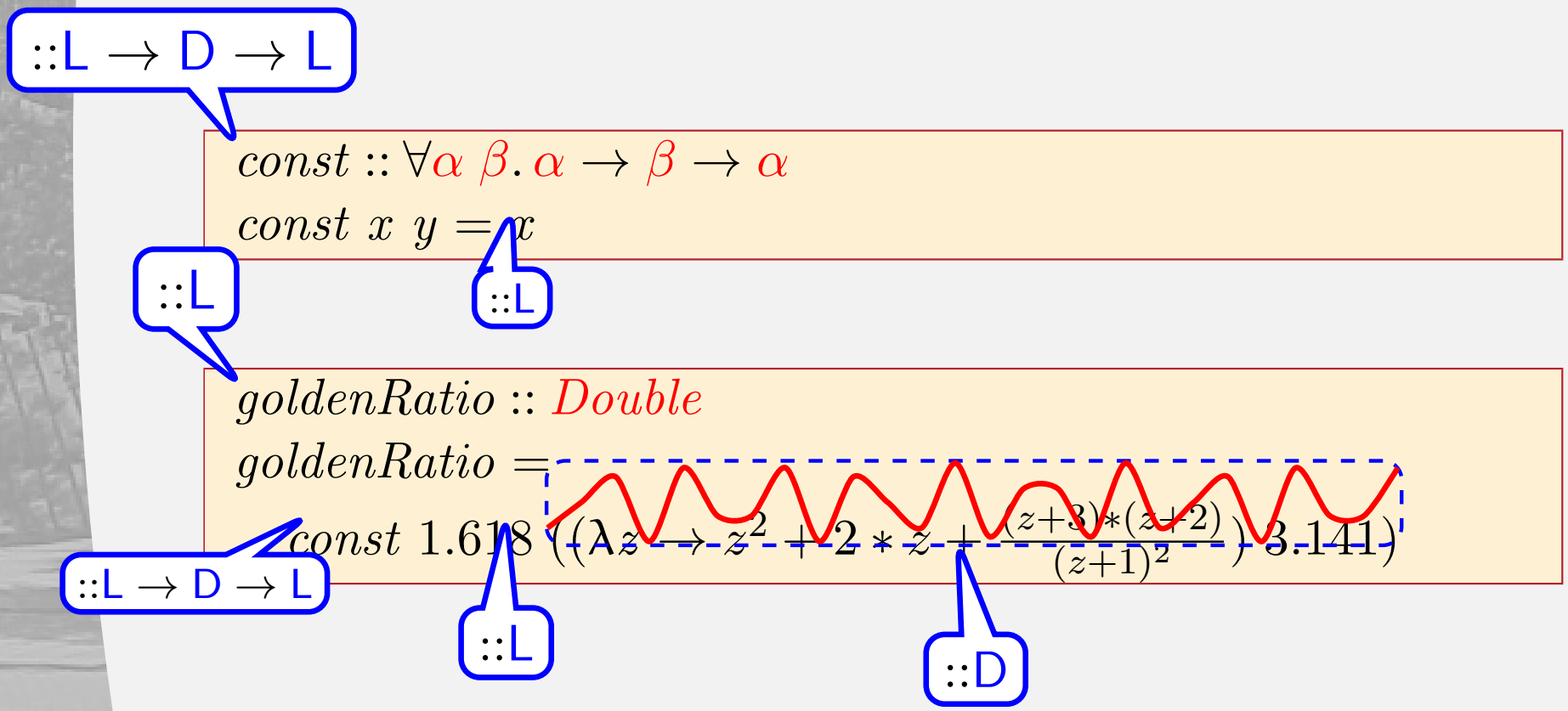
Type-driven DCE

Analysis: annotate the program with liveness types.

- D for code that is *guaranteed not to be evaluated*.
- L for code that *may be evaluated*.
- $\cdot \to \cdot$ for *functions*.

Synthesis: replace code with type D by $\bot$.

Type-driven DCE: example



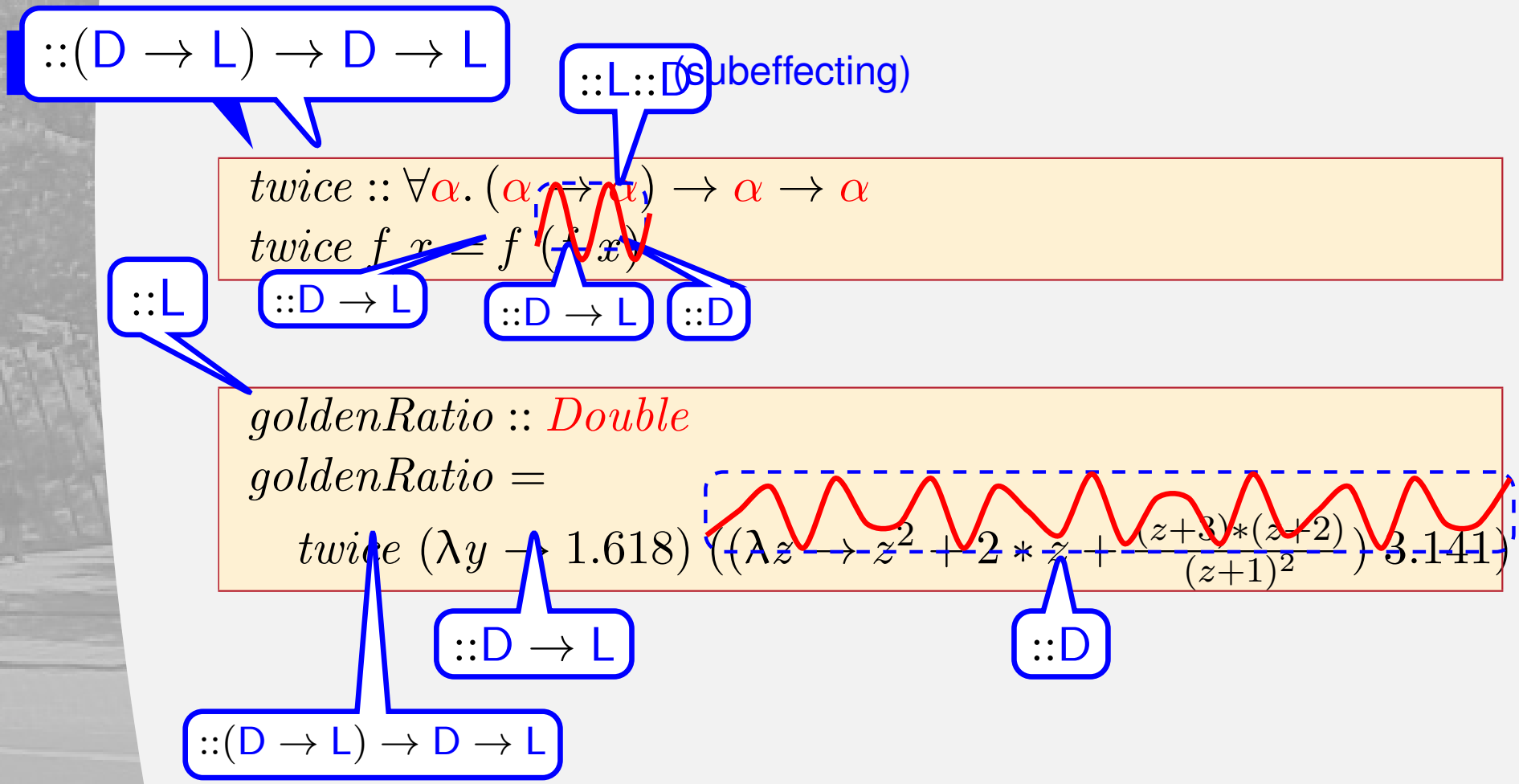
Subeffecting

- It is safe to silently "cast" an expression of type L to type D.
- In particular: live arguments can be bound to dead parameters.

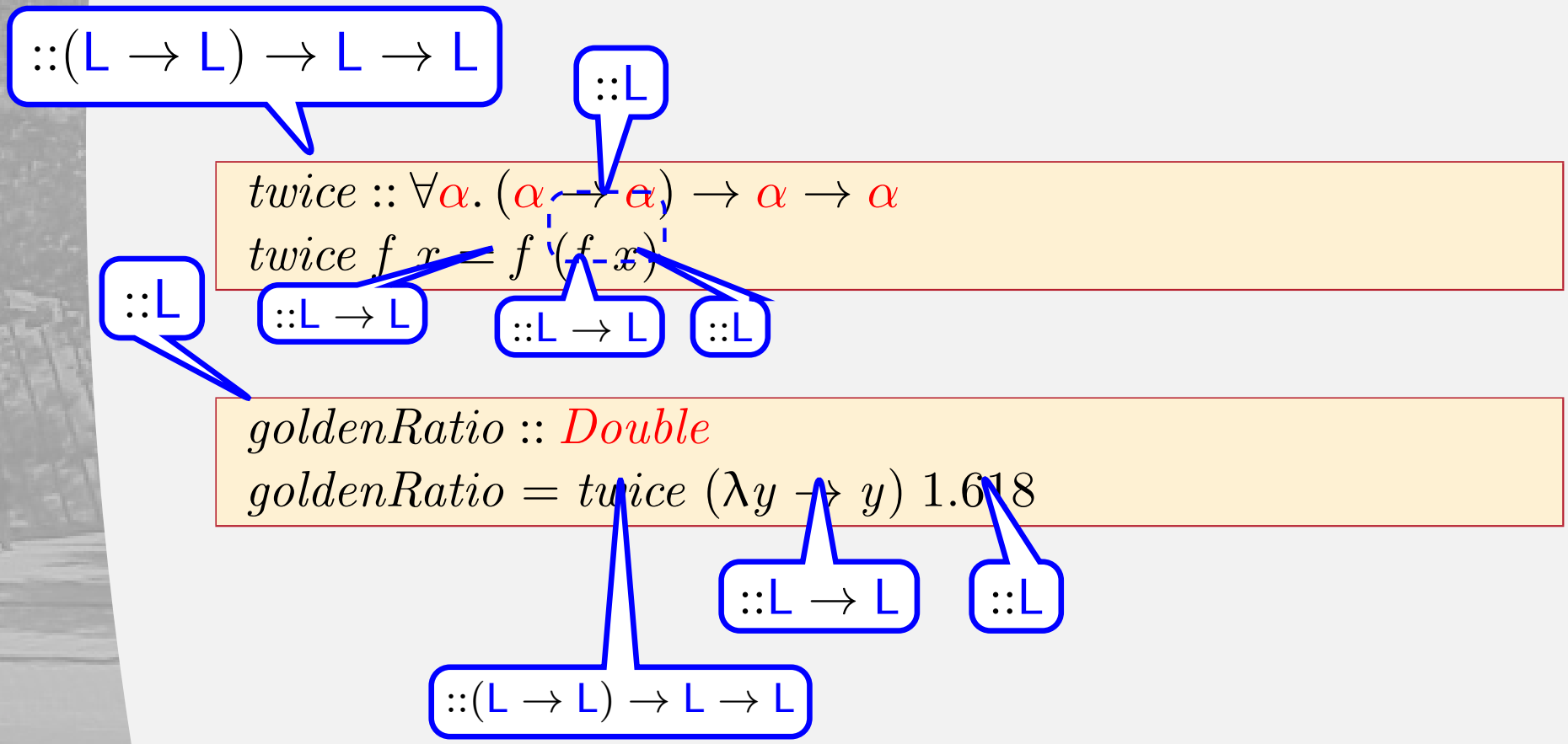
$$f\ x = const\ x\ x$$

- Akin to subtyping in object-oriented languages.

Subeffecting: example



Another HOF example



Modularity

What liveness type to assign to an HOF depends on how it's used.

$$twice\ (\lambda y \to 1.618)\ ((\lambda z \to z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2})\ 3.141)$$

gives $twice :: (\mathsf{D} \to \mathsf{L}) \to \mathsf{D} \to \mathsf{L}$.

$$twice\ (\lambda y \to y)\ 1.618$$

gives $twice :: (\mathsf{L} \to \mathsf{L}) \to \mathsf{L} \to \mathsf{L}$.

But what if we require separate compilation?

☞ The uses of an exported function may not be known at compile-time.

Pessimisation

(Wansbrough, 2002)

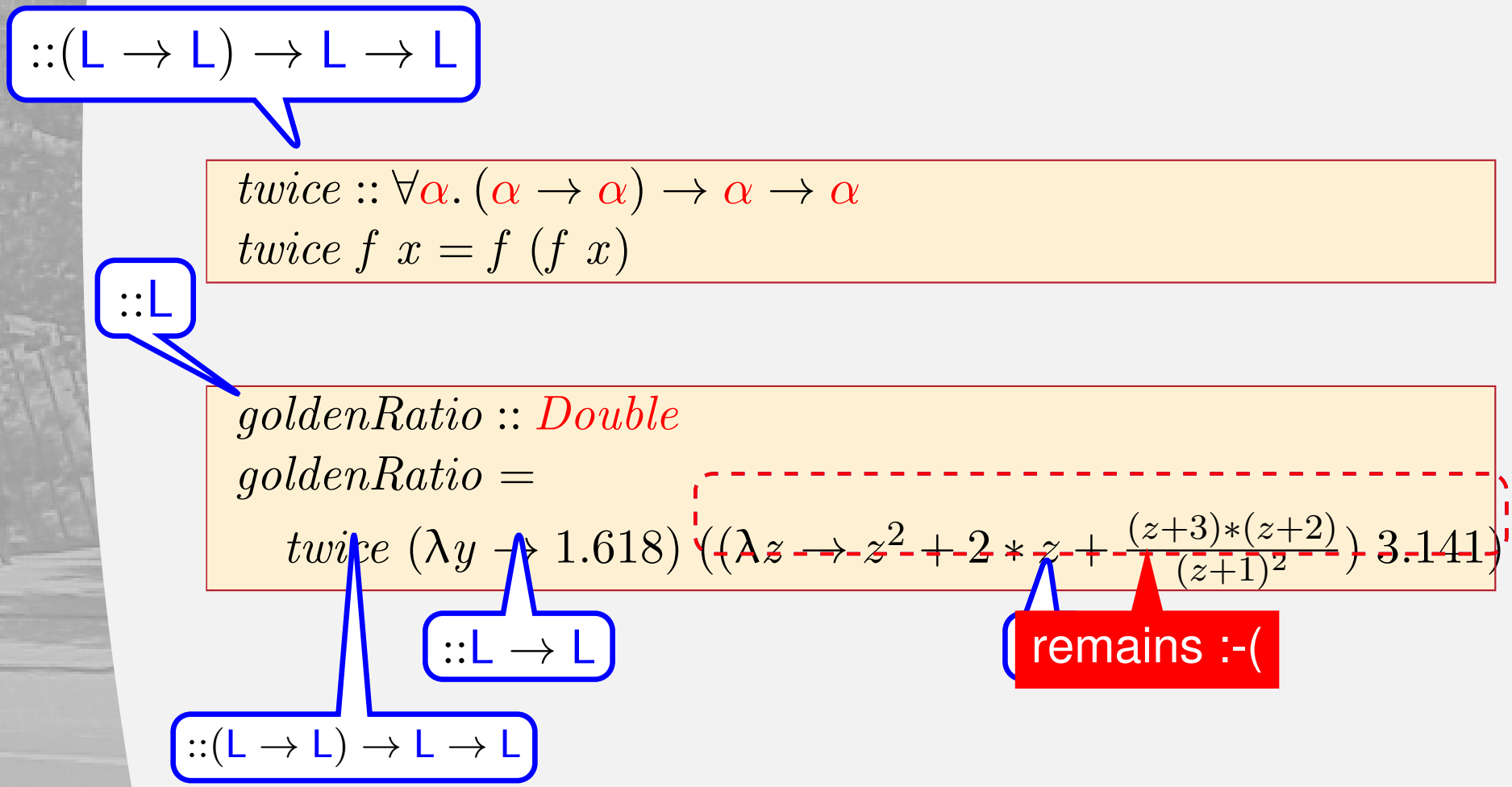
Assume that parameters of function type are to be bound to functions that may use all their arguments.

For example:

$$twice :: (\mathsf{L} \to \mathsf{L}) \to \mathsf{L} \to \mathsf{L}$$

☞ This is always safe, but pessimism typically propagets to use sites.

Pessimisation: example



Polyvariance

Allow liveness types to abstract over liveness properties.

That is, use polymorphic types as in ML or Haskell:

$$twice :: \forall\beta.\,(\beta \to \mathsf{L}) \to \beta \to \mathsf{L}$$

☞ Resulting transformation is polyvariant or context-sensitive.

Polyvariance: example

$::\forall\beta.\,(\beta \to \mathsf{L}) \to \beta \to \mathsf{L}$

still transformed pessimistically

$$twice :: \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$$
$$twice\ f\ x = f\ (f\ x)$$

::L

$$goldenRatio :: Double$$
$$goldenRatio =$$
$$twice\ (\lambda y \to 1.618)\ ((\lambda z \to z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2})\ 3.141)$$

$::\mathsf{D} \to \mathsf{L}$

::D

$::(\mathsf{D} \to \mathsf{L}) \to \mathsf{D} \to \mathsf{L}$ (instantiation)

Implementation

Type systems provide useful idioms for designing and defining analyses and transformations: subeffecting, polymorphism, . . .

What about implementing type-driven transformations?

It seems natural to adapt an off-the-shelf type-inference algorithm for Haskell-like languages.

But. . .

Principal types

Standard type-inference algorithms associate functions with their most polymorphic type.

For example:

$$twice :: \forall \beta_1\ \beta_2\ \beta_3\ \beta_4.\ (\beta_1 \to \beta_1 \sqcup \beta_2 \sqcup \beta_3) \to \beta_1 \sqcup \beta_4 \to \beta_2$$

$$\varphi_1 \sqcup \varphi_2 = \begin{cases} \mathsf{D}, & \text{if } \varphi_1 = \varphi_2 = \mathsf{D} \\ \mathsf{L}, & \text{otherwise} \end{cases}$$

☞ Principal types guarantee the highest degree of context-sensitivity.

Local functions

$$
\begin{array}{l}
goldenRatio = \\
\quad \textbf{let} \\
\quad\quad twice\ f\ x = f\ (f\ x) \\
\quad \textbf{in} \\
\quad\quad twice\ (\lambda y \to 1.618) \\
\quad\quad\quad ((\lambda z \to z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2})\ 3.141)
\end{array}
$$

- Assigning $twice$ its principal type means that the body of $twice$ is transformed pessimistically.
- Assigning $twice$ the monomorphic type $(\mathsf{D} \to \mathsf{L}) \to \mathsf{D} \to \mathsf{L}$ means that we eliminate the subexpression $(f\ x)$ from the body of $twice$.

Local functions (cont'd)

So, should local functions always have monomorphic types?

$$
\begin{array}{l}
goldenRatio = \\
\quad \textbf{let} \\
\quad\quad twice\ f\ x = f\ (f\ x) \\
\quad \textbf{in} \\
\quad\quad twice\ (\lambda y \to 1.000)\ 3.141 + twice\ (\lambda z \to z)\ 0.618
\end{array}
$$

- ☞ The only safe monomorphic type for $twice$ is $(\mathsf{L} \to \mathsf{L}) \to \mathsf{L} \to \mathsf{L}$, which prevents the elimination of 3.141.
- ☞ Poisoning: a single use with a "bad" type spreads to all use sites.

Strategy for higher-order functions

- If a closed-scope HOF is only applied to dead arguments, annotate the corresponding parameter with D. (Body can be optimised agressively.)
- If a closed-scope HOF is only applied to live arguments, annotate the corresponding parameter with L. (Nothing can be gained from annotating it polymorphically.)
- If a closed-scope HOF may be applied to both dead and live arguments, annotate the corresponding parameter polymorphically. (Avoids poisoning.)

Open-scope HOFs are always assigned their principal types.
(Ensures highest degree of safety and flexibility.)

Minimal typing derivations

(Bjørner, ML 1994)

A typing derivation for a given expression is minimal if no other typing derivation for the same expression would avoid type abstractions where the derivation under consideration could not.

☞ Type-driven polyvariant program transformations are best implemented with algorithms that compute MTDs rather than standard algorithms such as Algorithm W.

Conclusions

- Polymorphism: balancing modularity and precision.
- Minimal typing derivations: preventing pessimisation at definition sites of local functions.

- Some degree of flexibility in what constitutes a module.