

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

Stefan Holdermans

*Joint work with Jurriaan Hage (Utrecht University)*

Vector Fabrics Colloquium

April 9, 2010



# About this work

- Paper 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:

- 1 *Analysis*: annotating a source program with **types** from a nonstandard type system capable of expressing certain properties of interest.
- 2 *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 ::  $\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha$   
const x y = x
```

```
goldenRatio :: Double  
goldenRatio =
```

```
const 1.618 (( $\lambda z \rightarrow z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2}$ ) 3.141)
```

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

# Type-driven Dead-code Elimination

- 1 *Analysis*: annotate the program with **liveness types**.
  - Type **D** for code that is **guaranteed not to be evaluated**.
  - Type **L** for code that **may be evaluated**.
  - Types  $\cdot \rightarrow \cdot$  for **functions**.
  
- 2 *Synthesis*: replace code with type **D** by  $\perp$ .

# Type-driven Dead-code Elimination

## Example

$::L \rightarrow D \rightarrow L$

```
const ::  $\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha$   
const x y = x
```

$::L$

$::L$

```
goldenRatio :: Double  
goldenRatio =
```

```
const 1.618 (( $\lambda z \rightarrow z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2}$ ) 3.141)
```

$::L \rightarrow D \rightarrow L$

$::L$

$::D$

# 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.

$f\ x = \text{const } x\ x$

- Akin to **subtyping** in object-oriented languages.

# Subeffecting

## Example

$::(D \rightarrow L) \rightarrow D \rightarrow L$

$::D$  (subeffecting)

$\text{twice} :: \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$   
 $\text{twice } f \ x = f(f\ x)$

$::L$

$::D \rightarrow L$

$::D \rightarrow L$

$\text{goldenRatio} :: \text{Double}$

$\text{goldenRatio} =$

$\text{twice } (\lambda y \rightarrow 1.618) \left( \left( \lambda z \rightarrow z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2} \right) 3.141 \right)$

$::D \rightarrow L$

$::D$

$::(D \rightarrow L) \rightarrow D \rightarrow L$



# Higher-order Functions

## Another Example

$::(L \rightarrow L) \rightarrow L \rightarrow L$

`twice ::  $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$`   
`twice f x = f (f x)`

$::L$   $::L \rightarrow L$   $::L \rightarrow L$   $::L$

`goldenRatio :: Double`  
`goldenRatio = twice ( $\lambda y \rightarrow y$ ) 1.618`

$::L \rightarrow L$

$::L$

$::(L \rightarrow L) \rightarrow L \rightarrow L$

# Modularity

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

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

gives twice ::  $(D \rightarrow L) \rightarrow D \rightarrow L$ .

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

gives twice ::  $(L \rightarrow L) \rightarrow L \rightarrow L$ .

- 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.

`twice :: (L → L) → L → L`

- This is always **safe**, but pessimism typically **propagates to use sites**.

# Polyvariance

- Allow liveness types to abstract over liveness properties.
  - That is, use polymorphic types as in ML or Haskell:

`twice` ::  $\forall \beta. (\beta \rightarrow L) \rightarrow \beta \rightarrow L$

- Resulting transformation is **polyvariant** or **context-sensitive**.

# Polyvariance

## Example

$::\forall\beta. (\beta \rightarrow L) \rightarrow \beta \rightarrow L$

still transformed pessimistically

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

$::L$

$goldenRatio :: Double$   
 $goldenRatio =$

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

$::D \rightarrow L$

$::D$

$::(D \rightarrow L) \rightarrow D \rightarrow L$  (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**.

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

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

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

# Local Functions

```
goldenRatio =  
  let twice f x = f (f x)  
  in twice ( $\lambda y \rightarrow 1.618$ ) ( $(\lambda z \rightarrow z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2}) 3.141$ )
```

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



# Local Functions

(Continued)

- So, should **local functions** always have **monomorphic types**?

```
goldenRatio =
```

```
  let twice f x = f (f x)
```

```
  in twice ( $\lambda y \rightarrow 1.000$ ) 3.141 + twice ( $\lambda z \rightarrow z$ ) 0.618
```

- The only **safe** monomorphic type for twice is  $(L \rightarrow L) \rightarrow L \rightarrow L$ , which **prevents** the elimination of 3.141.
- **Poisoning**: a **single** use with a “bad” type affects **all** use sites (**Wansbrough and Peyton Jones, POPL 1999**).

# Strategy for Higher-order Functions

- **Open-scope** HOFs are always assigned their **principal types**. (Ensures highest degree of safety and flexibility.)
- If a **closed-scope** HOF is **only applied to dead arguments**, annotate the corresponding parameter with **D**. (Body can be optimised aggressively.)
- 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.)

# Minimal Typing Derivations

- A typing derivation for a given expression is **minimal** if no other derivation for the **same expression and typing** would **avoid type abstractions** where the derivation under consideration could not (**Bjørner, ML 1994**).
- Type-driven polyvariant program transformations are best implemented with algorithms that compute MTDs rather than standard algorithms such as Algorithm W.

# Flexibility w.r.t. Modularity

- For having transformations being driven by minimal typing derivations, it doesn't matter what exactly constitutes a **module**.
  - A module can be a single function, a binding group, a source file, a package, a whole program, . . .
- Even when performing a **whole-program analysis**, minimal typing derivations play an important rôle in avoiding **poisoning**.

# What's in the Paper?

- A complete formulation of a type-driven dead-code eliminator.
- Examples.
- Metatheory: **principal solutions** rather than principal types give a notion of “best” transformations.
- Not in the paper:
  - A **one-pass** algorithm for dead-code elimination. (Bjørner's algorithm requires **two** passes.)