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

Stefan Holdermans
Joint work with Jurriaan Hage (Utrecht University)

LDTA 2010
March 27, 2010

Paradijslaan 28
5611 KN Eindhoven
The Netherlands
E-mail: stefan@vectorfabrics.com
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

\[
\begin{align*}
\text{const} :: & \forall \alpha \beta. \alpha \to \beta \to \alpha \\
\text{const } x \ y &=\ x
\end{align*}
\]

goldenRatio :: Double
goldenRatio =

\[
\begin{align*}
\text{const } 1.618 \ ( (\lambda z \to z^2 + 2 \ast z + \frac{(z+3)\ast(z+2)}{(z+1)^2}) ) &= 3.141
\end{align*}
\]

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 $\cdot \to \cdot$ for **functions**.

2. **Synthesis**: replace code with type $D$ by $\bot$. 
Type-driven Dead-code Elimination
Example

\[
\text{const} :: \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha
\]
\[
\text{const } x y = x
\]

\[
\text{goldenRatio} :: \text{Double}
\]
\[
\text{goldenRatio} =
\]
\[
\text{const } 1.618 ( (\lambda z . z^2 + 2 \times z + \frac{(z+3) \times (z+2)}{(z+1)^2} ) \times 3.141 )
\]
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**

\[ \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \]

\[
twice f x = f(f(x))
\]

\[
goldenRatio :: Double
\]

\[
goldenRatio =
\]

\[
twice (\lambda y \rightarrow 1.618) ; ((\lambda z \rightarrow z^2 + 2 * z + \frac{(z+3)*(z+2)}{(z+1)^2}) \cdot 3.141)
\]
Higher-order Functions
Another Example

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

goldenRatio :: Double
goldenRatio = twice \((\lambda y \to y)\) 1.618
Modularity

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

\[
\text{twice} \ (\lambda y \to 1.618) \ ((\lambda z \to z^2 + 2 \times z + \frac{(z+3) \times (z+2)}{(z+1)^2}) \ 3.141)
\]
gives twice :: (D \to L) \to D \to L.

\[
\text{twice} \ (\lambda y \to y) \ 1.618
\]
gives twice :: (L \to L) \to L \to L.

- But what if we require separate compilation?
- The uses of an exported function may not be known at compile-time.
• Assume that parameters of function type are to be bound to functions that may use all their arguments.

\[ \text{twice} :: (L \rightarrow L) \rightarrow L \rightarrow L \]

• 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:

  \[
  \text{twice} :: \forall \beta. (\beta \to L) \to \beta \to L
  \]

- Resulting transformation is polyvariant or context-sensitive.
Polyvariance
Example

twice :: \( \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \)
twice \( f \ x = f (f \ x) \)
goldenRatio :: Double
goldenRatio =
twice (\lambda y \rightarrow 1.618) ((\lambda z \rightarrow z^2 + 2 \times z + \frac{(z+3) \times (z+2)}{(z+1)^2}) 3.141)

still transformed pessimistically

:: \( \forall \beta. (\beta \rightarrow \text{L}) \rightarrow \beta \rightarrow \text{L} \)
:: L
:: (D \rightarrow \text{L}) \rightarrow D \rightarrow \text{L} (instantiation)
:: D → L
:: D
• 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**.

\[
\text{twice} :: \forall \beta_1 \beta_2 \beta_3 \beta_4. (\beta_1 \rightarrow \beta_1 \sqcup \beta_2 \sqcup \beta_3) \rightarrow \beta_1 \sqcup \beta_4 \rightarrow \beta_2
\]

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

- Principal types guarantee the highest degree of context-sensitivity.
Local Functions

goldenRatio =
  let twice f x = f (f x)
  in twice (λy → 1.618) ((λz → 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 → L) → D → L means that we **eliminate** the subexpression (f x) from the body of twice.
• So, should local functions always have monomorphic types?

goldenRatio =

let twice f x = f (f x)
in  twice (λy → 1.000) 3.141 + twice (λz → 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.)
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.)