Type-Changing Rewriting and Semantics-Preserving Transformation

Johan Jeuring
Sean Leather   Andres Löh   Bram Schuur

Utrecht University

WG2.1 – March, 2014
Introduction

- Program transformation
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
  - Type-driven
Introduction

- Program transformation
  - With isomorphic types \( \mathcal{A} \) and \( \mathcal{R} \)
  - Type-safe
  - Type-driven
  - Semantics-preserving
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
Introduction

- Program transformation
  - With isomorphic types $A$ and $R$
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
  - Whole-program
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
  - Whole-program

- Formalization
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
  - Whole-program

- Formalization
  - Type-and-transform systems
Introduction

- Program transformation
  - With isomorphic types \( \mathcal{A} \) and \( \mathcal{R} \)
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
  - Whole-program

- Formalization
  - Type-and-transform systems
  - Small language
Introduction

- Program transformation
  - With isomorphic types $\mathcal{A}$ and $\mathcal{R}$
  - Type-safe
  - Type-driven
  - Semantics-preserving
  - Automated
  - Whole-program

- Formalization
  - Type-and-transform systems
  - Small language
  - Prototype implementation
A Haskell Program

quantify :: Int → String → String → String → String
quantify num a s word =
  if num ≡ 1 then a ++ " " ++ word else show num ++ " " ++ word ++ s

main = do
  putStrLn "How many velociraptors did you see?"
  s ← getline
  let n = read s
  putStrLn ("You saw " ++ quantify n "an" "s" "velociraptor" ++ "]")
  putStrLn (if n > 0 then "What? I though they were extinct." 
            else "Whew! That was close."
Appending Strings

- The `String` type in Haskell is a list of characters: `[Char]`.
Appending Strings

- The `String` type in Haskell is a list of characters: `[Char]`.
- The append operator `++` is right-associative and pattern-matches on the left.
Appending Strings

- The `String` type in Haskell is a list of characters: `[Char]`.
- The append operator `+` is right-associative and pattern-matches on the left.
- Applying `quantify` and unfolding it:

```
"You see " + quantify 5 "an" "s" "velociraptor" + "?
≡ "You see " + ("5" + " " + "velociraptor" + "s") + "?"
```
Appending Strings

- The `String` type in Haskell is a list of characters: `[Char]`.
- The append operator `+` is right-associative and pattern-matches on the left.
- Applying `quantify` and unfolding it:

  "You see " + quantify 5 "an" "s" "velociraptor" + "?"
  \equiv "You see " +("5" + " " + "velociraptor" + "s") + "?"

- The string "5 velociraptors" is effectively traversed twice, once by the `+` inside the `()` and once again by the last `+`.
Appending Strings

- The `String` type in Haskell is a list of characters: `[Char]`.
- The append operator `++` is right-associative and pattern-matches on the left.
- Applying `quantify` and unfolding it:
  
  "You see " ++ `quantify` 5 "an" "s" "velociraptor" ++ "?"
  ≡ "You see " ++ ("5" ++ " " ++ "velociraptor" ++ "s") ++ "?"

- The string "5 velociraptors" is effectively traversed twice, once by the `++` inside the `()` and once again by the last `++`.
- This is a well-known problem that can be avoided by “delaying” the `++` and appending functions (`String → String`) instead of `String s`. 
A Haskell Program (Revised)

\[
\begin{align*}
\textbf{type } S &= \text{String} \quad \text{-- for brevity} \\
\textbf{newtype } Z &= Z \ (S \to S) \\
(\Diamond) &: Z \to Z \to Z \\
Z f \Diamond Z g &= Z \ (f \circ g)
\end{align*}
\]

\[
\begin{align*}
\text{abs} &: Z \to S \\
\text{abs} \ (Z f) &= f \ "\ " \\
\text{rep} &: S \to Z \\
\text{rep} \ xs &= Z \ (xs \#) \\
\text{quantify} &: \text{Int} \to Z \to Z \to Z \to Z \\
\text{quantify num a s word} &= \\
\text{if } \ num \equiv 1 \text{ then } a \Diamond \text{rep} "\ " \Diamond \text{word} \text{ else } \text{rep} \ (\text{show num}) \Diamond \text{rep} "\ " \Diamond \text{word} \Diamond s
\end{align*}
\]

\[
\begin{align*}
\text{main} &= \ldots \\
\text{putStrLn} \ (\text{abs} \ (\text{rep} "\text{You saw }" \\
\quad \Diamond \text{quantify n (rep "an") (rep "s") (rep "velociraptor") \\
\quad \Diamond \text{rep } "?")))
\end{align*}
\]
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose
  We would like to use but not write the revised program. Let's transform the original instead.

Note that
- the types $S$ and $Z$ have an isomorphism (rep and abs assuming $Z$ is abstract), but
- their interfaces (construction and elimination) are not equivalent, so, consequently,
we cannot use simple term rewriting to transform the original program.
Observations

- Compared to the original program, the revised program is
  - more efficient
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose

Note that the types $S$ and $Z$ have an isomorphism (rep and abs assuming $Z$ is abstract), but their interfaces (construction and elimination) are not equivalent, so, consequently, we cannot use simple term rewriting to transform the original program.

Johan Jeuring, Sean Leather, Andres Löh, Bram Schuur (Utrecht University)
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose

- We would like to use but not write the revised program.
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose
- We would like to use but not write the revised program.
- Let’s transform the original instead.
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose
- We would like to use but not write the revised program.
- Let’s transform the original instead.
- Note that
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose
- We would like to use but not write the revised program.
- Let’s transform the original instead.
- Note that
  - the types $S$ and $Z$ have an isomorphism (rep and abs assuming $Z$ is abstract), but
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose

- We would like to use but not write the revised program.
- Let’s transform the original instead.
- Note that
  - the types $S$ and $Z$ have an isomorphism (\textit{rep} and \textit{abs} assuming $Z$ is abstract), but
  - their interfaces (construction and elimination) are not equivalent, so, consequently,
Observations

- Compared to the original program, the revised program is
  - more efficient
  - more verbose
- We would like to use but not write the revised program.
- Let’s transform the original instead.
- Note that
  - the types $S$ and $Z$ have an isomorphism ($rep$ and $abs$ assuming $Z$ is abstract), but
  - their interfaces (construction and elimination) are not equivalent, so, consequently,
  - we cannot use simple term rewriting to transform the original program.
Type-and-Transform Systems

- A *type-and-transform system* (TTS) defines:
A type-and-transform system (TTS) defines:

- for a given typed language
A *type-and-transform system* (TTS) defines:

- for a given typed language
- how to relate two programs such that
A type-and-transform system (TTS) defines:

- for a given typed language
- how to relate two programs such that
- the programs may have parts that differ in their terms and types but,
A type-and-transform system (TTS) defines:

- for a given typed language
- how to relate two programs such that
- the programs may have parts that differ in their terms and types but,
- as “complete” programs, both have the same type and are semantically equivalent.
Components of a TTS

transformation: a structure that relates a source and a target (an optionally modified version of the source)
Components of a TTS

**transformation:** a structure that relates a *source* and a *target* (an optionally modified version of the source)

**typed rewrite rule:** a tuple of left and right term patterns and types that describes a step of change in a target program.
Components of a TTS

**transformation**: a structure that relates a *source* and a *target* (an optionally modified version of the source)

**typed rewrite rule**: a tuple of left and right term patterns and types that describes a step of change in a target program.

![Diagram of components of a TTS](image)
Our Language

Syntax:

Terms: \( e, f \ ::= x \mid fe \mid \lambda x.e \mid \text{fix } e \mid \text{let } x = e_1 \text{ in } e_2 \)

Types: \( \tau, \nu \ ::= \alpha \mid B \mid \tau \rightarrow \nu \)

Type Schemes: \( \varsigma \ ::= \forall \vec{\alpha}.\tau \)

Environments: \( \Gamma \ ::= e \mid \Gamma, \nu : \varsigma \)

Variables: \( \nu \ ::= x \mid m \)
Our Language

Syntax:

Terms: 

\[ e, f ::= x \mid f \, e \mid \lambda x. \, e \mid \text{fix} \, e \mid \text{let} \, x = e_1 \, \text{in} \, e_2 \]

Types: 

\[ \tau, \nu ::= \alpha \mid B \mid \tau \rightarrow \nu \]

Type Schemes: 

\[ \zeta ::= \forall \bar{\alpha}. \tau \]

Environments: 

\[ \Gamma ::= \varepsilon \mid \Gamma, \nu : \zeta \]

Variables: 

\[ \nu ::= x \mid m \]

Hindley-Milner \textbf{let} -polymorphism typing
Our Language

- **Syntax:**

  Terms: \[ e, f \ := \ x \mid f\ e \mid \lambda x.e \mid \text{fix } e \mid \text{let } x = e_1 \text{ in } e_2 \]

  Types: \[ \tau, \nu \ := \ \alpha \mid B \mid \tau \rightarrow \nu \]

  Type Schemes: \[ \varsigma \ := \ \forall \alpha.\tau \]

  Environments: \[ \Gamma \ := \ e \mid \Gamma, \nu : \varsigma \]

  Variables: \[ \nu \ := \ x \mid m \]

- Hindley-Milner **let**-polymorphism typing

- Base types \( S \) and \( Z \)
Our Language

- **Syntax:**
  - Terms: 
    \[ e, f ::= x \mid f \cdot e \mid \lambda x. e \mid \text{fix} \ e \mid \text{let } x = e_1 \text{ in } e_2 \]
  - Types: 
    \[ \tau, \nu ::= \alpha \mid B \mid \tau \to \nu \]
  - Type Schemes: 
    \[ \zeta ::= \forall \vec{\alpha}. \tau \]
  - Environments: 
    \[ \Gamma ::= \varepsilon \mid \Gamma, \nu : \zeta \]
  - Variables: 
    \[ \nu ::= x \mid m \]

- Hindley-Milner **let**-polymorphism typing
- Base types **S** and **Z**
- Flexibility for infix operator syntax presentation
Introduction to Transformation

- Given some types $\mathcal{A}$ and $\mathcal{R}$ with an isomorphism witnessed by $abs :: \mathcal{R} \rightarrow \mathcal{A}$ and $rep :: \mathcal{A} \rightarrow \mathcal{R}$.
Introduction to Transformation

- Given some types $\mathcal{A}$ and $\mathcal{R}$ with an isomorphism witnessed by $\text{abs} :: \mathcal{R} \rightarrow \mathcal{A}$ and $\text{rep} :: \mathcal{A} \rightarrow \mathcal{R}$,

- we transform a program using $\mathcal{A}$-terms to a program where some $\mathcal{A}$-terms are replaced by $\mathcal{R}$-terms for some type $\mathcal{R}$. 
Introduction to Transformation

- Given some types $\mathcal{A}$ and $\mathcal{R}$ with an isomorphism witnessed by $\text{abs} :: \mathcal{R} \to \mathcal{A}$ and $\text{rep} :: \mathcal{A} \to \mathcal{R}$,

- we transform a program using $\mathcal{A}$-terms to a program where some $\mathcal{A}$-terms are replaced by $\mathcal{R}$-terms for some type $\mathcal{R}$.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a&quot; : S</td>
<td>rep &quot;a&quot; : Z</td>
</tr>
<tr>
<td>S → S → S</td>
<td>Z → Z → Z</td>
</tr>
<tr>
<td>x + &quot;b&quot; : S</td>
<td>x ◦ rep &quot;b&quot; : Z</td>
</tr>
<tr>
<td>(λx.x + &quot;b&quot;) &quot;a&quot; : S</td>
<td>abs ((λx.x ◦ rep &quot;b&quot;) (rep &quot;a&quot;)) : S</td>
</tr>
<tr>
<td>(λx.x + &quot;b&quot;) &quot;a&quot; : S</td>
<td>abs ((λx.rep x ◦ rep &quot;b&quot;) &quot;a&quot;) : S</td>
</tr>
<tr>
<td>(λx.x + &quot;b&quot;) &quot;a&quot; : S</td>
<td>(λx.abs (rep x ◦ rep &quot;b&quot;)) &quot;a&quot; : S</td>
</tr>
</tbody>
</table>
Foundation: Difunctors

- We need to relate type changes and term changes between programs.
We need to relate type changes and term changes between programs.

A difunctor is a mixed-variant binary type constructor $F$ with:

$$\text{dimap} : \forall a \ a' \ b \ b'. (a' \rightarrow b') \rightarrow (b \rightarrow a) \rightarrow F b' \ b \rightarrow F a' \ a$$

$$\text{dimap \ id \ id} \equiv \text{id}$$

$$\text{dimap} \ (g \circ h) \ (i \circ j) \equiv \text{dimap} \ h \ i \circ \text{dimap} \ g \ j$$
**Foundation: Difunctors**

- We need to relate type changes and term changes between programs.
- A *difunctor* is a mixed-variant binary type constructor $F$ with:

  \[
  \text{dimap} : \forall a \ a' \ b \ b'. (a' \to b') \to (b \to a) \to F b' \ b \to F a' \ a
  \]

  \[
  \text{dimap} \ id \ id \equiv id
  \]

  \[
  \text{dimap} \ (g \circ h) \ (i \circ j) \equiv \text{dimap} \ h \ i \circ \text{dimap} \ g \ j
  \]

- The contravariant parameter is necessary for relating variables in $\lambda$ and the typing environment of a (sub-)program.
Foundation: Type Functors

A type functor \( \hat{\tau} \) is a difunctor \( F \) where \( \hat{\tau}(a) = F a a \).

\[
\hat{\tau}, \hat{\upsilon} ::= \alpha \mid B \mid \hat{\tau} \Rightarrow \hat{\upsilon} \mid \text{I}
\]
A type functor $\tilde{\tau}$ is a difunctor $F$ where $\tilde{\tau}(a) = F a a$.

$$\tilde{\tau}, \tilde{\upsilon} ::= \alpha \mid B \mid \tilde{\tau} \rightarrow \tilde{\upsilon} \mid \iota$$

Project a type functor to a type:

$$\alpha / B(\upsilon) = \alpha / B$$

$$(\tilde{\tau} \rightarrow \tilde{\upsilon})(\upsilon) = \tilde{\tau}(\upsilon) \rightarrow \tilde{\upsilon}(\upsilon)$$

$$\iota(\upsilon) = \upsilon$$
Foundation: Type Functors

- A type functor $\hat{\tau}$ is a difunctor $F$ where $\hat{\tau}(a) = F \ a \ a$.

$$\hat{\tau}, \hat{\upsilon} ::= \ a \ | \ B \ | \ \hat{\tau} \to \hat{\upsilon} \ | \ I$$

- Project a type functor to a type:

$$\frac{\alpha \ / \ B \langle \upsilon \rangle}{\hat{\alpha} \ / \ B} = \alpha \ / \ B$$

$$(\hat{\tau} \to \hat{\upsilon}) \langle \upsilon \rangle = \hat{\tau} \langle \upsilon \rangle \to \hat{\upsilon} \langle \upsilon \rangle$$

$$I \langle \upsilon \rangle = \upsilon$$

- For brevity, we write the dimap for type functors as $D_{\hat{\tau}}$:

$$D_{\hat{\tau}} : \forall a \ b. (a \to b) \to (b \to a) \to \hat{\tau}(b) \to \hat{\tau}(a)$$

$$D_{\alpha \ / \ B} \ f \ g = id$$

$$D_{\hat{\tau} \to \hat{\upsilon}} f \ g = \lambda x \to D_{\hat{\upsilon}} f \ g \circ x \circ D_{\hat{\tau}} g f$$

$$D_{I} f \ g = g$$
A *type functor* $\tau$ is a difunctor $F$ where $\tau(a) = F a a$.

\[ \tau, \upsilon ::= \alpha \mid B \mid \tau \to \upsilon \mid \eta \]

Project a type functor to a type:

\[
\begin{align*}
\alpha / B \langle \upsilon \rangle &= \alpha / B \\
(\tau \to \upsilon) \langle \upsilon \rangle &= \tau \langle \upsilon \rangle \to \upsilon \langle \upsilon \rangle \\
\eta \langle \upsilon \rangle &= \upsilon
\end{align*}
\]

For brevity, we write the *dimap* for type functors as $D_\tau$:

\[
D_\tau : \forall a b. (a \to b) \to (b \to a) \to \tau(b) \to \tau(a)
\]

\[
\begin{align*}
D\alpha/B & \quad f \ g = id \\
D\tau/\upsilon & \quad f \ g = \lambda x \to D_\upsilon f \ g \circ x \circ D_\tau g \ f \\
D\eta & \quad f \ g = g
\end{align*}
\]

We also have type scheme and environment (di-)functors: $\xi$, $\Gamma$. 

Johan Jeuring  Sean Leather, Andres Löh, Bram Schuur (Utrecht University)
Typed Rewrite Rules

- A typed rewrite rule is a term rewrite rule extended with an environment and left and right type functors:

  Patterns \( p ::= v \mid p_1 p_2 \)

  Rules \( \rho ::= \Gamma \triangleright p_l: \tau_l \leadsto p_r: \tau_r \)

Rules for the example transformation:

\[ \Gamma \triangleright m : S \leadsto m : S; \]

\[ \text{rep } m : \iota \leadsto m : \iota; \]

\[ \text{abs } m : S; \epsilon \leadsto + + : S \rightarrow S \rightarrow S; \]

\[ \circ : \iota \rightarrow \iota \rightarrow \iota; \]
Typed Rewrite Rules

- A typed rewrite rule is a term rewrite rule extended with an environment and left and right type functors:

  \[
  \text{Patterns } p ::= v \mid p_1 p_2 \\
  \text{Rules } \rho ::= \Gamma \triangleright p_l : \tau_l \leadsto p_r : \tau_r
  \]

- Rules for the example transformation:

  \[
  \begin{align*}
  \Gamma \triangleright p_l : \tau_l & \leadsto p_r : \tau_r \\
  \{ m : S \} \triangleright m : S & \leadsto \text{rep } m : \iota \\
  \{ m : \iota \} \triangleright m : \iota & \leadsto \text{abs } m : S \\
  \varepsilon \triangleright \# : S \rightarrow S \rightarrow S & \leadsto \diamond : \iota \rightarrow \iota \rightarrow \iota
  \end{align*}
  \]
Typed Rewrite Rules (2)

Rules are typed by $\Gamma \vdash \rho :$

$\Gamma \cup \Gamma \langle R \rangle \vdash p_l : \tau_l \langle R \rangle$

$\Gamma \cup \Gamma \langle R \rangle \vdash p_r : \tau_r \langle R \rangle$

$\hat{\Gamma} \vdash \tau_l \langle A \rangle \equiv \tau_r \langle A \rangle$

$\Gamma \vdash (\hat{\Gamma} \triangleright p_l : \tau_l \leadsto p_r : \tau_r)$
Typed Rewrite Rules (2)

- Rules are typed by $\Gamma \vdash \rho :$

$$\Gamma \cup \Gamma \langle R \rangle \vdash p_l : \tau_l \langle R \rangle$$  $$\Gamma \cup \Gamma \langle R \rangle \vdash p_r : \tau_r \langle R \rangle$$  $$\Gamma \vdash \tau_l \langle A \rangle \equiv \tau_r \langle A \rangle$$  $$\Gamma \vdash (\Gamma \triangleright p_l : \tau_l \leadsto p_r : \tau_r)$$

- Each rule $\rho$ requires a term proof for $\Gamma \vdash \tau_l \langle A \rangle \equiv \tau_r \langle A \rangle :$

$$D_{\tau_l \text{rep abs}} (D_{\hat{\Gamma}} \text{rep abs})p_l \equiv D_{\tau_r \text{rep abs}} (D_{\hat{\Gamma}} \text{rep abs})p_r$$
Transformations

- A transformation is:

\[
\vdash e_s \xrightarrow{\tau} e_t : \Gamma
\]

Transformation requires the following proof:

\[ e_s \equiv \text{D} \tau \text{rep abs}( \text{D} \Gamma \text{rep abs}) e_t \]
Transformations

- A transformation is:
  - a derivation of the judgment $\Gamma \vdash e_s \rightsquigarrow e_t : \tau$ and
Transformations

A transformation is:

- a derivation of the judgment $\hat{\Gamma} \vdash e_s \rightsquigarrow_R e_t : \hat{\tau}$ and
- a relation between source $e_s$ and program $e_t$ with a common type functor $\hat{\tau}$ with a common context $\hat{\Gamma}$.
Transformations

- A transformation is:
  - a derivation of the judgment \( \Gamma \vdash e_s \sim^R e_t : \tau \) and
  - a relation between source \( e_s \) and program \( e_t \) with a common type functor \( \hat{\tau} \) with a common context \( \hat{\Gamma} \).

- Transformation requires the following proof:

\[
e_s \equiv \mathcal{D}_{\hat{\tau}} \text{rep abs} \left( \mathcal{D}_{\hat{\Gamma}} \text{rep abs} \right) e_t
\]
Transformations

- A transformation is:
  - a derivation of the judgment $\Gamma \vdash e_s \overset{R}{\Rightarrow} e_t : \hat{\tau}$ and
  - a relation between source $e_s$ and program $e_t$ with a common type functor $\hat{\tau}$ with a common context $\hat{\Gamma}$.

- Transformation requires the following proof:
  \[ e_s \equiv D_{\hat{\tau}} \text{rep abs} (D_{\hat{\Gamma}} \text{rep abs}) e_t \]

- But first...
Visualizing the Types

Transformation: $\Gamma \vdash e_s \overset{R}{\Rightarrow} e_t : \tau$

Rewriting: $\Gamma \triangleright p_l : \tau_l \leadsto p_r : \tau_r$
Transformations (2)

- Transformation is specified by inference rules.
Transformations (2)

- Transformation is specified by inference rules.
- Most rules are very similar to the HM type inference rules:

\[
\Gamma \vdash f : \tau \rightarrow \nu \\
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash fe : \nu \\
\]

(App)

\[
\check{\Gamma} \vdash f_s R \Rightarrow f_t : \check{\tau} \rightarrow \check{\nu} \\
\check{\Gamma} \vdash e_s R \Rightarrow e_t : \check{\tau} \\
\hline
\check{\Gamma} \vdash f_s e_s R \Rightarrow f_t e_t : \check{\nu} \\
\]

(T-App)
Transformations (2)

- Transformation is specified by inference rules.
- Most rules are very similar to the HM type inference rules:

\[
\begin{align*}
\Gamma &\vdash f : \tau \to \nu \\
\Gamma &\vdash e : \tau \\
\hline
\Gamma &\vdash f\ e : \nu
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash f_s \rightsquigarrow f_t : \hat{\tau} \to \hat{\nu} \\
\Gamma &\vdash e_s \rightsquigarrow e_t : \hat{\tau} \\
\hline
\Gamma &\vdash f_s\ e_s \rightsquigarrow f_t\ e_t : \hat{\nu}
\end{align*}
\]

- With one addition:

\[
\begin{align*}
(\hat{\Gamma}_m \triangleright p_l : \hat{\tau}_l \rightsquigarrow p_r : \hat{\tau}_r) &\in R \\
\hat{\Gamma} &\vdash e_s \rightsquigarrow e_t : \hat{\tau}_l \\
\hat{\Gamma};\hat{\Gamma}_m &\vdash e_s \rightsquigarrow p_l\ l e_t \Rightarrow \theta \\
\hat{\Gamma} &\vdash e_s \rightsquigarrow \theta p_r : \hat{\tau}_r
\end{align*}
\]
Complete Transformations

A complete transformation is a transformation where the $\bar{\tau}$ and $\bar{\Gamma}$ are free of $\bar{\iota}$:

- $\bar{\iota}(\alpha/B) = true$
- $\bar{\iota}(\iota) = false$
- $\bar{\iota}(\bar{\tau} \rightarrow \bar{\upsilon}) = \bar{\iota}(\bar{\tau}) \land \bar{\iota}(\bar{\upsilon})$
Complete Transformations

- A *complete transformation* is a transformation where the $\tau$ and $\Gamma$ are free of $\iota$:
  \[
  \bar{i}(\alpha/B) = \text{true} \\
  \bar{i}(\iota) = \text{false} \\
  \bar{i}(\tau \rightarrow \nu) = \bar{i}(\tau) \land \bar{i}(\nu)
  \]

- In other words, the types of the source and target are the same under the same environment.
Discussion

- Proofs:

  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

Algorithm:

  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.

$\star$ A source can have many targets.

$\star$ We describe heuristics to choose the "best" target.

More in a paper (PEPM 2014):

- Parameterized type constructors and difference lists
- Other example applications of TTSs
- Related work

Johan Jeuring  Sean Leather, Andres Löh, Bram Schuur (Utrecht University)
Discussion

- Proofs:
  - The proof for transformation is quite involved but only needs to be done once (per language).

- Algorithm:
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.

- A source can have many targets.
- We describe heuristics to choose the "best" target.

More in a paper (PEPM 2014):
  - Parameterized type constructors and difference lists
  - Other example applications of TTSs

Related work
Discussion

**Proofs:**

- The proof for transformation is quite involved but only needs to be done once (per language).
- The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

**Algorithm:**

- We adapted algorithm $W$ to automatically transform programs.
- It is sound but not complete.

⋆ A source can have many targets.
⋆ We describe heuristics to choose the “best” target.

More in a paper (PEPM 2014):

- Parameterized type constructors and difference lists
- Other example applications of TTSs
- Related work
Discussion

- Proofs:
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- Algorithm:

- A source can have many targets.
- We describe heuristics to choose the "best" target.

More in a paper (PEPM 2014):

- Parameterized type constructors and difference lists
- Other example applications of TTSs
- Related work
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm $W$ to automatically transform programs.
Discussion

- Proofs:
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- Algorithm:
  - We adapted algorithm $\mathcal{W}$ to automatically transform programs.
  - It is sound but not complete.
Discussion

Proofs:
- The proof for transformation is quite involved but only needs to be done once (per language).
- The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

Algorithm:
- We adapted algorithm $W$ to automatically transform programs.
- It is sound but not complete.
  - A source can have many targets.
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.
Discussion

- Proofs:
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- Algorithm:
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.

- More in a paper (PEPM 2014):
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.

- **More in a paper (PEPM 2014):**
  - Parameterized type constructors and difference lists
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm \( W \) to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.

- **More in a paper (PEPM 2014):**
  - Parameterized type constructors and difference lists
  - Other example applications of TTSs
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.

- **More in a paper (PEPM 2014):**
  - Parameterized type constructors and difference lists
  - Other example applications of TTSs
  - Related work
Discussion

- **Proofs:**
  - The proof for transformation is quite involved but only needs to be done once (per language).
  - The difficulty of the proofs for each rewrite rule is probably proportional to the complexity of the isomorphism. The examples we have tried in this language were relatively easy.

- **Algorithm:**
  - We adapted algorithm $W$ to automatically transform programs.
  - It is sound but not complete.
    - A source can have many targets.
    - We describe heuristics to choose the “best” target.

- **More in a paper (PEPM 2014):**
  - Parameterized type constructors and difference lists
  - Other example applications of TTSs
  - Related work

*FIN*