Transforming Types

Johan Jeuring

Joint work with Rui Guerra

Introduction

- Information possesses structure (has a type), and structural information is used to store, edit, view, and search in data.
- There are many applications in which you want to view (values of) certain types as other types, or transform types to other types:
  - when two types are isomorphic, you want to use functionality on one type also on the other type;
  - to suggest program corrections in type checking;
  - cut & paste;
  - coercive subtyping;
  - schema/data type evolution;
  - …

Isomorphic types

Suppose you want to use of two different libraries with functionality on dates. The first one defines Date by

```
data Date  = Date Day Month Year
data Day   = Day   Int
data Month = Month Int
data Year  = Year  Int
```

the second by:

```
data Date' = Date' (Int, Int, Int)
```

How can I mix functions from the two libraries in a single program?

Suggesting program corrections I

The following example is inspired by 'How to Repair Type Errors Automatically' from Bruce McAdam (Trends in functional programming, 2002). Consider the following program

$$
\begin{aligned}
&square :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&square\ i = i \star i \\
&squareList :: \mathsf{Int} \rightarrow [\mathsf{Int}] \\
&squareList\ n = map\ ([1\,..\,n], square)
\end{aligned}
$$

This program is incorrect, the programmer probably meant:

$$
\begin{aligned}
&square :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&square\ i = i \star i \\
&squareList :: \mathsf{Int} \rightarrow [\mathsf{Int}] \\
&squareList\ n = map\ square\ [1\,..\,n]
\end{aligned}
$$

but didn't know how to use *map* properly.

Suggesting program corrections II

The type of *map* in the prelude is

$(a \to b) \to [a] \to [b]$

map's expected type is

$([a], a \to b) \to [b]$

These types are isomorphic under (un)currying and product commutativity.

Cut & Paste

- In an editor you want to cut and paste data from one place to another. But what if the types don't match?

Cut & Paste

- In an editor you want to cut and paste data from one place to another. But what if the types don't match?
- Transform!

Cut & Paste

- In an editor you want to cut and paste data from one place to another. But what if the types don't match?
- Transform!
- If, for example, I paste

  $Date'\ (20, 02, 2005)$

  to a location that expects values of type Date, I want it to be transformed silently to

  $Date\ (Day\ 20)\ (Month\ 02)\ (Year\ 2005)$

Cut & Paste

- In an editor you want to cut and paste data from one place to another. But what if the types don't match?
- Transform!
- If, for example, I paste

  $\mathit{Date}'\ (20, 02, 2005)$

  to a location that expects values of type Date, I want it to be transformed silently to

  $\mathit{Date}\ (\mathit{Day}\ 20)\ (\mathit{Month}\ 02)\ (\mathit{Year}\ 2005)$

- This problem has been studied in the structure editors community. For example: Akpotsui, Quint, Roisin. Type Modelling for Document Transformation in Structured Editing Systems.

Coercive subtyping

- Kiessling and Luo (Coercions in Hindley-Milner systems, Types 2004): 'Coercive subtyping is a framework of abbreviation for dependent type theories.'
- If you want to silently coerce an integer to a float, you can write the following code in Kiessling and Luo's system:

$int2float :: \mathsf{Int} \rightarrow \mathsf{Float}$
$int2float = \ldots$

cdec $int2float :: \mathsf{Int} \rightarrow \mathsf{Float}$

Schema evolution

The database community has been working (a lot) on Schema transformation, integration, and translation.

Approaches to type transformations

In all these examples we want to have a function that transforms values of one type to another type, with as little effort as possible.

- Obviously, generic transformations between isomorphic types are of no help for non-isomorphic types.

Approaches to type transformations

In all these examples we want to have a function that transforms values of one type to another type, with as little effort as possible.

- Obviously, generic transformations between isomorphic types are of no help for non-isomorphic types.
- The suggestions for type corrections do not generate transformations.

Approaches to type transformations

In all these examples we want to have a function that transforms values of one type to another type, with as little effort as possible.

- Obviously, generic transformations between isomorphic types are of no help for non-isomorphic types.
- The suggestions for type corrections do not generate transformations.
- The type transformations in structure editors are built-in, and only described informally.

Approaches to type transformations

In all these examples we want to have a function that transforms values of one type to another type, with as little effort as possible.

- Obviously, generic transformations between isomorphic types are of no help for non-isomorphic types.
- The suggestions for type corrections do not generate transformations.
- The type transformations in structure editors are built-in, and only described informally.
- The Hindley-Milner system extended with coercions only allows a single coercion between two types.

This talk

A ‘type system’ and a ‘transformation inference algorithm’:

- ➜ Type transformation rules.
- ➜ An algorithm for calculating the minimum cost type transformation.
- ➜ Soundness and completeness claims.

Given two types, the minimum cost type transformation between these types is *inferred*.

It is a different problem to *refactor* a given type to a different type

Type transformations

Definition 1 (Type Transformation) *A* type transformation *between types* a *and* b *is a* t *such that* $\mathsf{a} \quad \mapsto_t \quad \mathsf{b}$ *is derivable using the following rules.*

Basic type transformation rules

$$\frac{}{\mathsf{a} \;\mapsto_{id}\; \mathsf{a}}$$

$$\frac{\mathsf{a} \;\mapsto_{m}\; \mathsf{b} \qquad \mathsf{b} \;\mapsto_{n}\; \mathsf{c}}{\mathsf{a} \;\mapsto_{trans\ (m,n)}\; \mathsf{c}}$$

Placeholder transformation rules: example

If two types don't match, I still want to be able to transform values from one to the other.

$\mathsf{Int} \quad \mapsto_{string} \quad \mathsf{String}$

This should be expensive.

Alternatively, it should be possible to add special-purpose coercions, together with their cost, to the type transformation system.

Placeholder transformation rules

$$\frac{}{\mathsf{a} \quad \mapsto_{unit} \quad \mathsf{Unit}}$$

$$\frac{}{\mathsf{a} \quad \mapsto_{string} \quad \mathsf{String}}$$

$$\frac{}{\mathsf{a} \quad \mapsto_{int} \quad \mathsf{Int}}$$

Product transformation rules

$$\frac{\mathsf{a} \qquad \mathsf{b}}{\mapsto_{prodIntro} \quad \mathsf{a} \times \mathsf{b}}$$

$$\frac{}{\mathsf{a} \times \mathsf{b} \ \mapsto_{fst} \ \mathsf{a}} \qquad \frac{}{\mathsf{a} \times \mathsf{b} \ \mapsto_{snd} \ \mathsf{a}}$$

$$\frac{}{\mathsf{a} \times \mathsf{b} \ \mapsto_{swapprod} \ \mathsf{b} \times \mathsf{a}}$$

$$\frac{\mathsf{a} \ \mapsto_{m} \ \mathsf{a}' \qquad \mathsf{b} \ \mapsto_{n} \ \mathsf{b}'}{\mathsf{a} \times \mathsf{b} \ \mapsto_{prod\ (m,n)} \ \mathsf{a}' \times \mathsf{b}'}$$

Sum transformation rules

$$\frac{}{\mathsf{a} \;\mapsto_{sumInl}\; \mathsf{a} + \mathsf{b}} \qquad \frac{}{\mathsf{b} \;\mapsto_{sumInr}\; \mathsf{a} + \mathsf{b}}$$

$$\frac{\mathsf{a} \;\mapsto_{m}\; \mathsf{c} \qquad \mathsf{b} \;\mapsto_{n}\; \mathsf{c}}{\mathsf{a} + \mathsf{b} \;\mapsto_{either\ (m,n)}\; \mathsf{c}}$$

$$\frac{}{\mathsf{a} + \mathsf{b} \;\mapsto_{swapsum}\; \mathsf{b} + \mathsf{a}}$$

$$\frac{\mathsf{a} \;\mapsto_{m}\; \mathsf{a}' \qquad \mathsf{b} \;\mapsto_{n}\; \mathsf{b}'}{\mathsf{a} + \mathsf{b} \;\mapsto_{sum\ (m,n)}\; \mathsf{a}' + \mathsf{b}'}$$

Function transformation rules

$$\frac{}{\mathsf{a} \times \mathsf{b} \rightarrow \mathsf{c} \ \mapsto_{curry} \ \mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{c}}$$

$$\frac{}{\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{c} \ \mapsto_{uncurry} \ \mathsf{a} \times \mathsf{b} \rightarrow \mathsf{c}}$$

$$\frac{}{\mathsf{a} \ \mapsto_{const} \ \mathsf{Unit} \rightarrow \mathsf{a}} \qquad \frac{}{\mathsf{Unit} \rightarrow \mathsf{a} \ \mapsto_{unconst} \ \mathsf{a}}$$

$$\frac{\mathsf{a} \ \mapsto_{m} \ \mathsf{a}' \qquad \mathsf{b} \ \mapsto_{n} \ \mathsf{b}'}{\mathsf{a}' \rightarrow \mathsf{b} \ \mapsto_{fun\ (m,n)} \ \mathsf{a} \rightarrow \mathsf{b}'}$$

Constructor transformation rules

$$\frac{}{\mathsf{Con}\ c\ \mathsf{a}\ \ \mapsto_{rmConstr}\ \ \mathsf{a}}$$

$$\frac{}{\mathsf{a}\ \ \mapsto_{addConstr}\ \ \mathsf{Con}\ c\ \mathsf{a}}$$

$$\frac{\mathsf{a}\ \ \mapsto_{m}\ \ \mathsf{a}'}{\mathsf{Con}\ c\ \mathsf{a}\ \ \mapsto_{con\ (m)}\ \ \mathsf{Con}\ c\ \mathsf{a}'}$$

About the rules

Are these rules the minimal set of type rules?

- The sum and product rules are the standard monoidal iso's.

About the rules

Are these rules the minimal set of type rules?

- The sum and product rules are the standard monoidal iso's.
- The function rules correspond to the laws of the exponentials. The rule

  $$\overline{(\mathsf{a} + \mathsf{b}) \rightarrow \mathsf{c} \ \mapsto_{sumprod} \ (\mathsf{a} \rightarrow \mathsf{c}) \times (\mathsf{b} \rightarrow \mathsf{c})}$$

  and its converse are derivable, and therefore omitted. However, we might want to add them because we want these transformations to be 'cheap'.

About the rules

Are these rules the minimal set of type rules?

- The sum and product rules are the standard monoidal iso's.
- The function rules correspond to the laws of the exponentials. The rule

$$\overline{(\mathsf{a} + \mathsf{b}) \rightarrow \mathsf{c} \ \mapsto_{sumprod} \ (\mathsf{a} \rightarrow \mathsf{c}) \times (\mathsf{b} \rightarrow \mathsf{c})}$$

  and its converse are derivable, and therefore omitted. However, we might want to add them because we want these transformations to be 'cheap'.
- I suspect I want to add rules about subtyping.

Minimum cost type transformations

Suppose there exists an ordering on transformations.

Definition 2 (Minimum cost type transformation) *A* minimum cost type transformation *between types* a *and* b *is a type transformation t between* a *and* b *such that for any other type transformation t' between* a *and* b, $t \leqslant t'$.*

Theorem 1 *Given any two types* a *and* b*, there exists a minimum cost type transformation.*

In general this minimum cost type transformation will not be unique. The ordering on transformations should be such that:

Theorem 2 *Given two canonically isomorphic types* a *and* b*, the minimum cost type transformation between* a *and* b *corresponds (in some sense) to the isomorphism between* a *and* b.

Inferring minimum cost type transformations

I'd like to have a function that automatically infers a (or the) minimum cost type transformation TYPETRANSFORM between two types.

Frank Atanassow and I have shown how to generate the unique isomorphism between two isomorphic types.

We want to use similar techniques to infer a minimum cost type transformation.

[We haven't looked at the situation in which multiple solutions exist yet.]

TYPETRANSFORM **is a generic function**

- TYPETRANSFORM takes two types as arguments, and returns a function, the structure of which depends on the structure of the arguments types.

TYPETRANSFORM **is a generic function**

- TYPETRANSFORM takes two types as arguments, and returns a function, the structure of which depends on the structure of the arguments types.
- TYPETRANSFORM is a generic function that depends on *two* type arguments.

TYPETRANSFORM **is a generic function**

- TYPETRANSFORM takes two types as arguments, and returns a function, the structure of which depends on the structure of the arguments types.
- TYPETRANSFORM is a generic function that depends on *two* type arguments.
- Generic functions in Generic Haskell take a single type as argument.

TYPETRANSFORM **is a generic function**

- TYPETRANSFORM takes two types as arguments, and returns a function, the structure of which depends on the structure of the arguments types.
- TYPETRANSFORM is a generic function that depends on *two* type arguments.
- Generic functions in Generic Haskell take a single type as argument.
- We can get around this restriction by
  - producing a representation of the source value in a universal language (a generic function depending on the type Source),
  - and calculating the minimum cost type transformation from that representation to the target type (a generic function depending on the type Target).

High level structure

$$
\begin{aligned}
&\mathit{typetransform} :: \mathsf{Source} \to \mathsf{Target} \\
&\mathit{typetransform} = \mathit{mctt}\langle \mathsf{Target} \rangle \,.\, \mathit{reduce}\langle \mathsf{Source} \rangle \\
&\mathit{reduce}\langle \mathsf{t} :: \star \rangle :: \mathsf{t} \to \mathsf{Univ} \\
&\mathit{mctt}\langle \mathsf{t} :: \star \rangle \quad :: \mathsf{Univ} \to \mathsf{t}
\end{aligned}
$$

Reducing to a universal value

Function *reduce*⟨t⟩ reduces a value of type t to a value of a universal data type, defined by, for example

```
data Univ = UUnit Unit
          | UInt  Int
          | UStr  String
          | USum  Opt       Univ
          | UProd Univ      Univ
          | UCon  ConDescr Univ
data Opt  = ULeft | URight
reduce⟨t :: ⋆⟩ :: t → Univ
```

Costs

We define a data type Cost:

```
data Cost = IdCost
          | TransCost Cost Cost
          | UnitCost
          | IntCost
          | StringCost
          | ...
minCost :: [Cost] → Cost
```

Furthermore, we have two obvious mappings, *cost2tt* and *tt2cost*, from Cost to type transformations and vice versa.

The minimum cost type transformation

Function $mctt\langle \mathsf{t} \rangle$ returns the minimum cost type transformation. It is a kind of parsing function with type:

$$mctt\langle \mathsf{t} :: \star \rangle :: [\mathsf{Univ}] \rightarrow (\mathsf{t}, \mathsf{Cost}, [\mathsf{Univ}])$$

It implements the type rules given at the beginning of this talk. It is a large function, with arms of the form:

```
mctt⟨Int⟩ univ@((UInt int) : rest) =
   let id    = (int, IdCost, rest)
       phint = (0,   IntCost, univ)
   in minCost2nd [id, phint]
```

Soundness and optimality

We want to prove the following theorem:

Theorem 3 (TYPETRANSFORM is sound and optimal) *If*

$typetransform\ source = (target, cost, [\,])$

then cost2tt cost is a minimum cost type transformation.

Completeness

We would like to have the following result:

Theorem 4 (TYPETRANSFORM is complete) *If t is a minimum cost type transformation, then*

$$\mathit{typetransform}\ \mathit{source} = (\mathit{target}, \mathit{cost}, [\,])$$

where $\mathit{tt2cost}\ t = \mathit{cost}$.

However, since I expect that the minimum cost type transformation is not unique in general, this is unlikely to hold.

Conclusions and future work

- Finish the implementation, and develop some heuristics to increase efficiency.
- Work out some more realistic examples.
- (Dis)prove the theorems.
- …