Heuristic-Driven Theory Projection in Haskell

M. Petrowitch, X. Ripoll, R. Tosswill

Friday 4th February, 2022

Abstract

In [SKGK14], the authors provide an overview of Heuristic-Driven Theory Projection (HDTP), a logic-based computational model of analogical reasoning. In this framework, an agent's knowledge of a familiar domain S is represented as a first-order theory, which can be projected into another, less familiar domain T, by constructing a more general domain G, using anti-unification of formulas [Plo70].

To our knowledge, there has been so far no implementation of HDTP in a functional programming language. In the following report, we provide a functional algorithm in Haskell for reproducing many of the aspects of HDTP laid out in [SKGK14].

Contents

1 Analogical Reasoning and HDTP **2**

2 First-Order Theories and Basic Types **3**

3 Anti-Unification and Least General Generalization **4**

4 Basic Substitutions and Not Too General Generalizations **7**

4.1 Basic Substitutions 7

4.2 LGGs and the Basic Substitutions 9

4.3 Generating NTGGs 10

4.4 Alignments, Complexity of NTGGs, and Domain Generalizations 10

5 Analogical Transfer **13**

6 Testing the Algorithm **14**

7 Conclusion and Future Work **14**

Bibliography **15**

1 Analogical Reasoning and HDTP

Analogical reasoning is a central aspect of human problem-solving skills [GHK01], creativity [BP15], and concept formation [HS13], and as such, is a major focus of research in cognitive science. Traditionally, analogical reasoning is defined to be the process in which an agent will attempt to understand an unfamiliar domain of knowledge T (the *target* domain) by noticing its similarity to a more familiar domain S (the *source* domain). This similarity allows an agent to propose novel hypotheses about the target domain, supposing that they have discovered a useful similarity between source and target.

The HDTP framework as laid out in [SKGK14], approaches analogical reasoning from a syntactic point of view, representing domains of knowledge as first-order theories, and analogies as formal correspondences between formulae of those theories.

HDTP sees the process of analogical reasoning as consisting of three phases:

- *Retrieval*, in which the source domain is formalized as a first-order theory;
- *Mapping*, in which a formal correspondence is established between formulae in the source and target domains, by way of *anti-unification*;
- *Transfer*, in which the correspondence established in the mapping phase is used to generate new formulae in the target domain, to serve as hypotheses in further reasoning.

These three phases will guide our discussion of the current Haskell implementation over the course of the next three sections: Section 2 will provide details about how we have implemented first order theories, which will also serve to cover the retrieval phase of HDTP. Sections 3 and 4 will focus on the implementation of restricted higher-order anti-unification, which is the formal process at the heart of the mapping phase of HDTP. Section 5 will lay out how we have implemented a procedure for analogical transfer. Section 6 is an overview of the implementation of tests. In Section 7, we will summarize the achievements of the project so far, as well as what remains to be completed in further work.

2 First-Order Theories and Basic Types

Unlike other models of analogical reasoning (see e.g. [HM95], [Gen83]), but in line with broader trends in the cognitive science literature, HDTP represents an agent's knowledge of any given domain of knowledge as a finite, multi-sorted, first-order theory. In order to implement this in Haskell, we first need to define things from the ground up.

Classically, a first-order language $\mathcal{L}$ is a set of formulae built out of predicates, logical symbols, and terms, which are themselves build out of non-logical symbols drawn from a signature. The particular formulation written below draws heavily from the formalization presented in [SKKG09]:

Definition 2.1 (Signature)**.** We define a signature to be a 5-tuple $\Sigma = (Sort_\Sigma, Func_\Sigma, Pred_\Sigma, arP, arF)$, where

- $Sort_\Sigma$ is a set of *sorts*;
- $Func_\Sigma$ is a set of *function symbols*;
- $Pred_\Sigma$ is a set of *predicate symbols*;
- $arP : Pred_\Sigma \to Sort_\Sigma^*$ is an *arity function* on predicate symbols, giving each predicate symbol its arity;
- $arF : Func_\Sigma \to Sort_\Sigma^* \times Sort_\Sigma$ is an *arity function* on functions, sending each function symbol to its domain and codomain.

In Haskell, we first define some types for function, predicate and variable symbols as well as sorts.

```
module HDTP where
import Data.List ((\\), find, sortBy, minimumBy)
```

```
type FunSymb = String
newtype PredSymb = PS String deriving (Eq, Show)
type VarSymb = String
newtype Sort = S String deriving (Eq, Show)
```

Here `FunSymb` and `VarSymb` are both of type string. This is not the safest way to implement those types but we make sure that we only use them within other type defitions that avoid any ambiguities.

Definition 2.2 (Terms)**.** Let $V = \{v_1 : s, v_2 : s \dots\}$ be an infinite set of sorted variables, and Σ a signature. We can define a term algebra $Term(\Sigma, V)$ recursively as the smallest set obeying the following conditions:

(1) If $x : s \in V$, then $x \in Term(\Sigma, V)$.

(2) If $f : s_1 \times \dots s_n \to s \in Func_\Sigma$, and $t_1 : s_1, \dots t_n : s_n \in Term(\Sigma, V)$, then $f(t_1, \dots t_n) : s \in Term(\Sigma, V)$.

Definition 2.3 (Formula)**.** Given signature Σ and term algebra $Term(\Sigma, V)$, we can define the set of formulae over Σ and V, $Form(\Sigma, V)$ recursively as the smallest set obeying the following conditions:

1. If $p : s_1 \times s_n \in Pred_\Sigma$, and $t_1 : s_1, \ldots t_n : s_n \in Term(\Sigma, V)$, then $p(t_1, \ldots, t_n) : s \in Form(\Sigma, V)$.
2. If $\alpha, \beta \in Form(\Sigma, V)$, then $\neg\alpha, \alpha \vee \beta, \forall x_i : s_i\ (\alpha) \in Form(\Sigma, V)$.

As they are defined, the function and variable symbols have a specific arity, consisting of lists of sorts that they take as arguments, together with the sort they return (just the latter in the case of constants). Unfortunately, it is impossible to use the Haskell type system to represent such sorts as types if we want to leave the sorts open to be defined by the final user of our implementation. This means that we need a layer on top of each formula that allows us to check the arity of every object, including predicates and functions.

Although we considered various options for this, none of them are failproof. One of our initial designs consisted of carrying the signature of each function along with its symbol, but it turned out to be cumbersome and limited in providing a universal way to check the sorts. We ended up opting for global "database" functions (`symbAr` for function and variable symbols, and `predAr` for predicates) that let us define a global store. Global variables are usually considered bad practice in a functional context, but they are a nice compromise between ease of use and technical complexity when used appropriately.

```
-- newtype Term Sg [[s], [f], [p], af, ap] [v] = v | f [Term] deriving (Eq,Ord,Show)
data TSymb = VS VarSymb | FS FunSymb deriving (Eq, Show)
data Term = T TSymb [Term] deriving (Eq, Show) -- Find a way to restrict this to respect
    sorts
data Form = FT PredSymb [Term] | Not Form | Disj Form Form | Forall VarSymb Form deriving
    (Eq, Show)

symbAr :: TSymb -> ([Sort], Sort)
symbAr (VS "F") = ([S "a"], S "b")
symbAr (VS "F'") = ([S "a"], S "b")
symbAr (VS "G") = ([S "a"], S "b")
symbAr (VS "W") = ([S "a"], S "b")
symbAr _ = undefined
predAr :: PredSymb -> [Sort]
predAr = undefined
```

Finally, with all of these preliminaries in place, we can define the basic unit of knowledge in HDTP, the domain:

Definition 2.4 (Domain)**.** We define a *domain* to be a finite set of formulae $\mathcal{D}_\alpha = \{\alpha_1, \ldots \alpha_n\}$ (where $\alpha_i \in \mathcal{L}_\Sigma$) closed under logical consequence.

```
type Domain = [Form]
```

3 Anti-Unification and Least General Generalization

The proper identification of common structure between seemingly disparate domains of knowledge is an essential ingredient of analogical reasoning, and as HDTP represents domains as sets of

first-order formulae, what is needed is a well-defined notion of shared structure in formulae.

Originally proposed in [Plo70], generalization (also known as anti-unification) is just such a notion: an anti-unifier of any two terms s, t is a term containing only their shared syntactic structure: the distinguishing details of s and t have been abstracted away by replacing the constants with variables. More formally:

Definition 3.1 (First-order Substitution on Variables)**.** Given a term algebra $Term(\Sigma, V)$, a *first-order substitution* is a partial function $\sigma : V \to Term(\Sigma, V)$ mapping variables to terms, formally represented by $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ (provided sorts match). An application of a substitution σ to a term is defined by induction on the structure of a term as below:

- $$apply(x, \sigma) = \begin{cases} t & \text{if } x \mapsto t \in \sigma \\ x & \text{otherwise} \end{cases}$$
- $apply(f(t_1, \ldots, t_n), \sigma) = f(apply(t_1, \sigma), \ldots, apply(t_n, \sigma))$

Definition 3.2 (First-order Generalization)**.** Let s, t be first-order terms (resp. formulae). A *generalization* is a triple $g = \langle a, \sigma, \tau \rangle$ with term (resp. formula) a and substitutions σ, τ such that $s \xleftarrow{\sigma} a \xrightarrow{\tau} t$. We say that a is an *anti-unifier* of $\{s, t\}$.

By themselves, generalizations aren't necessarily helpful. A generalization that removes too much detail leaves us with a term devoid of any real content.

For example, let $s := 2 + 6$ and $t := 3 + 6$. If we consider the anti-unifer $a := x + y$ in generalization $g = \langle x + y, (x \mapsto 2, y \mapsto 6), (x \mapsto 3, y \mapsto 6) \rangle$, we can see that while g is most certainly a generalization of s, t, it is one which has abstracted away too much information, as it has failed to notice that both s and t are terms in which 6 is being added to another number. Instead, the generalization g is only capable of showing that *some* numbers are being added together.

What's needed to get around this is some notion of minimality:

Definition 3.3 (Least General Generalization)**.** For terms s, t, we call a generalization $g = \langle a, \sigma, \tau \rangle$ the *least general generalization (lgg)*, if for every generalization $g' = \langle a', \sigma', \tau' \rangle$, there exists a (unique) substitution $\gamma : a' \to a$ such that $\sigma' = \sigma \circ \gamma$ and $\tau' = \tau \circ \gamma$. We say that a is the *most specific anti-unifier* of $\{s, t\}$.

In this way, least general generalizations constitute limits in the category of first-order substitutions. As proven in [Plo70], this category has all finite limits: for any finite set of first-order terms, one can always find a lgg.

In the original 1970 paper, this result was shown constructively with an imperative algorithm. Helpfully for us, a functional algorithm to do the same is detailed in [TTF13, 3]. It runs as follows.

$$\lambda(t, t, \theta) = (t, \theta))$$
$$\lambda(f(t_1, \ldots, t_n), f(u_1, \ldots, u_n), \theta_0) = (f(x_1, \ldots, x_n), \theta_n)) \quad \text{where } \lambda(t_i, u_i, \theta_{i-1}) = (x_i, \theta_i)$$
$$\lambda(t, u, \theta) = (x, \theta)) \quad \text{if } \theta(x) = (t, u)$$
$$\lambda(t, u, \theta) = (y, \theta') \quad \text{where } y \notin \mathrm{dom}(\theta) \text{ and } \theta' = \theta + \{y \mapsto (t, u)\}$$

$$\texttt{leastGen}(t, u) = \pi_1(\lambda(t, u, \{\}))$$

To implement this algorithm in Haskell, we first build a helper function which takes a list of variable symbols and gives us a new variable name that is not yet in that list. Exploiting Haskell's lazyness this function can create new variable names for a given list of any length. We first add X, Y, Z, W to the infinite list of possible variable symbols as those are our preferred variable names.

```
newVariable :: [VarSymb] -> VarSymb
newVariable xs = head (allVarSymb \\ xs) where
   allVarSymb = ["X","Y","Z","W"] ++ [ c : s | s <- "": allVarSymb, c <- ['A'..'Z']]
```

The function `lambdaForTerms` is an implementation of the lgg algorithm for terms as stated above and in [TTF13, 3].

```
lambdaForTerms :: Term -> Term -> [TermGen] -> (Term, [TermGen])
lambdaForTerms t u theta | t == u = (t, theta)
lambdaForTerms (T (FS f) ts) (T (FS f') us) theta | f == f' = (T (FS f) (map fst
    termSubsList), snd (last termSubsList)) where
  termSubsList = sameTop ts us theta
lambdaForTerms t u theta = case find (\(_, t', u') -> t == t' && u == u') theta of
  Just (x, _, _) -> (T (VS x) [], theta)
  Nothing -> (T (VS x) [], (x, t, u):theta) where x = newVariable (map (\(vs, _, _) -> vs)
      theta)
```

Here we use the helper function `sameTop` that takes two lists of terms $[t_1, \ldots, t_n]$, $[u_1, \ldots, u_n]$ and a list of generalisations θ_0 and computes recursively the i-th generalisation taking the i-th terms from the lists and the (i-1)-th generalisation. The result is of the form $[(x_1, \theta_1), \ldots (x_n, \theta_n)]$.

```
sameTop :: [Term] -> [Term] -> [TermGen] -> [(Term, [TermGen])]
sameTop [] [] _ = []
sameTop (u:us) (t:ts) theta = lambdaOfTerms : sameTop us ts (snd lambdaOfTerms) where
                                  lambdaOfTerms = lambdaForTerms u t theta
sameTop _ _ _ = undefined -- sameTop is only applied to two equal predicates, hence same
    number of arguments
```

The original algorithm from [TTF13] is designed for generalizing terms, not formulas. We have adapted it to pairs of general formulas, provided that they have the same predicate structure.

```
lambda :: Form -> Form -> [TermGen] -> (Form, [TermGen])
lambda phi psi theta | phi == psi = (phi, theta)
lambda (FT ps ts) (FT ps' us) theta | ps == ps' = (FT ps (map fst prSubsList), snd (last
    prSubsList)) where
                                                    prSubsList = sameTop ts us theta
lambda (Not phi) (Not psi) theta = (Not outForm, subs) where (outForm, subs) = lambda phi
    psi theta
lambda (Disj phi phi') (Disj psi psi') theta = (Disj outForm outForm', subs ++ subs')
    where
```

```
  (outForm, subs) = lambda phi psi theta
  (outForm', subs') = lambda phi' psi' theta
lambda (Forall vs phi) (Forall _ psi) theta = (Forall vs outForm, subs) where (outForm,
    subs) = lambda phi psi theta
lambda _ _ _ = undefined -- We only anti-unify formulas that have the same predicate
    structure
```

4 Basic Substitutions and Not Too General Generalizations

While first-order anti-unification is useful for many purposes, it isn't strong enough to capture similarities which humans regularly can when forming analogies. For example, consider terms $s := 2 + 3$ and $t := 2 \times 3$. Using only first-order substitutions, the lgg of $\{s, t\}$ is simply $g = \langle x, x \mapsto 2 + 3, x \mapsto 2 \times 3 \rangle$. Because first-order substitutions cannot instantiate function symbols, the obvious generalization is not found.

4.1 Basic Substitutions

To better capture these kinds of situations, HDTP replaces the simple form of first-order substitutions with four so-called "basic substitutions", functions $bs : Term(\Sigma, V) \to Term(\Sigma, V)$ which allow for some second-order properties.

Following what appears to be the authors' intent, we can generalize these basic substitutions to apply to formulae. Our implementation of these substitutions models the information that they carry with themselves, rather than their action as functions. We then provide a function `apply` that actually executes the action they represent. Having the subsitutions as data structures instead of Haskell functions allows for destructuring and comparing them.

```
data Sub = SR Renaming | SF Fixation | SI Insertion | SP Permutation deriving (Eq, Show)

apply :: Sub -> Form -> Form
apply (SR r) (FT predSymb ts) = applyRenaming r (FT predSymb ts)
apply (SF f) (FT predSymb ts) = applyFixation f (FT predSymb ts)
apply (SI i) (FT predSymb ts) = applyInsertion i (FT predSymb ts)
apply (SP p) (FT predSymb ts) = applyPermutation p (FT predSymb ts)
apply r (Not form) = Not (apply r form)
apply r (Disj form form') = Disj (apply r form) (apply r form')
apply r (Forall w form) = Forall w (apply r form)

apply' :: [Sub] -> Form -> Form
apply' subs phi = foldr apply phi subs
```

In Haskell, we implement renamings, fixations, argument insertions and permutations as follows.

Definition 4.1 (Renaming)**.** A *renaming* $\rho^F_{F'}$ replaces a variable $F : s_1 \times \cdots \times s_n \to s \in V_n$ with a variable $F' : s_1 \times \cdots \times s_n \to s \in V_n$

$$F(t_1 : s_1, \ldots t_n : s_n) : s \xrightarrow{\rho^F_{F'}} F'(t_1 : s_1, \ldots t_n : s_n) : s$$

```
newtype Renaming = R (VarSymb, VarSymb) deriving (Eq, Show)
```

```haskell
-- Checks whether two variables have the same arity
sameArity :: VarSymb -> VarSymb -> Bool
sameArity v v' = symbAr (VS v) == symbAr (VS v')

renaming :: VarSymb -> VarSymb -> Maybe Renaming
renaming v v' | sameArity v v' = Just $ R (v, v')
              | otherwise = Nothing

renameInVar :: Renaming -> VarSymb -> VarSymb
renameInVar (R (v, v')) w | w == v    = v'
                          | otherwise = w

applyRenaming :: Renaming -> Form -> Form
applyRenaming r (FT p ts) = FT p (map renameInTerm ts) where
  renameInTerm :: Term -> Term
  renameInTerm (T (VS w) ts') = T (VS (renameInVar r w)) ts'
  renameInTerm (T (FS f) ts') = T (FS f) (map renameInTerm ts')
applyRenaming _ _ = undefined -- Recursive cases handled by apply
```

Definition 4.2 (Fixation)**.** A *fixation* φ_f^F replaces a variable $F : s_1 \times \cdots \times s_n \to s \in V_n$ with a function symbol $f : s_1 \times \cdots \times s_n \to s \in Func_\Sigma$

$$F(t_1 : s_1, \ldots t_n : s_n) : s \xrightarrow{\varphi_f^F} f(t_1 : s_1, \ldots t_n : s_n) : s$$

```haskell
newtype Fixation = F (VarSymb, FunSymb) deriving (Eq, Show)

applyFixation :: Fixation -> Form -> Form
applyFixation (F (v, f)) (FT p ts) = FT p (map fixInTerm ts) where
  fixInTerm :: Term -> Term
  fixInTerm (T (VS w) ts') | w == v = T (FS f) ts'
                           | otherwise = T (VS w) ts'
  fixInTerm (T (FS f') ts') = T (FS f') (map fixInTerm ts')
applyFixation _ _ = undefined -- Recursive cases handled by apply
```

Definition 4.3 (Argument Insertion)**.** An *argument insertion* $\iota_{G,i}^{F,F'}$, with $0 \leq o \leq n$, $F : s_1 \times \cdots \times s_n \to s \in V_n$, $G : s_i \times \cdots \times s_{i+k-1} \to s_g \in V_k$ with $k \leq n - i$ and $F' : s_1 \times \cdots \times s_{i-1} \times s_g \times s_{i+k} \times \cdots \times s_n \to s \in V_{n-k+1}$ is defined as:

$$F(t_1 : s_1, \ldots t_n : s_n) : s \xrightarrow{\iota_{G,i}^{F,F'}}$$
$$F'(t_1 : s_1, \ldots t_{i-1} : s_{i-1}, G(t_i : s_i, \ldots, t_{i+k-1} : s_{i+k-1}) : s_g, t_{i+k} : s_{i+k}, \ldots, t_n : s_n) : s$$

```haskell
                                -- F,       F',      G,       i
newtype Insertion = AI (VarSymb, VarSymb, VarSymb, Int) deriving (Eq, Show)

applyInsertion :: Insertion -> Form -> Form
applyInsertion (AI (f, f', g, i)) (FT p ts) = FT p (map insertInTerm ts) where
  insertInTerm :: Term -> Term
  insertInTerm (T (VS v) ts') | v /= f = T (VS v) (map insertInTerm ts)
                              | otherwise = let
    k = length $ fst $ symbAr $ VS g -- Amount of arguments that g takes
    arguments = [ t | (j, t) <- zip [0..] ts', i <= j, j <= i+k-1 ] -- Arguments of f that
        will become arguments of g
    in T (VS f') [ if j == i then T (VS g) (map insertInTerm arguments) else insertInTerm
        t | (j, t) <- zip [0..] ts', j `notElem` [i+1..i+k-1] ]
  insertInTerm (T (FS f'') ts') = T (FS f'') (map insertInTerm ts')
applyInsertion _ _ = undefined -- Recursive cases handled by apply
```

Definition 4.4 (Permutation)**.** A *permutation* $\pi_\alpha^{F,F'}$ with variables $F : s_1 \times \cdots \times s_n \to s \in V_n$ and $F' : s_1 \times \cdots \times s_n \to s \in V_n$, together with a bijective function $\alpha : \{1, \ldots, n\} \to \{1, \ldots, n\}$ (which is not the identity function), rearranges the arguments of a term:

$$F(t_1 : s_1, \ldots t_n : s_n) : s \xrightarrow{\pi_\alpha^{F,F'}} F'(t_{\alpha(1)} : s_{\alpha(1)}, \ldots t_{\alpha(n)} : s_{\alpha(n)}) : s$$

```
newtype Permutation = P (VarSymb, VarSymb, [(Int, Int)]) deriving (Eq, Show)
```

The following recursive helper function permutes a list, given a function from indices of that list to indices of that list, provided as a list of tuples. Because of the recursive charcater of this helper function the list needs to be given twice as an argument.

We leave it up to the user to define semantically correct subsitutions. Lifting this reponsibility to the type level (and thus compile time) is unfortunately out of scope due to the lean design of our types.

```
unsafeLookup :: Eq a => a -> [(a, b)] -> b
unsafeLookup key dict = case lookup key dict of
  Just value -> value
  Nothing -> error "Key not in dictionary"

permute :: [a] -> [a] -> [(Int, Int)] -> [a]
permute [] _ _ = []
permute (_:xs) l f = l!!unsafeLookup (length l - (length xs + 1)) f : permute xs l f
```

The following function allows then to apply a permutation to a formula.

```
applyPermutation :: Permutation -> Form -> Form
applyPermutation p (FT pr ts) = FT pr (map (permInTerm p) ts) where
  permInTerm :: Permutation -> Term -> Term
  permInTerm (P (v, v', f)) (T (VS w) ts') | v == w = T (VS v') (permute ts' ts' f)
                                           | otherwise = T (VS w) ts'
  permInTerm r (T (FS f) ts') = T (FS f) (map (permInTerm r) ts')
applyPermutation _ _ = undefined -- Recursive cases handled by apply
```

4.2 LGGs and the Basic Substitutions

The authors of [SKGK14] claim that once we allow for restricted higher-order substitution, we can no longer speak of a single least general generalization. This claim presents a number of issues. First, lggs, as defined in [Plo70], are necessarily unique. When this uniqueness property is removed, it isn't clear what precisely the authors mean when they refer to an lgg. Second, due to this problem of definition, it is difficult to verify any relationship between higher-order substitutions and the uniqueness of lggs. The authors provide no proof, nor can one be found in a search of related literature ([KSGK07], [SKKG09]). The closest we find to a justification of this claim is a figure on page 174 of [SKGK14], in which the authors claim that the pair of terms below

$$f(g(a,b,c),d), \quad f(d,h(a))$$

have three most specific anti-unifiers:

$$f(X,Y), \quad F(d,G(a)) \quad F'(G(a),d)$$

How these all three of these terms can be *the* most specific anti-unifier of the pair of terms above is not entirely clear.

In lieu of more clarity from the literature, we can only attempt to provide a new definition that we hope might be a better fit for the authors' intuitions. First, we can define generalizations as we expect the authors intend:

Definition 4.5 (Restricted Second-Order Generalization)**.** Let s, t be first-order terms (resp. formulae). A *restricted second-order generalization* (rsog) is a triple $g = \langle a, \sigma, \tau \rangle$ where a is a term (resp. formula) a and σ, τ are of the form $bs_1 \circ \cdots \circ bs_n$ where for $1 \leq i \leq n$, bs_i is a basic substitution, and such that $s \xleftarrow{\sigma} a \xrightarrow{\tau} t$. We say that a is an *anti-unifier* of $\{s, t\}$.

Next, as lggs constitute limits in the category of substitutions, and the authors appear to be speaking of sets of lggs, we're reminded of the categorical notion of a *multi-limit*:

Definition 4.6 (Multi-Lgg)**.** For terms s, t, we define a *multi-lgg* to be a set of tuples $ML = \{\langle a_1, \sigma_1, \tau_1 \rangle, \ldots, \langle a_n, \sigma_n, \tau_n \rangle\}$ such that, for every generalization $g' = \langle a', \sigma', \tau' \rangle$, there exists a (unique) chain of basic substitutions $\gamma : a' \to a_i$ for a unique $a_i \in ML$, such that $\sigma' = \sigma_i \circ \gamma$ and $\tau' = \tau_i \circ \gamma$.

Proving whether first-order terms and chains of basic substitutions constitute a category, or whether multi-lggs can indeed be constructed for any set of first-order terms, is unfortunately beyond the scope of the current report. We include these remarks only for the purpose of guiding future research.

In the present work, we shall continue by proposing an algorithm to produce restricted second-order generalizations which are helpfully specific, something we will call *not too general generalizations*, or *ntgg*s for short.

Definition 4.7 (Not Too General Generalization)**.** For terms s, t, we call a rsog $g = \langle a, \sigma, \tau \rangle$ a *not too general generalization (ntgg)*, if for some suitably large number of rsogs $g' = \langle a', \sigma', \tau' \rangle$, there exists a chain of basic substitutions $\gamma : a' \to a$ such that $\sigma' = \sigma \circ \gamma$ and $\tau' = \tau \circ \gamma$. We say that a is the *specific enough anti-unifier* of $\{s, t\}$.

Obviously, the use of the term "suitably large" in this definition is not as precise as could be hoped for, but for the present report, this level of vagueness is useful.

4.3 Generating NTGGs

As a compromise between taking advantage of the existing literature and accounting for the needs of HDTP, we have attempted to re-purpose the algorithm in [TTF13] to create our ntggs.

4.4 Alignments, Complexity of NTGGs, and Domain Generalizations

To create an analogy between two domains, there needs to be a method for selecting which formulae from the source domain should be seen as analogous to which formulae in the target domain. [SKGK14] calls for a process for generating an *alignment* between domains:

Definition 4.8 (Alignment)**.** Given two domains $\mathcal{D}_\alpha, \mathcal{D}_\beta$, an *alignment* is a set of pairs of formulae $[\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_n, \beta_n \rangle]$, with $\alpha_i \in \mathcal{D}_\alpha$ and $\beta_i \in \mathcal{D}_\beta$.

The algorithm in place so far simply finds an ntgg for every pair of formulae $(\alpha, \beta) \in \mathcal{D}_\alpha \times \mathcal{D}_\beta$. What is needed is a means to find the "best-fitting" pairs of formulae. We will do so by making use of the notion of "complexity of generalization" discussed in [SKGK14]:

Definition 4.9 (Complexity of Substitutions)**.** We define the complexity of a basic substitution bs as:

$$\mathcal{C}(bs) = \begin{cases} 0 & \text{if } bs \text{ is a renaming} \\ 1 & \text{if } bs \text{ is a fixation} \\ k+1 & \text{if } bs \text{ is an argument insertion, and} \\ & \quad \text{the inserted argument is a variable of arity } k \\ 1 & \text{if } bs \text{ is a permutation} \end{cases}$$

The complexity of a composition of basic substitutions is simply the sum of each basic substitution composed:

$$\mathcal{C}(bs_1 \circ \cdots \circ bs_n) = \sum_{i=1}^{n} \mathcal{C}(bs_i)$$

Definition 4.10 (Complexity of Generalizations)**.** Let $g = \langle a, \sigma, \tau \rangle$ be an ntgg for a pair of terms $\{s, t\}$. We define the complexity of g as $\mathcal{C}(\langle a, \sigma, \tau \rangle) = \mathcal{C}(\sigma) + \mathcal{C}(\tau)$.

The following functions compute the complexity of a single substitution, of a list of substitutions and of a generalisation.

```
type Comp = Int

cSimple :: Sub -> Comp
cSimple (SR _) = 0
cSimple (SF _) = 1
cSimple (SI (AI (_, _, g, _))) = length (fst (symbAr(VS g))) + 1
cSimple (SP _) = 1

cList :: [Sub] -> Comp
cList [] = 0
cList xs = foldr ((+) . cSimple) 0 xs

cGen :: Gen -> Comp
cGen (_, s, s') = cList s + cList s'
```

Finally, we have a function that computes the generalisation with least complexity given a list of generalisations.

```
prefGen :: [Gen] -> Gen
prefGen = minimumBy (\gen gen' -> cGen gen `compare` cGen gen')
```

The intuition for this measure, as described in the original work, is that human subjects appear to have more difficulty processing the final three substitutions compared to the first.

In the original [SKGK14], the complexity of a generalization was used to find a "Preferred Generalization" among the "multiple lggs" constructed for any pair of formulae. As previously discussed, this notion of "multiple lggs" wasn't sufficiently well-defined, and so was replaced with our notion of "not too general generalization". We can now use the complexity of an ntgg as a metric to compare the desirability of different alignments, as we can select the "best" alignment as the one whose generalizations are least costly.

This is precisely the tactic we use to construct the basic framework for forming an analogy between two domains, [SKGK14]'s *domain generalization*:

Definition 4.11 (Domain Generalization)**.** Given an alignment $[\langle\alpha_1, \beta_1\rangle, \ldots, \langle\alpha_n, \beta_n\rangle]$, a *domain generalization* is a set of ntggs $\mathcal{D}_g = [g_1, \ldots g_n]$, where for $1 \leq i \leq n$, $g_i = \langle a_i, \sigma_i, \tau_i\rangle$ such that $\alpha_i \xleftarrow{\sigma_i} a_i \xrightarrow{\tau_i} \beta_i$.

What follows is the core of our implementation: an algorithm capable of generating such alignments. Due to the differences between theory and implementation, our function not only returns the align formulae, but also the generalizations used to produce them. The algorithm calculates all the NTGGs from all possible pairs between a source domain with n different formulae, and a target domain with m formulae. In order to maintain the amount of generated formulae manageable, we (somewhat arbitrarily) pick the m pairs with the lowest complexity. This upper bound of m is our interpretation of the "not too general" part of ntggs.

```
-- output: [(0,1,2,3,4)] such that 2 <-1- 0 -3-> 4
align :: Domain -> Domain -> [(Form, [Sub], Form, [Sub], Form)]
align d d' = take (length d') $ sortBy (\p p' -> complexity p `compare` complexity p') [
    result phi psi | phi <- d, psi <- d' ] where
  complexity (_, subPhi, _, subPsi, _) = cList subPhi + cList subPsi
  result phi psi = (antiUnifier, concat $ sourceSubsOf $ map (termToFormGen phi) subs, phi
      , concat $ targetSubsOf $ map (termToFormGen psi) subs, psi) where
    (antiUnifier, subs) = lambda phi psi []

alignsToTransfers :: [(Form, [Sub], Form, [Sub], Form)] -> [Gen]
alignsToTransfers = map alignToTransfer where
  alignToTransfer :: (Form, [Sub], Form, [Sub], Form) -> Gen
  alignToTransfer (antiUnifier, subPhi, _, subPsi, _) = (antiUnifier, subPhi, subPsi)
```

One of the artifacts of combining the algorithm of [TTF13, 3] in conjunction with the (higher-order) theoretical frame from [SKGK14] is that we have to adapt the output of `align` to use the four basic substitutions. On paper it is easy to imagine how one would approach this. The Haskell implementation is a bit convoluted, but it tries to obtain the simplest chain of basic substitutions that, applied one after the other, result in the conceptual substitution given by [TTF13, 3].

```
type Gen = (Form, [Sub], [Sub])
-- Term generalization from Tabareau
type TermGen = (VarSymb, Term, Term)

-- Variables in a given form
varsInForm :: Form -> [VarSymb]
varsInForm (FT (PS _) ts) = concatMap varsInTerm ts
varsInForm (Not phi) = varsInForm phi
varsInForm (Disj phi psi) = varsInForm phi ++ varsInForm psi
varsInForm (Forall _ phi) = varsInForm phi

-- Variables in a given term
varsInTerm :: Term -> [VarSymb]
varsInTerm (T (VS v) ts) = v:concatMap varsInTerm ts
```

```
varsInTerm (T _ ts) = concatMap varsInTerm ts

-- First argument: the "context" formula in which we apply the second argument
-- Second argument: t <- x -> u
-- Result: the context formula and the corresponding substitutions in it
termToFormGen :: Form -> TermGen -> Gen
termToFormGen context (vs, t, u) = (context, aux context vs t, aux context vs u) where
  -- First argument: the "context" formula in which we apply the second and third argument
  -- Second argument: x
  -- Third argument: t
  -- Result: the (chain of) substitutions that result in x -> t inside of context
  aux :: Form -> VarSymb -> Term -> [Sub]
  aux phi vs' (T (FS fs) ts) = SF (F (newVar, fs)) : termToFormGenRec (newVar:varsInForm
      phi) vs' (T (VS newVar) ts) where
    newVar = newVariable (varsInForm phi)
  aux _ _ _ = undefined -- We only apply aux to function symbol terms

termToFormGenRec :: [VarSymb] -> VarSymb -> Term -> [Sub]
termToFormGenRec vars vs (T (VS vs') (T (FS fs) _:ts)) = SF (F (newVar, fs)) : SI (AI (
    newVar', vs', newVar, 0)) : termToFormGenRec (newVar':newVar:vars) vs (T (VS vs') ts)
    where
  newVar = newVariable vars
  newVar' = newVariable (newVar:vars)
termToFormGenRec _ vs (T (VS vs') []) = [SR (R (vs, vs'))]
termToFormGenRec _ _ _ = undefined -- Conversely, we only apply termToFormGenRec to
    variable symbol terms
```

5 Analogical Transfer

As stated previously, the ultimate purpose of generalizations is to allow an agent to form analogies between two domains of knowledge, and then to leverage her knowledge of the source domain to infer new formulae in the target domain. The present section will detail how we model this process in our Haskell implementation of HDTP.

For the most part, [SKGK14] provides few details about analogical transfer in HDTP, but drawing on sources such as [SKKG09], we can infer that analogical transfer is meant to be the composition of two chains of substitutions: first, for any given formula in the source domain $\alpha_i \in \mathcal{D}_\alpha$, apply $\sigma_i^{-1} : \alpha_i \to a_i$ to "send" α_i to its specific enough anti-unifier in the domain generalization $a_i \in \mathcal{D}_g$. Second, apply any number of the chains of substitutions $\{\tau_1, \ldots \tau_n\}$ in $\mathcal{D}_g$ in order to produce a new formula $\beta \in \mathcal{D}_\beta$ in the target domain.

Our implementation of `transfer` simply applies all the generalizations obtained from the alignment to all the formulae from the source domain. As we have already cut down on the amount of generalizations in the alignment algorithm, we are safe to return all these obtained formulae as the result of this automated analogical transfer.

```
targetSubsOf :: [Gen] -> [[Sub]] -- collects all the ``right projections'', the
    substitutions to the target domain
targetSubsOf [] = []
targetSubsOf gens = map (\(_, _, x) -> x) gens

sourceSubsOf :: [Gen] -> [[Sub]] -- collects all the ``left projections'', the
    substitutions to the source domain
sourceSubsOf [] = []
sourceSubsOf gens = map (\(_, x, _) -> x) gens

-- generalized domain -> analogical pairs (s,t'), where t' is the expanded target domain
```

```
transfer :: [Gen] -> [(Form, Form)]
transfer gens = [ (apply' s g, apply' t' g) | (g, s, _) <- gens, (_, _, t') <- gens ]
```

6 Testing the Algorithm

Based on the examples in [SKGK14] we implemented test cases with `Hspec` for the basic substituations and the ntgg, alignment and transfer algorithms. The tests can be run with `stack clean && stack test --coverage`. The code block below is an example for a test case of our ntgg algorithm.

```
CaseLambda {
  descriptionLambda = "revolves_around(earth, sun), revolves_around(electron, nucleus) ->
      revolves_around(X, Y)"
, inputLambda       = (FT (PS "revolves_around") [T (FS "earth") [], T (FS "sun") []], FT
    (PS "revolves_around") [T (FS "electron") [], T (FS "nucleus") []], [])
, expectedLambda    = (FT (PS "revolves_around") [T (VS "X") [], T (VS "Y") []], [("Y",T
   (FS "sun") [],T (FS "nucleus") []),("X",T (FS "earth") [],T (FS "electron") [])])
}
```

7 Conclusion and Future Work

We began this project because of our appreciation for what the authors of HDTP had been able to accomplish. We had hoped that by trying to translate some of their ideas into a functional programming context, we could learn more about HDTP in specific, and analogical reasoning more generally.

Like logic as a practice, the process of implementing an idea in code can be an excellent way to diagnose mistaken assumptions, flaws in reasoning, and fuzzy, ill-defined concepts. Our experience in trying to implement HDTP in Haskell was a perfect example of this. While the source material is an interesting analysis of analogical reasoning, the authors' attempt to formalize this often resulted in more confusion than clarity. As we've learned, translating something into formal language does not in and of itself guarantee the benefits of formal reasoning we described above.

Over the course of the project, we have had to fill in a number of conceptual holes which were not evident in the original papers. Conflations between terms and formulae, vagueness in the use of specific technical terms (like least general generalization), and a lack of proper specfication for some of the algorithms alluded to made the project more difficult than we expected. We did our best to remain true to the original work, while still settling issues where they arose.

We of course were limited as well in our implementation. For one, we have only tested our algorithm on examples from the original work, which may not present a full picture of either the successes or limitations of this model. In addition, time constraints prevented us from exploring more concrete design choices, such as in the definition of not too general generalizations, which remains vague.

In the end, with a bit of flexibility, we were happy to have created a Haskell program to see a system like HDTP in action. As well, we are proud to be pioneers in the application of Haskell to analogical reasoning. It is our hope that a program like ours might only be the beginning of more work on bringing HDTP into the broader computational and logical community.

References

[BP15] Tarek R. Besold and Enric Plaza. Generalize and blend: Concept blending based on generalization, analogy, and amalgams. In *ICCC*, 2015.

[Gen83] Dedre Gentner. Structure-mapping: A theoretical framework for analogy*. *Cognitive Science*, 7:155–170, 1983.

[GHK01] Dedre Gentner, Keith J. Holyoak, and Boicho N. Kokinov. *The Analogical Mind: Perspectives from Cognitive Science*. The MIT Press, 03 2001.

[HM95] Douglas Richard Hofstadter and Melanie Mitchell. The copycat project: a model of mental fluidity and analogy-making. In *Fluid Concepts and Creative Analogies*, 1995.

[HS13] Douglas R Hofstadter and Emmanuel Sander. *Surfaces and essences: Analogy as the fuel and fire of thinking*. Basic Books, 2013.

[KSGK07] Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In *Australian Conference on Artificial Intelligence*, 2007.

[Plo70] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence 5*, pages 153–163. American Elsevier, 1970.

[SKGK14] M. Schmidt, U. Krumnack, H. Gust, and K.U. Kühnberger. Heuristic-driven theory projection: An overview. In H. Prade and G. Richard, editors, *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548, pages 163–194. Springer, 2014.

[SKKG09] Angela Schwering, Ulf Krumnack, Kai-Uwe Kühnberger, and Helmar Gust. Syntactic principles of heuristic-driven theory projection. *Cognitive Systems Research*, 10:251–269, 2009.

[TTF13] Nicolas Tabareau, Éric Tanter, and Ismael Figueroa. Anti-unification with type classes. In *Journees Francophones des Langages Applicatifs (JFLA)*, 2013.