BAMBERGER BEITRÄGE

ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK

ISSN 0937-3349

Nr. 81

AAIP 2009
Proceedings of the ACM SIGPLAN Workshop on Approaches and Applications of Inductive Programming

Ute Schmid, Emanuel Kitzelmann, and

Rinus Plasmeijer (Eds.)

Affiliated with the 14th ACM SIGPLAN
International Conference on Functional Programming
(ICFP 2009)
Edinburgh, Scotland

September 4, 2009

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK

OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

Preface

Inductive programming is concerned with the automated construction of declarative, often functional, recursive programs from incomplete specifications such as input/output examples. The inferred program must be correct with respect to the provided examples in a generalising sense: it should be neither equivalent to it, nor inconsistent. Inductive programming algorithms are guided explicitly or implicitly by a language bias (the class of programs that can be induced) and a search bias (determining which generalised program is constructed first). Induction strategies are either generate-and-test or example-driven. In generate-and-test approaches, hypotheses about candidate programs are generated independently from the given specifications. Program candidates are tested against the given specification and one or more of the best evaluated candidates are developed further. In analytical approaches, candidate programs are constructed in an example-driven way. While generate-and-test approaches can – in principle – construct any kind of program, analytical approaches have a more limited scope. On the other hand, efficiency of induction is much higher in analytical approaches.

Inductive programming is still mainly a topic of basic research, exploring how the intellectual ability of humans to infer generalised recursive procedures from incomplete evidence can be captured in the form of synthesis methods. Intended applications are mainly in the domain of programming assistance – either to relieve professional programmers from routine tasks or to enable non-programmers to some limited form of end-user programming. Furthermore, in future inductive programming techniques might be applied to further areas such as support inference of lemmata in theorem proving or learning grammar rules.

Inductive automated program construction has been originally addressed by researchers in artificial intelligence and machine learning. During the last years, some work on exploiting induction techniques has been started also in the functional programming community. Therefore, the third workshop on Äpproaches and Applications of Inductive Programmingtook place for the first time in conjunction with the ACM SIGPLAN International Conference on Functional Programming (ICFP 2009). The first and second workshop were associated with the International Conference on Machine Learning (ICML 2005) and the European Conference on Machine Learning (ECML 2007).

AAIP'09 aimed to bring together researchers from the field of inductive functional programming from the functional programming and the artificial intelligence communities and advance fruitful interactions between these communities with respect to programming techniques for inductive programming algorithms, identification of challenge problems and potential applications.

The program committee consisted of members from both communities, namely:

The workshop was enriched by three invited talks from members from the functional programming community and we want to thank Lennart Augustsson, Pieter Koopman and Neil Mitchell for their support. We are very grateful to Martin Hofmann who invested much of his time to support the workshop organisation. Furthermore, we want to thank all presenters for submitting their work to our workshop and all attendants for stimulating discussions. For everybody interested in inductive programming we recommend to visit the website: `www.inductive-programming.org`.

September 2009 Ute Schmid, Emanuel Kitzelmann and Rinus Plasmeijer

AAIP'09 Workshop Organisers

3rd workshop on

Approaches and Applications of Inductive Programming

Schedule

09:00	Welcome	
09:05 – 10:00	Invited Talk	„The Theorem Prover Djinn“ Lennart Augustsson *Standard Chartered Bank*
10:00 – 10:30		*Tea and coffee break*
10:30 – 11:00	Paper	„Inductive Programming – A Survey“ Emanuel Kitzelmann
11:00 – 12:00	Invited Talk	„Deriving a DSL from One Example“ Neil Mitchell *Standard Chartered Bank*
12:00 – 13:30		*Lunch break*
13:30 – 14:00	Paper	„Incremental Learning in Inductive Programming“ Robert Henderson
14:00 – 14:30	Paper	„Enumerating Well-Typed Terms Generically“ Alexey Rodriguez Yakushev, Johan Jeuring
14:30 – 15:00	Paper	"Defining Inductive Operators Using Distances over Lists" V. Estruch, C. Ferri, J. Hernandez-Orallo, M.J. Ramirez-Quintana
15:00 – 15:30		*Tea and coffee break*
15:30 – 16:30	Invited Talk	„Synthesis of Functions Using Generic Programming“ Pieter Koopman *University of Nijmegen*
16:30 – 17:00	Paper	„Porting IgorII from Maude to Haskell“ Martin Hofmann, Emanuel Kitzelmann, Ute Schmid
17:00 – 17:30	Paper	„Automated Method Induction – Functional Goes Object Oriented“ Thomas Hieber, Martin Hofmann
17:30 – 18:00	Report	Work in Progress Reports on MagicHaskeller Susumu Katayama
18:00	Finish (latest)	
later		*post-workshop meeting and get together*

Contents

Putting Curry-Howard to work

Lennart Augustsson
Standard Chartered Bank
lennart.augustsson@gmail.com

Abstract

It is a well known fact that there is a correspondence between propositions and type, and similarly a correspondence between a proof of a proposition and a program of a type; this is the Curry-Howard correspondence. In this talk I will describe a program, Djinn, which takes a Haskell type and produces a program of that type, using the Curry-Howard correspondence. For the subset of Haskell types that Djinn can handle (no recursive types) it does quite well. For instance, it can derive the code for all the standard monads in Haskell, including continuations and call/cc.

Deriving a Relationship from a Single Example

Neil Mitchell

ndmitchell@gmail.com

Abstract

Given an appropriate domain specific language (DSL), it is possible to describe the relationship between Haskell data types and many generic functions, typically type-class instances. While describing the relationship is possible, it is not always an easy task. There is an alternative – simply give one example output for a carefully chosen input, and have the relationship derived.

When deriving a relationship from only one example, it is important that the derived relationship is the intended one. We identify general restrictions on the DSL, and on the provided example, to ensure a level of predictability. We then apply these restrictions in practice, to derive the relationship between Haskell data types and generic functions. We have used our scheme in the DERIVE tool, where over 60% of type classes are derived from a single example.

1. Introduction

In Haskell (Peyton Jones 2003), *type classes* (Wadler and Blott 1989) are used to provide similar operations for many data types. For each data type of interest, a user must define an associated instance. The instance definitions usually follow a highly regular pattern. Many libraries define new type classes, for example Trinder et al. (1998) define the NFData type class, which reduces a value to normal form. As an example, we can define a data type to describe some computer programming languages, and provide an NFData instance:

```
data Language = Haskell [Extension] Compiler
              | Whitespace
              | Java Version

instance NFData Language where
  rnf (Haskell x₁ x₂) = rnf x₁ `seq` rnf x₂ `seq` ()
  rnf (Whitespace   ) = ()
  rnf (Java x₁      ) = rnf x₁ `seq` ()
```

We also need to define NFData instances for the data types Extension, Compiler and Version. Any instance of NFData follows naturally from the structure of the data type: for each constructor, all fields have seq applied, before returning ().

Writing an NFData instance for a single simple data type is easy – but for multiple complex data types the effort can be substantial. The standard solution is to express the *relationship* between a data type and it's instance. In standard tools, such as DrIFT (Winstanley 1997), the person describing a relationship must be familiar with both the representation of a data type, and various code-generation functions. The result is that specifying a relationship is not as straightforward as one might hope.

Using the techniques described in this paper, these relationships can often be automatically inferred from a single example. To define the generation of *all* NFData instances, we require an example to be given for the Sample data type defined in Figure 1:

```
data Sample α = First
              | Second α α
              | Third  α
```

Figure 1. The Sample data type.

```
instance NFData α ⇒ NFData (Sample α) where
  rnf (First        ) = ()
  rnf (Second x₁ x₂) = rnf x₁ `seq` rnf x₂ `seq` ()
  rnf (Third x₁     ) = rnf x₁ `seq` ()
```

The NFData instance for Sample follows the same pattern as for Language. From this example, we can infer the general relationship. However, there are many possible relationships between the Sample data type and this result – for example the function might always generate the instance for Sample, regardless of the input type. We overcome this problem by requiring the relationship to be written in a domain specific language (DSL), and that the example has certain properties (see §2). With our restrictions, we can regain predictability.

1.1 Contributions

This paper makes the following contributions:

- We describe a scheme which allows us to infer predictable and correct relationships (§2).
- We describe how this scheme is applicable to instance generation, both in a high-level manner (§3), and more detailed practical concerns (§5).
- We outline a method for deriving a relationship in our DSL, without resorting to unguided search (§4).
- We give results (§6), including reasons why our inference fails (§6.1). In our experience, over 60% of Haskell type classes can be derived using our method.

2. Our Derivation Scheme

In this section we define a general scheme for deriving functions, which we later use to derive type-class instance generators. In general terms, a function takes an input to an output. In our case, we restrict ourselves to functions that can be described by a DSL (domain specific language). We need an apply function to serve as an interpreter for our DSL, which takes a DSL and an input and produces an output. Our scheme can be implemented in Haskell as follows:

```
data Input
data Output
data DSL

apply :: DSL → Input → Output
```

Now we turn to the derivation scheme. Given a single result of the Output type, for a particular sample Input, we wish to derive a suitable DSL. It may not be possible to derive a suitable DSL, so our derivation function must allow for the possibility of failure. Instead of producing at most one DSL, we instead produce a list of DSLs, following the lead of Wadler (1985). Once again, we can implement this in Haskell as:

```
sample :: Input
derive  :: Output → [DSL]
```

We require our scheme to have two properties – correctness (it works) and predictability (it is what the user intended). We now define both of these properties more formally, along with restrictions necessary to achieve them.

2.1 Correctness

The derivation of a particular output is correct if all derived DSLs, when applied to the sample input, produce the original output:

$\forall\ \mathsf{o} \in \mathsf{Output} \bullet \forall\ \mathsf{d} \in \mathsf{derive\ o} \bullet \mathsf{apply\ d\ sample} \equiv \mathsf{o}$

Given an existing derive′ function, which does not necessarily ensure correctness, we can create a correct version by filtering out the incorrect DSLs. By applying this modification we can remove some constraints from the derive′ function – either simplifying the implementation, or gaining a higher assurance of correctness.

$\mathsf{derive\ o} = [\mathsf{d} \mid \mathsf{d} \leftarrow \mathsf{derive'\ o}, \mathsf{apply\ d\ sample} \equiv \mathsf{o}]$

2.2 Predictability

A derived relationship is predictable if the user can be confident that it matches their expectations. In particular, we don't want the user to have to understand the complex derive function to gain predictability. In this section we attempt to simplify the task of defining predictable derivation schemes.

Before defining predictability, it is useful to define congruence of DSLs. We define two DSLs to be congruent ($\cong$), if for every input they produce identical results – i.e. apply $\mathsf{d}_1 \equiv$ apply d_2.

$\mathsf{d}_1 \cong \mathsf{d}_2 \Longleftrightarrow \forall\ \mathsf{i} \in \mathsf{Input} \bullet \mathsf{apply\ d}_1\ \mathsf{i} \equiv \mathsf{apply\ d}_2\ \mathsf{i}$

Our derive function returns a list of suitable DSLs. To ensure consistency, it is important that the DSLs are all congruent – allowing us to choose any DSL as the answer.

$\forall\ \mathsf{o} \in \mathsf{Output} \bullet \forall\ \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{derive\ o} \bullet \mathsf{d}_1 \cong \mathsf{d}_2$

This property is dependent on the implementation of the derive function, so is insufficient for ensuring predictability. To ensure predictability we require that all results satisfying the correctness property are congruent:

$\forall\ \mathsf{d}_1, \mathsf{d}_2 \in \mathsf{DSL} \bullet$
$\quad \mathsf{apply\ d}_1\ \mathsf{sample} \equiv \mathsf{apply\ d}_2\ \mathsf{sample} \Rightarrow \mathsf{d}_1 \cong \mathsf{d}_2$

The combination of this predictability property and the correctness property implies the consistency property. It is important to note that predictability does not impose conditions on the derive function, only on the DSL and sample input. The sample input is chosen by the author of the derivation scheme, so the user is not required to understand the reasons for it's form. To ensure predictability the user may have to know some details about the DSL, but hopefully these will not be too onerous.

2.3 Summary

If the predictability property holds for the DSL and sample value, and we use the modified derive in terms of derive′, then any result produced by derive will be a valid relationship. These properties allow us to write the derive function focusing on other attributes (which we discuss in §4).

To use this general scheme, we need to instantiate it to our particular problem (§3), check the predictability property (§3.4), and implement a derive function (§4).

3. Deriving Instances

In this section we apply the scheme from §2 to the problem of deriving type class instances. We let the output type be Haskell source code and the input type be a representation of algebraic data types. The DSL contains features such as list manipulation, constant values, folds and maps. We first describe each type in detail, then discuss the restrictions necessary to satisfy the predictability property.

3.1 Output

We wish to generate any sequence of Haskell declarations, where a declaration is typically a function definition or type class instance. There are several ways to represent a sequence of declarations:

String A sequence of Haskell declarations can be represented as the string of the program text. However, the lack of structure in a string poses several problems. When constructing strings it is easy to generate invalid programs, particularly given the indentation and layout requirements of Haskell. It is also hard to recover structure from the program that is likely to be useful for deriving relationships.

Pretty printing combinators Some tools such as DrIFT (Winstanley 1997) generate Haskell code using pretty printing combinators. These combinators supply more structure than strings, but the structure is linked to the presentation, rather than the meaning of constructs.

Typed abstract syntax tree (AST) The standard way of working with Haskell source code is using a typed AST – an AST where different types of fragment (i.e. declarations, expressions and patterns) are restricted to different positions within the tree. The first version of DERIVE used a typed AST, specifically Template Haskell (Sheard and Peyton Jones 2002). This approach preserves all the structure, and makes it reasonably easy to ensure the generated program is syntactically correct. By combining a typed AST with a parser and pretty printer we can convert between strings as necessary.

Untyped abstract syntax tree (AST) An untyped AST is an AST where all fragments have the same type, and types do not restrict where a fragment may be placed. The removal of types increases the number of invalid programs that can be represented – for example a declaration could occur where an expression was expected. However, by removing types we increase the similarity of the tree, in turn simplifying function that operate on the tree in a uniform manner.

For our purposes, it is clear that both strings and pretty printing combinators are unsuitable – they lack sufficient structure to implement the derive operation. The choice between a typed and untyped AST is one of safety vs simplicity. The use of a typed AST in the first version of DERIVE caused many complexities – notably the DSL was hard to represent in a well-typed manner and some functions had to be duplicated for each type. The loss of safety from using an untyped AST is not too serious, as both DSLs and ASTs are automatically generated, rather than being written by hand. Therefore, we chose to use untyped ASTs for the current version of DERIVE. We discuss possible changes to regain type safety in §8.

While we work internally with an untyped AST, existing Haskell libraries for working with ASTs use types. To allow the use of existing libraries we start from a typed AST and collapse it to a single type, using the Scrap Your Boilerplate generic programming library (Lämmel and Peyton Jones 2003, 2004).

The use of Template Haskell in the first version of DERIVE provided a number of advantages – it is built in to GHC and can represent a large range of Haskell programs. Unfortunately, there were also a number of problems:

- Being integrated in to GHC ensures Template Haskell is available everywhere GHC is, but also means that Template Haskell cannot be upgraded separately. Users of older versions of GHC cannot take advantage of improvements to Template Haskell, and every GHC upgrade requires modifications to DERIVE.
- Template Haskell does not support new GHC extensions – they are often implemented several years later. For example, Template Haskell does not yet support view patterns.
- Template Haskell allows generated instances to be used easily by GHC compiled programs, but it makes the construction of a standalone preprocessor harder.
- If Template Haskell is also used to read the input data type (as it was in the first version of DERIVE) then only data types contained in compilable modules can be used. In particular, all necessary libraries must be compiled before an instance could be generated.
- The API of Template Haskell is relatively complex, and has some inconsistencies. In particular the Q monad caused much frustration.

We have implemented the current version of DERIVE using the haskell-src-exts library (Broberg 2009). The haskell-src-exts library is well maintained, supports most Haskell extensions [1] and operates purely as a library. We convert the typed AST of haskell-src-exts to a universal data type:

```
data Output = OString String
            | OInt Int
            | OList [Output]
            | OApp String [Output]
```

OString and OInt represent strings and integers. The OList constructor generates a list from a sequence of Output values. The expression OApp c xs represents the constructor c with fields xs. For example Just [1, 2] would be represented by the expression OApp "Just" [OList [OInt 1, OInt 2]].

Our Output type can represent many impossible values, for example the expression OApp "Just" [] (wrong number of fields) or OApp "Maybe" [] (not a constructor). We consider any Output value that does not represent a haskell-src-exts value to be an error. The root Output value must represent a value of type [Decl]. We can translate between our Output type and the haskell-src-exts type [Decl]:

```
toOutput   :: [Decl] → Output
fromOutput :: Output → [Decl]
```

We have implemented these functions using the SYB generics library (Lämmel and Peyton Jones 2004), specifically we have implemented the more general:

```
toOut   :: Data α ⇒ α → Output
fromOut :: Data α ⇒ Output → α
```

These functions are partial – they only succeed if the Output value represents a well-typed haskell-src-exts value. When operating on the Output type, we are working without type safety. However, provided all DSL values are constructed by derive, and that derive only constructs well-formed DSL values, our fromOutput function will be safe.

[1] Haskell-src-exts supports even more extensions than GHC!

3.2 Input

While the output type is largely dictated by the need to generate Haskell, we have more freedom with the input type. The input type represents Haskell data types, but we can choose which details to include, and thus which relationships we can represent. For example, we can include the module name in which the data type is defined, or we can omit this detail. We choose not to include the module name, which eliminates some derivations, for example the Typeable type class (Lämmel and Peyton Jones 2003).

Our Input type represents algebraic data types. We include details such as the arity of each constructor (ctorArity), the 0-based index of each constructor (ctorIndex) and the number of type variables (dataVars), but omit details such as types and record field names. Our Input type is:

```
data Input = Input
  {dataName :: String, dataVars :: Int, dataCtors :: [Ctor]}
data Ctor = Ctor
  {ctorName :: String, ctorIndex :: Int, ctorArity :: Int}
```

Values of Input for the Sample data type (Figure 1) and the Language data type (§1) are:

```
sampleType :: Input
sampleType = Input "Sample" 1
  [Ctor "First"  0 0
  , Ctor "Second" 1 2
  , Ctor "Third"  2 1]

languageType :: Input
languageType = Input "Language" 0
  [Ctor "Haskell"    0 2
  , Ctor "Whitespace" 1 0
  , Ctor "Java"       2 1]
```

The Input constructor contains the name of the data type, and the number of type variables the data type takes. For each constructor we record the name, 0-based index, and arity. These choices allow derivations to depend on the arity or index of a constructor, but not the types of a constructors arguments. In §6 we consider possible extensions to the Input type.

3.3 DSL

Our DSL type is given in Figure 2, and our apply function is given in Figure 3. The operations in the DSL are split in to six groups – we first give a high-level overview of the DSL, then return to each group in detail. The apply function is written in terms of applyEnv, where an environment is passed including the input data type, and other optional fields. Some functions in the DSL add to the environment (i.e. MapCtor), while others read from the environment (i.e. CtorName). Any operation reading a value from the environment must be nested within an operation placing that value in the environment.

Some operations require particular types – for example Reverse requires it's argument to evaluate to OList. Where possible we have annotated these restrictions in the DSL definition using comments. We have used view patterns, as implemented in GHC 6.10 (The GHC Team 2009), to perform matches on the evaluated argument DSLs. Our use of view patterns can be understood with the simple translation[2]:

[2] View-patterns and pattern-guards in GHC have different scoping behaviour, but this difference does not effect our apply function.

```
data DSL
        -- Constants
    = String String
    | Int Int
    | List [DSL]
    | App String DSL {-[α] -}
        -- Operations
    | Concat DSL {-[[α]] -}
    | Reverse DSL {-[α] -}
    | ShowInt DSL {-Int -}
        -- Fold
    | Fold DSL DSL
    | Head
    | Tail
        -- Constructors
    | MapCtor DSL
    | CtorIndex
    | CtorArity
    | CtorName
        -- Fields
    | MapField DSL
    | FieldIndex
        -- Custom
    | DataName
    | Application DSL {-[Exp] -}
    | Instance [String] String DSL {-[InstDecl] -}
```

Figure 2. DSL data type

```
f (Reverse (f → OList xs)) = ...
    ≡
f (Reverse v₁) | OList xs ← f v₁ = ...
    ≡
f (Reverse v₁) | case v₂ of OList { } → True; _ → False = ...
  where v₂ = f v₁; OList xs = v₂
```

Some operations have restrictions on what their arguments must evaluate to, and what environment values must be available. It would be possible to capture many of these invariants using either phantom types (Fluet and Pucella 2002) or GADTs (Peyton Jones et al. 2006). However, for simplicity, we choose not to.

3.3.1 Constants

We include constants in our DSL, so we can lift values of Output to values of DSL. The String, Int, List operations are directly equivalent to the corresponding Output values. The App constructor is similar to OApp, but instead of taking a *list* of arguments, App takes a single argument, which must to evaluate to an OList. Requiring an OList rather than an explicit list allows the arguments to App to be constructed by operations such as Reverse or Concat.

3.3.2 Operations

The operations group consists of useful functions for manipulating lists, strings and integers. The operations have been added as required, based on functions in the Haskell Prelude. The Concat operation corresponds to concat, and concatenates either a list of lists, or a list of strings. The Reverse operation performs reverse on a list. The ShowInt operation performs show, converting an integer to a string. We have only included functions for which we have found a specific need, for example Reverse cannot be applied to a string, even though there is a sensible interpretation. We do not

```
apply :: DSL → Input → Output
apply dsl input = applyEnv dsl Env {envInput = input}

data Env = Env {envInput :: Input
               , envCtor :: Ctor
               , envField :: Int
               , envFold :: (Output, Output) }

applyEnv :: DSL → Env → Output
applyEnv dsl env@(Env input ctor field fold) = f dsl
  where
  vars = take (dataVars input) $ map (:[]) ['a' ..]

  f (Instance ctx hd body) =
    OApp "InstDecl"
        [toOut
          [ClassA (UnQual $ Ident c) [TyVar $ Ident v]
           | v ← vars, c ← ctx]
        , toOut $ UnQual $ Ident hd
        , toOut [foldl TyApp
          (TyCon $ UnQual $ Ident $ dataName input)
          [TyVar $ Ident v | v ← vars]]
        , f body]

  f (Application (f → OList xs)) =
    foldl1 (λa b → OApp "App" [a, b]) xs

  f (MapCtor dsl) = OList [applyEnv dsl env {envCtor = c}
     | c ← dataCtors input]
  f (MapField dsl) = OList [applyEnv dsl env {envField = i}
     | i ← [1 .. ctorArity ctor]]

  f DataName = OString $ dataName input
  f CtorName  = OString $ ctorName ctor
  f CtorArity  = OInt     $ ctorArity ctor
  f CtorIndex  = OInt     $ ctorIndex ctor
  f FieldIndex  = OInt     $ field

  f Head = fst fold
  f Tail  = snd fold
  f (Fold cons (f → OList xs)) =
    foldr1 (λa b → applyEnv cons env {envFold = (a, b)}) xs

  f (List xs) = OList $ map f xs
  f (Reverse (f → OList xs)) = OList $ reverse xs
  f (Concat (f → OList [ ])) = OList [ ]
  f (Concat (f → OList xs)) = foldr1 g xs
        where g (OList    x) (OList    y) = OList    (x ++ y)
              g (OString x) (OString y) = OString (x ++ y)
  f (String x) = OString x
  f (Int x) = OInt x
  f (ShowInt (f → OInt x)) = OString $ show x
  f (App x (f → OList ys)) = OApp x ys
```

Figure 3. The apply function.

provide an append or $(+\!\!+)$ operation, but one can be created from a combination of List and Concat. These operations are all simple, and would be appropriate for many DSLs.

Some examples of these operations in use are:

```
Concat (List [String "hello ", String "world"])
  ≡ OString "hello world"
Reverse (List [Int 1, Int 2, Int 3])
  ≡ OList [OInt 3, OInt 2, OInt 1]
ShowInt (Int 42) ≡ OString "42"
```

3.3.3 Fold

The Fold operation corresponds to foldr1, but can be combined with Reverse to simulate foldl1. The first argument of Fold is a function – a DSL containing Head and Tail operations. The second argument must evaluate to a list containing at least one element. If the list has exactly one element, that is the result. If there is more than one element, then Head is replaced by the first element, and Tail is replaced by a fold over the remaining elements. This can be described by:

```
Fold fn [x] = x
Fold fn (x : xs) = fn [x / Head, Fold fn xs / Tail]
```

For example, to implement concat in terms of an Append operation would be Fold (Append Head Tail) (ignoring the case of the empty list). The fold operation is more complicated than the previous operations, but may still be useful to other DSLs.

3.3.4 Constructors

To insert information from the constructors we provide MapCtor. This operation generates a list, with the argument DSL evaluated once with each different constructor in the environment. The argument to MapCtor may contain CtorName, CtorIndex and CtorArity operations, which retrieve the information associated with the constructor. CtorName produces a string, while the others produce integers. An example of MapCtor on the Sample data type is:

```
MapCtor CtorName ≡ OList
  [OString "First", OString "Second", OString "Third"]
```

3.3.5 Fields

The MapField operation is similar to MapCtor, but maps over each field within a constructor. MapField is only valid within MapCtor. Within MapField, the FieldIndex operation returns the 1-based index of the current field. While most indexing in Haskell is 0-based, fields usually correspond to variable indices (i.e. x_1), which tend to be 1-based. As an example of MapField, using Second as the constructor in the environment:

```
Concat (List [List [CtorName],
  MapField (Concat (List [String "v", ShowInt FieldIndex]))])
  ≡ ["Second", "v1", "v2"]
```

3.3.6 Custom

The final set of operations are all specific to our particular problem. The simplest operation in this group is DataName, which returns the string corresponding to the name of the data type.

The second operation is Application. The haskell-src-exts library uses binary application, where multiple applications are often nested – we provide Application to represent vector application. Vector application is often used to call constructors with arguments resulting from MapField.

The final operation is Instance, and is used to represent a common pattern of instance declaration. For example, given the type Either α β, a typical instance declaration might be:

```
instance (Show α, Ord α, Show β, Ord β) ⇒
  ShowOrd (Either α β) where ...
```

This pattern requires each type variable to be a member of a set of type classes. The resulting instance construction is:

```
Instance ["Show", "Ord"] "ShowOrd" ...
```

The Instance fields describe which classes are required for each type variable (i.e. Show and Ord in this example), what the main class is (i.e. ShowOrd), and a DSL to generate the body. To specify this pattern without a specific Instance operation would require operations over type variables – something we do not support.

3.4 Restrictions for Predictability

To ensure predictability there must be no non-congruent DSL values which give equal results when applied to the sample input. Currently this invariant is violated – consider the counterexample DataName vs String "Sample". When applied to the sample input, both will generate OString "Sample", but when applied to other data types they generate different values. To regain predictability we impose three additional restrictions on the DSL:

1. The strings Sample, First, Second and Third cannot be contained in any String construction. Therefore, in the above example, String "Sample" is invalid.
2. All instances must be constructed with Instance.
3. Within MapCtor we require that the argument DSL *must* include CtorName.

We have already seen an example of the first restriction in practice, and the second restriction has similar motivation – to avoid making something constant when it should not be. Now let us examine the third restriction, with a practical example:

```
instance Arities (Sample α) where
  arities _ = [0, 2, 1]
```

Given this instance, we could either infer the arities function always returns [0, 2, 1], or it returns the arity of each constructor. While a human can spot the intention, there is a potential ambiguity. Using the second restriction, we conclude that this must represent the constant operation. To derive a version returning the arities we can write:

```
instance Arities (Sample α) where
  arities _ = [const 0 First{}
              , const 2 Second{}
              , const 1 Third{}]
```

While this code code is more verbose, any good optimiser (i.e. GHC) will generate identical code. We return to the issue of possible simplifications in §5.2.

While our DSL has forms of iteration (i.e. MapCtor), it does not have any conditional constructs such as **if** or **case**. The lack of conditionals is important for predictability – for every possible choice it would be necessary for the Sample type to choose all branches, thus increasing the size of Sample.

The restrictions in this section ensure that no fragment of output can be represented by both a constant and be parameterised by the data type. The Sample type ensures no fragment can be parameterised in multiple ways, by having different artiy/index values for some constructors – explaining why the Second constructor has arity 2, while the Third has arity 1. The restrictions in this section, along with the Sample data type, ensure predictability. We have checked the predictability property using QuickCheck (Claessen and Hughes 2000).

4. Implementing derive

This section covers the implementation of a derive function, as described in §2. There are many ways to write a derive function, our approach is merely one option – we hope that the scheme we have described provides ample opportunity for experimentation.

Before implementing derive it is useful to think about which properties are desirable. It is not necessary to guarantee correctness (see §2.1), but our method chooses to only generate correct results. We have shown that our DSL and sample input guarantee predictability without regard to the derive function, provided we meet the restrictions in §3.4, which we obey. We want our derive function to terminate, and ideally terminate within a reasonable time bound. Finally, we would like the derive function to find an answer if one exists, i.e.:

$\forall$ o $\in$ Output, d $\in$ DSL $\bullet$ null (derive o) $\Rightarrow$ apply d sample $\not\equiv$ o

We were unable to implement a derive function meeting this property for our problem which performed acceptably. Our method is a trade off between runtime and success rate, with a particular desire to succeed for real-world examples.

Our derive implementation is based around a parameterised guess. Each fragment of output is related to a guess – a DSL parameterised by some aspect of the environment. For example, OString "First" results in the guess CtorName parameterised by the first constructor. Concretely, our central Guess type is:

```
data Guess = Guess DSL
           | GuessCtr Int DSL   -- 0-based index
           | GuessFld Int DSL   -- 1-based index

derive :: Output → [DSL]
derive o = [d | Guess d ← guess o]

guess :: Output → [Guess]
```

Applying guess (OString "First") produces a guess of GuessCtr 0 CtorName. The GuessCtr and GuessFld guesses are paramterised by either constructors or fields, and can only occur within MapCtor or MapField respectively. The Guess guess is either parameterised by the entire data type, or is a constant which does not refer to the environment at all.

To generate a guess for the entire output, we start by generating guesses for each leaf node of the Output value, then work upwards combining them. If at any point we see an opportunity to apply one of our custom rules (i.e. Instance), we do so. The important considerations are how we create guesses for the leaves, how we combine guesses together, and where we apply our custom rules. We require that all generated guesses are correct, defined by:

$\forall$ o $\in$ Output $\bullet$ $\forall$ g $\in$ guess o $\bullet$ applyGuess g $\equiv$ o

```
applyGuess :: Guess → Output
applyGuess (Guess      d) = applyEnv d
  Env {envInput = sample}
applyGuess (GuessCtr i d) = applyEnv d
  Env {envInput = sample, envCtor = dataCtors sample !! i}
applyGuess (GuessFld i d) = applyEnv d
  Env {envInput = sample, envField = i}
```

4.1 Guessing Constant Leafs

4.1.1 String

To guess an OString value is simple – if it has a banned substring (i.e. Sample or one of the constructors) we generate an appropriately parameterised guess, otherwise we use the constant string. Some examples:

```
OString "hello" ≡ Guess (String "hello")
OString "Sample" ≡ Guess DataName
OString "First" ≡ GuessCtr 0 CtorName
OString "isThird" ≡ GuessCtr 2
  (Concat (List [String "is", CtorName]))
```

4.1.2 Application

The guess for an OApp is composed of two parts – the name of the constructor to apply and the list of arguments. The name of the constructor in App always exactly matches that in OApp. The arguments to App are created by applying guess to the list, and wrapping the generated DSL in App op. The guess for OApp can be written as:

```
guess (OApp op xs) = map (lift (App op)) (guess (OList xs))

lift :: (DSL → DSL) → Guess → Guess
lift f (Guess      d) = Guess      (f d)
lift f (GuessCtr i d) = GuessCtr i (f d)
lift f (GuessFld i d) = GuessFld i (f d)
```

4.1.3 Integer

Given an integer there may be several suitable guesses. An integer could be a constant, a constructor index or arity, or a field index. We can guess an OInt as follows:

```
guess (OInt i) =
  [GuessFld i FieldIndex | i ∈ [1, 2]] ++
  [GuessCtr 1 CtorIndex  | i ≡ 1] ++
  [GuessCtr 1 CtorArity  | i ≡ 2] ++
  [Guess (Int i)]
```

And some examples:

```
OInt 0 ≡ [Guess (Int 0)]
OInt 1 ≡ [GuessFld 1 FieldIndex, GuessCtr 1 CtorIndex
         , Guess (Int 1)]
OInt 2 ≡ [GuessFld 2 FieldIndex, GuessCtr 1 CtorArity
         , Guess (Int 2)]
OInt 3 ≡ [Guess (Int 3)]
```

When guessing an OInt, we never generate guesses for any constructors other than Second (represented by GuessCtr 1) – the reason is explained in §4.2.3.

4.2 Lists

Lists are the most complex values to guess. To guess a list requires a list of suitable guesses for each element, which can be collapsed into a single guess. Given a suitable collapse function we can write:

```
guess (OList xs) = mapMaybe
  (liftM fromLists ∘ collapse ∘ toLists) (mapM guess xs)

fromLists = lift Concat
toLists   = map (lift (λx → List [x]))

collapse :: [Guess] → Maybe Guess
```

The mapM function uses the list monad to generate all possible sequences of lists. The toLists function lifts each guess to a singleton list, and the fromLists function concatenates the results – allowing adjacent guesses to be collapsed without changing the result type. The function collapse applies the following three rules, returning a Just result if any possible sequence of rule applications reduces the list to a singleton element.

4.2.1 Promotion

The promotion rule adds a parameter to a guess. We can promote Guess to either GuessFld or GuessCtr, with any parameter value. The value Guess d, can be promoted to either of GuessCtr i d or GuessFld i d, for any index i. The promotion rule does not reduce the number of elements in the list, but allows other rules to apply, in particular the conjunction rule.

4.2.2 Conjunction

If two adjacent guesses have the same parameter value, they can be combined in to one guess. For example, given GuessCtr 2 d_1 and GuessCtr 2 d_2 we produce GuessCtr 2 (Concat (List [d_1, d_2])). This rule shows the importance of each guess evaluating to a list.

4.2.3 Sequence

The sequence rule introduces either MapField or MapCtor from a list of guesses. Given two adjacent guesses we can apply the rule:

```
(GuessFld 1 d₁) (GuessFld 2 d₂)
  | applyGuess (GuessFld 2 d₁) ≡ applyGuess (GuessFld 2 d₂)
  = GuessCtr 1 (MapField d₁)
```

It is important that the fields are in the correct order, one of the DSL values (in this case d_1) is applicable to both problems, and the resultant guess is paramterised by the Second constructor (which has two fields). We also permit sequences in reverse order, which we generate by reversing the list before, and inserting a Reverse afterwards.

The sequence construction for fields can be extended to constructors by demanding three guesses parameterised by consecutive constructors. For constructors we only check using the DSL relating to the Second constructor, as this DSL is the only one that could have a MapField construct within it. Because we only test against the Second DSL, we can avoid generating CtorArity and CtorIndex guesses for the other constructors. We also require that when creating a MapCtor the guess contains a CtorName, to ensure the restrictions from §3.4 are met.

4.3 Folds

The addition of fold to our DSL is practically motivated – a number of real derivations require it. Currently we only attempt to find folds in a few special cases. We require folds to start with one of the following patterns:

```
OApp m [OApp m [x, op, y], op, z]
OApp m [x, op, OApp m [y, op, z]]
```

Given such a pattern, we continue down the tree finding all matching patterns of op and m. After constructing a fold we then apply guess to the residual list.

4.4 Application

As with fold, the introduction of Application is practically motivated. We replace any sequence of left-nested OApp "App" expressions with Application.

4.5 Instance

As per the restrictions given in §3.4, the only way of creating an Instance value as output is to use the Instance DSL operator – it is forbidden to use App "Instance". Given this restriction, we translate values to Instance where they follow the pattern set out in §3.3.6.

5. Using Derived DSLs

This section discusses possible uses of a DSL after it has been derived. We start by showing how to simplify a DSL, then how to simplify the output produced by applying a DSL. Finally we give some alternative uses for a DSL, other than applying it to an input. We use the Arities type class from §3.4 as a recurring example.

5.1 DSL Simplification

We can replace a DSL with a simplified version provided the simplified version is congruent to the original. Using the apply function from Figure 3, we can determine a number of identities:

```
Concat (List (a ++ [List xs, List ys] ++ b)) ≡
  Concat (List (a ++ [List (xs ++ ys)] ++ b))
Concat (List (a ++ [List [ ]] ++ b)) ≡ Concat (List (a ++ b))
Concat (List [x]) ≡ x
Concat (List [ ]) ≡ List [ ]
```

To simplify a DSL we apply these identities from left to right wherever they occur, using the Uniplate generics library (Mitchell and Runciman 2007).

Unfortunately, even after simplifying the DSL, small examples still produce complex DSLs[3]. As an example, we give an abbreviated form of the Arities DSL – the full DSL is given in Appendix A. To simplify the presentation we have omitted some haskell-src-ext nodes (i.e. Ident, UnQual, SrcLoc), and added some syntactic sugar. We have written all DSL constructors in lower-case, and used upper-case for App constructors. After these translations, the Arities DSL is:

```
⟦instance [ ] "Arities" ⟦InsDecl (FunBind ⟦Match
  "arities"
  ⟦PWildCard⟧
  Nothing
  (List (mapCtor (application
    ⟦Var "const", Int ctorArity, RecConstr ctorName ⟦⟧⟧
  )))
  (BDecls ⟦⟧)⟧
)⟧⟧
```

5.2 Output Simplification

To obey the restrictions from §3.4 we require the addition of const applications in the Arities instance. While these const applications will be optimised away by a compiler, their removal simplifies the output for human readers. After apply, we translate the Output type to a haskell-src-exts type using the function fromOutput (§3.1). We then perform a number of simplifications, mainly simple constant folding, often using inbuilt knowledge of particular functions. Some of these simplifications can be applied by GHC (i.e. const), while others can't as they involve recursive functions (i.e. length).

We present the output simplification directly as we have implemented it, using the Uniplate generics library (Mitchell and Runciman 2007). All of the rules operate at the expression level, and each rule is correct individually. To easily express matches we introduce (≃), which converts complex expressions to strings before checking for equality.

```
simplify :: Biplate α Exp ⇒ α → α
simplify = transformBi f
```

[3] Hence the advantage of having these relationships derived, rather than writing them by hand.

```
where
  x ≃ y = prettyPrint x ≡ y

  f (App op (List xs))
      | op ≃ "length" = Lit $ Int $ fromIntegral $ length xs
      | op ≃ "head" = head xs
  f (InfixApp (Lit (Int i)) op (Lit (Int j)))
      | op ≃ "-" = Lit $ Int $ i − j
      | op ≃ ">" = Con $ UnQual $ Ident $ show $ i > j
  f (InfixApp x op y) | op ≃ "‘const‘" = x
  f (App (App con x) y) | con ≃ "const" = x
  f (Paren (Var x)) = Var x
  f (Paren (Lit x)) = Lit x
  f x = x
```

Some of these simplifications could be applied directly to the DSL, for example the removal of const. However, other simplifications can't be performed until after apply has been called, such as the reduction of (CtorArity $-$ 1). Currently we do not perform any of these simplifications directly to the DSL, only to the output.

5.3 DSL Usage

The obvious way to use a value of our DSL is to apply it to an input to generate an output, a haskell-src-exts AST. From an AST we can pretty-print it, and compile the resulting code. Alternatively we can use the haskell-src-meta library (Morrow 2009) to translate the output into Template Haskell, which can be integrated into GHC compiled programs.

5.3.1 Specialised Instance Generators

From a DSL we can generate a specialised instance generator, that takes an input and produces an output directly, without the interpretative step of the apply function. This construction corresponds to the first Futamura projection (Futamura 1999). For example with Arities, we could produce:

```
generateArities :: Input → [Decl]
generateArities input = [InstDecl srcLoc [ ]
  (UnQual $ Ident "Arities")
  [foldl TyApp
     (TyCon $ UnQual $ Ident $ dataName input)
     (map (TyVar ◦ Ident) vars)]
  [InsDecl (FunBind [Match srcLoc
     (Ident "arities") [PWildCard] Nothing
     ...
     (BDecls [ ])])]]
  where vars = take (dataVars input) $ map (:[ ]) [’a’ ..]
```

Since our apply and fromOutput functions are both terminating, the generateArities function can be constructed as:

```
generateArities = fromOutput ◦ apply aritiesDSL
```

The fromOutput and apply functions can then be unfolded and reduced until aritiesDSL has disappeared.

The first version of DERIVE generated a string corresponding to the source code of a specialised instance generator – primarily because it lacked a complete representation of the DSL. For the new version of DERIVE we do not create specialised instance generators – the only benefit would be the removal of interpretive overhead, which we believe to be negligible.

5.3.2 Dynamic Instance Generators

In Haskell each instance is defined by some fragment of source code, and new instances cannot be constructed at runtime. However, using Haskell's reflection capabilities (Lämmel and Peyton Jones 2004), one instance can define implementations for many data types. For example, all algebraic data types can be given an Arities instance with:

```
instance Data d_type ⇒ Arities d_type where
  arities _ =
      [ const (d_ctorArity d_ctor) (d_ctorNull d_ctor :: d_type)
      | d_ctor ← d_dataCtors (undefined :: d_type)]
```

This instance declaration was generated automatically from the Arities DSL. The instance requires that the type support the Data class, allowing type information to be queried at runtime. The expression makes use of a number of library functions defined by DERIVE, namely d_dataCtors, d_ctorArity and d_ctorNull – all defined in terms of operations within the Data class.

To use a dynamic instance generator it is necessary to enable some Haskell extensions. The first is ScopedTypeVariables, which allows the d_type variable to be bound in the instance declaration head and used within instance member functions. The second extension is to allow unrestricted overlapping instances, so that custom Arities declarations can be provided for basic types. Finally, it is necessary to have Data and Typeable instances for each type of interest – these can be derived automatically using either the DERIVE tool[4] or the extension DeriveDataTypeable.

Currently the creation of dynamic instances is limited to a small number of examples, but we believe many more instances could be dealt with. However, there are some instances which cannot be produced dynamically. We have identified two cases so far:

1. DSLs which generate name bindings using information from the data type, such as the name of the constructor (i.e. isFirst), cannot be constructed.
2. If an instance makes use of a particular type class on fields, but that class does not have an instance for all types implementing Data, then the instance will not type check.

The use of dynamic instances removes the inconvenience of a separate preprocessor, but only works on a restricted set of instances. Dynamic instances increase runtime, due to the overhead of reflection and the reduction of optimisation opportunities. Previously, only a handful of classes have provided dynamic instances – the only one we are aware of is the Binary class. One reason for not providing dynamic instances is that they are complex to write – use of the SYB libraries requires an intricate combination of type-level and value-level programming. Using DERIVE many type classes could have dynamic instances created with ease, by first deriving an instance from one example and then translating the DSL.

6. Results

This section discusses the results of using our automatic derivation scheme on real examples. We first categorise the instances we are unable to derive, then share some of the tricks we have developed to succeed with more examples. For each limitation we discuss possible modifications to our system to overcome it. Finally we give timing measurements for our implementation.

6.1 Limitations of Automatic Derivation

The instance generation scheme given is not complete – there exist instances whose generator cannot be determined. The DERIVE tool (Mitchell and O'Rear 2007) generates instances for user defined data types. Of the 24 instances supported by DERIVE, 15 are derived from one example, while 9 require hand-written instance

[4] While the Data instance can be derived from a single example, alas the Typeable instance cannot (see §6.1.1), but it is still available within DERIVE.

generators. All the examples which can't be derived are due to the choices of abstraction in our Input type. We now discuss each of the pieces of information lacking from Input that result in some instances being inexpressible.

6.1.1 Module Names

Some type classes require information about the module containing a type, for example Typeable instances (Lämmel and Peyton Jones 2003) follow the pattern:

```
typename_Language = mkTyCon "ModuleName.Language"

instance Typeable Language where
  typeOf _ = mkTyConApp typename_Language []
```

The Typeable class performs runtime type comparison, so each distinct type needs a distinct string to compare, and the module name is used to disambiguate. Our Input type does not include the module name, so cannot be used to derive Typeable. It would be possible to define the string "Module.Name" as the module name of the sample, and treat it in a similar manner to the string "Sample". However, the only instance we are aware of that requires the module name is Typeable, so we do not provide module information.

6.1.2 Infix Constructors

Some instances treat infix constructors differently, for example the Show instance on a prefix constructor is:

```
instance Show PrefixConstructor where
  show (Prefix x y) = "Foo " ++ show x ++ " " ++ show y
```

But using an infix constructor:

```
instance Show InfixConstructor where
  show (x :+: y) = show x ++ " :+: " ++ show y
```

Our Input type does not express whether a constructor is infix or prefix, so cannot choose the appropriate behaviour. The loss of infix information mainly effects instances which display information to the user, i.e. Show and pretty printing (Hughes 1995). For most type classes, the infix information is not used, and infix constructors can be bracketed and treated as prefix (:+:). To deal with infix constructors would require an infix constructor added to the Sample data type, and modifications to the DSL to allow different results to be generated depending on infix information. These changes would pose difficulties to predictability and require all example instances to have at least one additional case defined – we do not consider this a worthwhile trade off for a small number of additional instances.

6.1.3 Record-based definitions

Haskell provides records, which allow some fields to be labelled. Some operations make use of the record fields within a data type, for example using the data type:

```
data Computer = Desktop {memory :: Int}
              | Laptop  {memory :: Int, weight :: Int}
```

It is easy to write the definition:

```
hasWeight Desktop{} = False
hasWeight Laptop{}  = True
```

Where hasWeight returns True if the weight selector is valid for that constructor, and False if weight x $\equiv \bot$. Unfortunately our Input type does not contain information about records, so cannot express this definition. There are only a few type classes which exhibit label specific behaviour, such as Show which outputs the field name if present.

Record fields are not present in our Sample type, but could be added. The difficulty is that Haskell allows for one field name to be shared by multiple constructors, and allows some constructors to have field names while others do not. This flexibility results in a massive number of possible combinations, and so a Sample type with sufficient generality would require many constructors. Allowing records would be more feasible for a language such as F#, where records contain only one constructor and all fields must be named.

6.1.4 Type-based definitions

Our Sample data type has a simple type structure, and our DSL does not allow decisions to be made on the basis of type – these restrictions means some type classes can't be defined. For example, a Monoid instance processes items of the same type using mappend, but items of a different type using mempty. Several other type classes require type specific behaviour, including Functor, Traversable and Uniplate.

The lack type information has other consequences. For example, we can write the definition:

```
fromFirst  (First        ) = const First{}  $ tuple0
fromSecond (Second x₁ x₂) = const Second{} $ tuple2 x₁ x₂
fromThird  (Third  x₁    ) = const Third{}  $ tuple1 x₁
```

This function returns the elements contained within a constructor, generalising operations such fromJust, and has seen extensive use in the Yhc compiler (The Yhc Team 2007). When compiled with GHC this code generates a warning that no top-level type signatures have been given. These type signatures can be inferred, but the Haddock documentation tool (Marlow 2002) won't include functions lacking type signatures. Without type information in Input, we can't generate appropriate type signatures.

We see no easy way to include type information in our derivation scheme – types have too much variety, and different type classes make use of different type information. It may be possible to identify some restricted type information that could be used for a subset of type-based instances, but we have not done so.

6.2 Practical Experiences

This section describes our experiences of specifying instances in a form suitable for derivation. Ideally, we would write all instances in a natural way, but sometimes we need to make concessions to our derivation algorithm. Using the techniques given here, it seems possible to write most instances which are based on information included in the Input type.

6.2.1 Brackets Matter

The original DERIVE program used Template Haskell, which include brackets in the abstract syntax tree. For example, the expressions (First) and First are considered equal. However, using haskell-src-exts, brackets are explicit and care must be taken to ensure every constructor has the same level of bracketing. Examples of otherwise unnecessary brackets can be seen in §6.1.4, where the constructor First is bracketed. Currently some redundant brackets are removed by the transformations described in §5.2.

6.2.2 Variable Naming

When naming variables it is important that a sequence of variables follow a pattern. For example, in §6.1.4 we use Second x_1 x_2, rather than Second x y. By naming variables with consecutive numbers we are able to derive the fields correctly.

6.2.3 Explicit Fold Base-Case

When performing a fold, it is important to explicitly include the base-case. In the introductory example of NFData the Second alternative is specified as rnf x_1 \`seq\` rnf x_2 \`seq\` (), however we can show that:

$\forall$ x $\bullet$ rnf x \`seq\` () $\equiv$ rnf x

Therefore we could write the Second alternative more compactly as rnf x_1 \`seq\` rnf x_2. However, doing so would mean there was not one consistent pattern suitable for all constructors, and the derivation would fail. In general, when considering folds, the base-case should always be written explicitly.

6.2.4 Empty Record Construction

One useful feature of Haskell records is the empty record construction. The expression Second{} creates the value Second $\perp$ $\perp$. This expression is useful for generating constructors to pass as the second argument to const[5], for some generic programming operations, and for values that are lazily evaluated. The pattern Second{} matches all Second constructors, regardless of their fields.

6.2.5 Constructor Count

Some instances aren't inductive – for example Binary instances require a tag indicating which constructor has been stored, but only if there is more than one constructor. This pattern can be written as:

```
instance Binary α ⇒ Binary (Sample α) where
  put x = case x of
    First        → do putTag 0
    Second x1 x2 → do putTag 1; put x1; put x2
    Third  x1    → do putTag 2; put x1
    where
      useTag = length [First{}, Second{}, Third{}] > 1
      putTag = when useTag ∘ putWord8

  get = do
    i ← getTag
    case i of
      0 → do return (First)
      1 → do x1 ← get; x2 ← get; return (Second x1 x2)
      2 → do x1 ← get; return (Third x1)
      _ → error "Corrupted binary data for Sample"
    where
      useTag = length [First{}, Second{}, Third{}] > 1
      getTag = if useTag then getWord8 else return 0
```

The value length [First{}, Second{}, Third{}] is used to compute the number of constructors in the data type, which can be tested to get the correct behaviour. This pattern is used in other classes, for example Enum and Arbitrary. Using the simplifications from §5.2 we can remove the test and produce code specialised to the number of constructors.

The pattern for the number of constructors is useful, but seems a little verbose. In the first version of DERIVE the constructor count was guessed from the number 3. Unfortunately, the inclusion of this guess breaks the restrictions we have imposed for predictability. Another way of simplifying this pattern would be to introduce a meta function ctorCount, which expanded to the number of constructors. This solution would mean inputs were not real example instances, and would require users to learn part of the DSL – something we have tried to avoid. In the end, we simply accept that the constructor count is slightly verbose.

6.3 Timing Properties

We have implemented the methods described in this paper, and have used them to guess all 15 examples referred to in §6.1, along with 2 additional test cases. For each example we perform the following steps:

1. We derive the DSL from an example.
2. We apply the DSL (without output optimisation) to the Sample data type and check it matches the input example.
3. We apply the DSL to three other data types, namely lists, the eight element tuple and the expression type from the Yhc Core library (Golubovsky et al. 2007).

To perform all steps for 17 examples takes 0.3 seconds when compiled with GHC -O0 on a laptop with a 2GHz CPU and 1Gb of RAM. We consider these times to be more than adequate, so have not carried out further experiments or investigated additional optimisations.

7. Related Work

An earlier version of the DERIVE tool was presented in a previous paper (Mitchell 2007). The previous work described only the derivation algorithm. There was no intermediate DSL, and no predictability. Given a single example the tool could produce multiple different answers, and would always use the first generated – not always corresponding to the users intention. This paper presents a much more general scheme, along with many improvements to the previous work. Some of the areas of future work in the previous paper have been tackled, such as dynamic instances (see §5.3.2). Crucial improvements have been made to the derivation algorithm, particularly when dealing with lists.

We are unaware of any work (other than our own) that attempts to automatically derive Haskell type classes. Therefore we split the remaining related work in to two sections – that which explicitly defines instance relationships, and that which tries to derive relationships.

7.1 Specifying Type Classes

From an end-user perspective, the DrIFT tool (Winstanley 1997) is similar to DERIVE – both take data types and produce associated instances. To add a type-class to DrIFT the programmer manually writes a translation from input types to Haskell source code, using pretty-printing combinators. There is no automatic derivation of instance generators, and no underlying DSL. As a result, it is substantially easier to add generators which can be derived from one example to DERIVE.

Another mechanism for specifying type classes is to use generic type classes (Hinze and Peyton Jones 2000), a language extension supported by GHC. A programmer can write default instances for type classes in terms of the structure of a type, using unit, products and sums. There are many restrictions on such classes, including restrictions on the type of instance methods and the structure of the input type. Using the abstraction of products and sums, it is impossible to represent many instances such as those dealing with records or containing type specific behaviour.

7.2 Deriving Relationships

The purpose of our work is to find a pattern, which is generalised to other situations. Genetic algorithms (Goldberg 1989) are often used to automatically find patterns in a data set. Genetic algorithms work by evolving a hypothesis (a gene sequence) which is tested against a sample problem. While genetic algorithms are good for search, they usually use a heuristic to measure closeness – so lack the exactness of our approach.

There is much research on learning relationships from a collection of input/output pairs, often using only positive examples (Kitzelmann 2007). Some work tackles this problem using exhaus-

[5] "Second" would also work, but the use of a string feels too unpleasant.

tive search (Katayama 2008), a technique that could possibly replace our derive function. Instead of using specific examples, some work generalises a set of non-recursive equations into a recursive form (Kitzelmann and Schmid 2006; Kitzelmann 2008). All these pieces of work require a set of input/output examples, in contrast to our method that requires only one output for a specific input.

The closest work we are aware of is that of the theorem proving community. Induction is a very common tactic for writing proofs, and well supported in systems such as HOL Light (Harrison 1996). Typically the user must suggest the use of induction, which the system checks for validity. Automatic inference of an induction argument has been tried (Mintchev 1994), but is rarely successful. However, these systems all work from one positive example, attempting to determine a reasonably restricted pattern.

8. Conclusions and Future Work

We have presented a scheme for deriving a DSL from one example, which we have used to automatically derive instance generators for Haskell type classes. Our technique has been implemented in the DERIVE tool, where 60% of instance generators are specified by example. The ease of creating new instances has enabled several users to contribute instance generators. The DERIVE tool can be downloaded from Hackage[6], and we encourage interested users to try it out.

One of the key strengths of our derivation scheme is that concerns of correctness and predictability are separated from the main derivation function. Correctness is easy to test for, so incorrect derivations can simply be discarded. Predictability is a property of the DSL and sample input, and can be determined in isolation from the derivation function. The derivation function merely needs to take a best guess at what derivation might work, allowing greater freedom to experiment.

We see several lines of future work:

- By deriving an explicit DSL, we can reuse the DSL for other purposes. We have already shown the creation of dynamic instances in §5.3.2, but there are other possible uses. A DSL could be used to prove properties, for example that all Eq instances are reflexive, or that put/get in Binary are inverses. Another use might be to generate human readable documentation of an instance. We suspect there are many other uses.
- The Sample data type (Figure 1) allows many instances to be inferred – but more would be desirable. We have discussed possible extensions in §6.1, but none seems to offer compelling benefits. An alternative approach would be to introduce new sample data types with features specifically for certain types of definition. Care would have to be taken that these definitions still preserved predictability, and did not substantially increase the complexity of writing examples.
- While our scheme is implemented in a typed language, most of the actual DSL operations work upon a universal data type with runtime type checking – essentially a dynamically typed language. In order to preserve types throughout we could make use of GADTs (Peyton Jones et al. 2006).
- We have implemented our scheme specifically for instance generators in Haskell, but the same scheme could be applied to other computer languages and other situations. One possible target would be F#, where there are interfaces instead of type classes. Another target could be an object-orientated language, where design patterns (Gamma et al. 1995) are popular.

[6] http://hackage.haskell.org/package/derive

Computers are ideally suited to applying a relationship using new parameters, but specifying these relationships can be complex and error prone. By specifying a single example, instead of the relationship, a user can focus on what they care about, rather than the mechanism by which it is implemented.

Acknowledgements

Thanks to Stefan O'Rear for help writing the first version of the DERIVE tool. Thanks to Niklas Broberg for the excellent haskell-src-exts library. Thanks to Hongseok Yang for fruitful discussions on the original instance generation work. Thanks to Mike Dodds for constructive criticism on earlier drafts.

A. Arities DSL

This section presents the full Arities DSL, a simplified version of which is shown in §5.1.

```
List [Instance [] "Arities" (List [App "InsDecl" (
  List [App "FunBind" (List [List [
    App "Match" (List
      [App "Ident" (List [String "arities"])
      , List [App "PWildCard" (List [])]
      , App "Nothing" (List [])
      , App "UnGuardedRhs" (List [App "List" (List [
        MapCtor (Application (List
          [App "Var" (List [App "UnQual" (List [
            App "Ident" (List [String "const"])])])
          , App "Lit" (List [App "Int" (List [CtorArity])])
          , App "RecConstr" (List [App "UnQual" (List [
            App "Ident" (List [CtorName])]), List []])]
        ))
      ])])
      , App "BDecls" (List [List []])]
    )
  ]])]
)])]
```

References

Niklas Broberg. haskell-src-exts. http://www.cs.chalmers.se/~d00nibro/haskell-src-exts/, 2009.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. ICFP '00*, pages 268–279. ACM Press, 2000.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *Proc. TCS '02*, pages 448–460, Deventer, The Netherlands, 2002.

Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core – from Haskell to Core. *The Monad.Reader*, 1(7): 45–61, April 2007.

John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proc. Formal Methods in*

Computer-Aided Design, FMCAD'96, volume 1166 of *LNCS*, pages 265–269. Spinger-Verlag, 1996.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proc Haskell Workshop 2000*. Elsevier Science, September 2000.

John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

Susumu Katayama. Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, volume 5351, pages 199–210. LNCS, 2008.

Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *Proceedings of the Workshop on Approaches and Applications of Inductive Progamming (AAIP'07)*, pages 15–26, 2007.

Emanuel Kitzelmann. Data-driven induction of functional programs. In *Proc. ECAI 2008*. IOS Press, July 2008.

Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs – An explanation based generalization approach. *Journal of Machine Learning Research*, 7(Feb):429–454, 2006.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. TLDI '03*, pages 26–37. ACM Press, March 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proc. ICFP '04*, pages 244–255. ACM Press, 2004.

Simon Marlow. Haddock, a Haskell documentation tool. In *Proc. Haskell Workshop 2002*, Pittsburgh Pennsylvania, USA, October 2002. ACM Press.

Sava Mintchev. Mechanized reasoning about functional programs. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming*, pages 151–166. Springer, Berlin, Heidelberg, 1994.

Neil Mitchell. Deriving generic functions by example. In Jan Tobias Mühlberg and Juan Ignacio Perna, editors, *Proc. York Doctoral Symposium 2007*, pages 55–62. Tech. Report YCS-2007-421, University of York, October 2007.

Neil Mitchell and Stefan O'Rear. Derive - project home page. `http://community.haskell.org/~ndm/derive/`, March 2007.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proc. Haskell '07*, pages 49–60. ACM, 2007.

Matt Morrow. haskell-src-meta. `http://hackage.haskell.org/package/haskell-src-meta`, 2009.

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Haskell Workshop '02*, pages 1–16. ACM Press, 2002.

The GHC Team. The GHC compiler, version 6.10.3. `http://www.haskell.org/ghc/`, May 2009.

The Yhc Team. The York Haskell Compiler – user manual. `http://www.haskell.org/haskellwiki/Yhc`, February 2007.

Philip Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, January 1998.

Philip Wadler. How to replace failure by a list of successes. In *Proc. FPCA '85*, pages 113–128. Springer-Verlag New York, Inc., 1985.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.

Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*. BCS Workshops in Computer Science, September 1997.

Synthesis of Functions Using Generic Programming

Pieter Koopman and Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract

Inductive programming aims to synthesize functions or programs from a small number of input-output pairs. In general there will by many functions that have the desired behavior. From this family of solutions we are interested in the smallest or simplest function. In some situations there are (often well- know) algorithms to construct such functions, for instance for fitting a linear function through a set of points in the R2. In general it is very hard to construct functions for arbitrary data types in this way.

Instead of constructing a function that has the desired behavior we use a generate-and-test based approach. Our system generates a sequence of more and more complex candidate functions, the system verifies if these candidates have the desired behavior and yields the first candidate that passes this test. Since there are enormous many candidate functions one has to guide this search procedure in one way or another to synthesize the desired function in reasonable time.

In this paper we show how we can control the synthesis of candidate functions effectively by defining a tailor made data type for the grammar of the candidate functions. Instances of this data type are the abstract syntax trees of the candidate functions. The instances of these data types representing the candidate functions are generated by a generic algorithm. Instances of this synthesis algorithm for specific data types can be derived automatically by the compiler of the host language (the functional programming language Clean). It appears that the generic algorithm for generating instances of a data type that is used to generate test suites in the model-based test tool Gast is very effective to synthesize candidate functions in inductive programming.

In order to verify if a synthesized abstract syntax tree represents the correct function, the system needs to be able to execute it as a function. This is done by a user defined instance of a type class that transforms the abstract syntax tree to the corresponding function. These instances are always very simple. For a new application domain the user has to define the grammar of candidate functions as a data type and how instances of this data type are transformed to functions. The system synthesizes the instances and tests the candidates until one (or more) functions with the desired behavior are found. This approach has been proven to be effective in generating small functional programs and lambda- expressions.

Inductive Programming

A Survey of Program Synthesis Techniques

Emanuel Kitzelmann
Faculty of Information Systems and Applied Computer Science, University of Bamberg
emanuel.kitzelmann@uni-bamberg.de

Abstract

Inductive programming—the use of inductive reasoning methods for programming, algorithm design, and software development—is a currently emerging research field. A major subfield is inductive program synthesis, the (semi-)automatic construction of programs from exemplary behavior. Inductive program synthesis is not a unified research field until today but scattered over several different established research fields such as machine learning, inductive logic programming, genetic programming, and functional programming. This impedes an exchange of theory and techniques and, as a consequence, a progress of inductive programming. In this paper we survey theoretical results and methods of inductive program synthesis that have been developed in different research fields until today.

1. Introduction

Inductive programming (IP) is an emerging field, comprising research on inductive reasoning theory and methods for computer programming, algorithm design, and software development. In this sense, albeit with different accentuation, the term has been used by Partridge [30], by Flener and Partridge [7], within the workshops on "Approaches and Applications of Inductive Programming", and within the ICML'06 tutorial on "Automatic Inductive Programming".

IP has intersections with machine learning, artificial intelligence, programming, software engineering, and algorithms research. Nevertheless, it goes beyond each of these fields in one or the other aspect and therefore is a research field in its own right, intrinsically.

It goes beyond classical machine learning in that the focus lies on learning general programs including loops and recursion, instead of merely (mostly non-recursive) models or classifiers in restricted representational frameworks, such as decision trees or neural networks.

In classical software engineering and algorithm design, a *deductive*—reasoning from the general to the specific—view of software development is predominant. One aspires a general problem description as starting point from which a program or algorithm is developed as a particular solution. Methods based on deductive reasoning exist to partly automatize the programming and verification process—such as automatic code generation from UML diagrams, (deductive) program synthesis to generate algorithmic parts, program transformation and refactoring to optimize programs, and theorem proving, model checking, and static analysis to verify programs. To emphasize this common deductive foundation one might speak of *deductive programming* to subsume established software development methods.

Inductive programming, on the other side, aims at developing methods based on *inductive*—from the specific to the general—reasoning (not to be confused with mathematical or structural induction) to assist in programming, algorithm design, and the development of software. Starting point for IP methods is specific data of a problem—use cases, test cases, desirable (and undesirable) behavior of a software, input/output examples (I/O-examples) of a function or a module interface, computation traces of a program for particular inputs and so forth. Such descriptions of a problem are known to be incomplete. Inductive methods produce a *generalization* of such an incomplete specification by identifying general patterns in the data. The result might be again a—more complete—specification or an actual implementation of a function, a module, or (other parts of) a program.

Inductive reasoning is per se unsound. Inductively obtained conclusions are *hypotheses* and incapable of proof regarding their premises. This is, perhaps, the most severe objection against IP. What is the use of methods whose results cannot be proven correct and possibly deviate from what was intended? However, if the data at hand is representative then it is likely that identified patterns actually hold in the general case and that, indeed, the induced result meets the general problem. On the other side, *all* software development necessarily makes a transition from a first *informal* and often incomplete problem description by the user or customer to a complete and ideally formal specification. This transition is (i) also incapable of formal proof and (ii) possibly based on—non-systematic, inexplicit—generalization. Also, IP should not be understood as a replacement for deductive methods but as an addition. IP may be used in different ways: to generate candidate solutions subject to further inspection, in combination with deductive methods to tackle a problem from the general description *as well as* from concrete (counter-)instances, to systematize occurring generalizations, or to check the representativeness of example cases provided by the user. Some problems, especially many problems in the field of artificial intelligence, elude a complete specification at all, e.g., face recognition. This factum is known as *knowledge-acquisition bottleneck*. Overall, there is no reason why systematically incorporating existing or easily formulated data by inductive methods should not improve efficiency and even validity of software development.

One important aspect of IP is the inductive synthesis of actual, executable *programs* including recursion or loops. Except to professional software development, possible application fields of the (semi-)automatic induction of programs from exemplary behavior are end-user programming and learning of recursive policies [34] in intelligent agents. Research on *inductive program synthesis (IPS)* started in the seventies. However, it has, since then, always been only a niche in several different research fields and communities such as artificial intelligence, machine learning, inductive logic programming (ILP), genetic programming, and functional programming. Until today, there is no uniform body of theory and methods. This fragmentation over different communities impedes exchange of results and may lead to redundancies. The problem is all the more profound as only few people and groups at all are working on IPS worldwide.

This paper surveys theoretical results and IPS methods that have been developed in different research fields until today. We grouped the work into three blocks: First the classical, analytic data-driven induction of LISP programs as invented by Summers [37] and its generalizations (Sec. 3), second ILP (Sec. 4), and third several generate-and-test based approaches to the induction of functional programs (Sec. 5). In Sec. 6 we state some conclusions and ideas of further research. As general preliminaries, we informally introduce some common IPS concepts in the following section.

This survey is quite comprehensive, yet not complete and covers functional generate-and-test methods less detailed than the other two areas. This is due to limited space in combination with the author's areas of expertise and shall not be interpreted as a measure of quality. We hope that it will be a useful resource for all people interested in IP.

2. Basic Inductive Programming Concepts

IPS aims at constructing a computer program or algorithm from a *(known-to-be-)incomplete specification* of a function to be implemented, called *target function*. Incomplete means, that the target function is not specified on its whole domain but only on (small) parts of it. Typically, an incomplete specification consists of a subset of the graph of the function: *input/output examples (I/O-examples)*. Variables may be allowed in I/O-examples and also more expressive formalisms have been used to specify the target function.

An induced program contains function *primitives*, predefined functions known to the IPS system. Primitives may be fixed within the IPS system or dynamically be given as an extra, problem-specific, input. Dynamically provided primitives are called *background knowledge*.

Example 1. Suppose the following I/O-examples on lists (whatever the list elements $A, x, y, z, 1, 2, 3, 5$ stand for; constants, variables, or compound objects), are provided: $(A) \mapsto (\,)$, $(x, y, z) \mapsto (x, y)$, $(3, 5, 2, 1) \mapsto (3, 5, 2)$. Given the common list constructors/destructors `nil`, `cons`, `head`, `tail`, the predicate `empty` to test for the empty list, and the `if-then-else`-conditional as primitives, an IPS system might return the following implementation of the *Init*-function returning the input list without its last element:

```
F(x) = if empty(tail(x)) then nil
         else cons(head(x),F(tail(x))) .
```

Given a particular set of primitives, some target function may not be representable by only one recursive function definition such that a non-specified recursive *subfunction* needs to be introduced; this is called *(necessary) predicate invention* in ILP. E.g., it is not possible to define the *Reverse* function by one recursive function definition of one parameter only using the primitives from the example above.

IPS is commonly regarded as a *search problem*. In general, the problem space consists of the representable programs as nodes and instances of the operators of the IPS system to transform one program into another as arcs. Due to *underspecification* in IP, typically infinitely many (semantically) different programs meet the specification. Hence, one needs criteria to choose between them. Such criteria are called *inductive bias* [23]. Two kinds of inductive bias exist: If an IPS system can only generate a certain proper subset of all (computable) functions of some domain, either because its language is restricted or because its operators are not able to reach each program, this constitutes a *restriction bias*. The order in which the problem space is explored and hence the ordering of solutions is the *preference bias*; it can be modelled as probability distribution over the program space.

3. The Analytical Functional Approach

A first systematic attempt to IPS was made by Summers [37]. He noticed that under particular restrictions regarding allowed primitives, program schema, and choice of I/O-examples, a recursive LISP program can be computed from I/O-examples without search in program space. His insights originated some further research.

3.1 Summers' Pioneering Work

Summers' approach to induce recursive LISP functions from I/O-examples includes two steps: First, a so-called *program fragment*, an expression of one variable and the allowed primitives, is derived for each I/O-pair such that applied to the input, evaluates to the specified output. Furthermore, predicates are derived to distinguish between example inputs. Integrated into a McCarthy conditional, these predicate/fragment pairs build a non-recursive program computing the I/O-examples and is considered as a first approximation to the target function. In a second step, recurrent relations between predicates and fragments each are identified and a recursive program generalizing them is derived.

Example inputs and outputs are *S-expressions*, the fundamental data structure of the LISP language [22]. We define the set of *subexpressions* of an S-expression to consist of the S-expression itself and, if it is non-atomic, of all subexpressions of both its components.

The programs constructed by Summers' technique use the LISP primitives $cons$, car, cdr, nil, $atom$, and T, the last denoting the truth value $true$. Particularly, no other predicates than $atom$ and T (e.g., eq for testing equality of S-expressions), and no atoms except for nil are used. This choice of primitives is not arbitrary but crucial for Summers' methodology of deriving programs from examples without search. The McCarthy conditional and recursion are used as control structure. Allowing $atom$ and T as only predicates and nil as only atom in outputs means that the atoms in the I/O-examples, except for nil, are actually considered as *variables*. Renaming them does not change the meaning. This implies that any semantic information must be expressed by the *structure* of the S-expression.

3.1.1 1. Step: Initial Non-recursive Approximation

Given a set of k I/O-examples, $\{\langle i_1, o_1\rangle, \ldots, \langle i_k, o_k\rangle\}$, a program fragment $f_j(x), j \in \{1, \ldots, k\}$, composed of $cons$, car, and cdr is derived for each I/O-pair, which evaluates to the output when applied to the input: $f_j(i_j) = o_j$.

S-expressions are uniquely constructed by $cons$ and destructed by car and cdr. We call car-cdr compositions *basic functions* (cp. [36]). Together with the following two conditions, this allows for determining unique program fragments. (i) Each atom may occur only once in each input. (ii) Each atom, except for nil, occurring in an output must also occur in the corresponding input. Due to the first condition, each subexpression occurs exactly once in an S-expression such that subexpressions are denoted by unique basic functions.

Deriving a program fragment works as follows: All subexpressions of an input, together with their unique basic functions, are enumerated. Then the output is rewritten by composing the basic functions from the input subexpressions with $cons$ and nil.

Example 2. Consider the I/O-pair $((a\ .\ b)\ .\ (c\ .\ d)) \mapsto ((d\ .\ c)\ .\ (a\ .\ b))$. The input contains the following subexpressions, paired with the corresponding unique basic functions:

$$\langle((a\ .\ b)\ .\ (c\ .\ d)), I\rangle\ ,\quad \langle(a\ .\ b), car\rangle\ ,\quad \langle(c\ .\ d), cdr\rangle\ ,$$
$$\langle a, caar\rangle\ ,\quad \langle b, cdar\rangle\ ,\quad \langle c, cadr\rangle\ ,\quad \langle d, cddr\rangle\ .$$

Since the example output is neither a subexpression of the input nor nil, the program fragment becomes a $cons$ of the fragments for the car- and the cdr-component, respectively, of the output. The car-part, $(d\ .\ c)$, again becomes a $cons$, namely of the basic functions for d: $cddr$, and c: $cadr$. The cdr-part, $(a\ .\ b)$, *is* a subexpression of the input, its basic function is car. With variable x denoting the input, the fragment for this I/O-example is thus:

$$cons(cons(cddr(x), cadr(x)), car(x))$$

Next, predicates $p_j(x), j = 1, \ldots, k$ must be determined. In order to get the correct program fragment f_j be evaluated for each input i_j, all predicates $p_{j'}(i_j), 1 \leq j' < j$ (positioned before p_j in the conditional) must evaluate to $false$ and $p_j(i_j)$ to $true$. Predicates fulfilling this condition exist if the example inputs form a chain.

We do not describe the algorithm here. Both algorithms, for computing fragments and predicates, can be found in [36]. Fig. 1 shows an example for the first step.

3.1.2 2. Step: Recurrence Relations

The basic idea in Summers' generalization method is this: The fragments are assumed to be the actual computations carried out by a *recursive* program for the intended function. Hence fragments of greater inputs must comprise fragments of lesser inputs as subterms, with a suitable substitution of the variable x and in a recurrent form along the set of fragments. The same holds analogously for the predicates. Summers calls this relation between fragments and predicates *differences*.

As a preliminary for the following, we need to define the concept of a *context*. A *(one-hole) context* $C[\,]$ is a term including exactly one occurrence of the distinguished symbol $\square$. $C[s]$ denotes the result of replacing the $\square$ by the (sub)term s in $C[\,]$.

Definition 1. A *difference* exists between two terms (fragments or predicates) t, t' iff $t' = C[t\sigma]$ for some context $C[\,]$ and substitution σ.

If we have $k+1$ I/O-examples, we only consider the first k fragment/predicate pairs because the last predicate is always 'T', such that no sensible difference can be derived for it.

Example 3. The following differences, written as recurrence relations ($2 \leq i \leq 3$), can be identified in the first $k = 4$ fragments/predicates of the program of Fig. 1.

$$\begin{aligned} p_1(x) &= atom(cdr(x)) & f_1(x) &= nil \\ p_2(x) &= atom(cddr(x)) & f_2(x) &= cons(car(x), nil) \\ p_{i+1}(x) &= p_i(cdr(x)) & f_{i+1}(x) &= cons(car(x), f_i(cdr(x))) \end{aligned}$$

In the general case, we have (for k fragments/predicates):

$j-1$ "constant" fragments (as derived from the examples):

$f_1, \ldots, f_{j-1}$,

further n constant base cases: $f_j, \ldots, f_{j+n-1}$,

finally, remaining $k-(j+n-1)$ cases recurring to (1)

previous cases: $f_{i+n} = C[f_i\sigma_1]$ for $i = j, \ldots, k-n$;

analogously for predicates:

$p_1,\ \ldots,\ p_{j-1},\ p_j,\ \ldots,\ p_{j+n-1},\ p_{i+n} = p_i(\sigma_2)$.

$$\begin{array}{rl}
(a) \mapsto nil, & F(x) = (atom(cdr(x)) \to nil \\
(a,b) \mapsto (a), & \quad atom(cddr(x)) \to cons(car(x), nil) \\
(a,b,c) \mapsto (a,b), & \quad atom(cdddr(x)) \to cons(car(x), cons(cadr(x), nil)) \\
(a,b,c,d) \mapsto (a,b,c), & \quad atom(cddddr(x)) \to cons(car(x), cons(cadr(x), cons(caddr(x), nil))) \\
(a,b,c,d,e) \mapsto (a,b,c,d)\,. & \quad T \to cons(car(x), cons(cadr(x), cons(caddr(x), cons(cadddr(x), nil)))))
\end{array}$$

Figure 1. I/O-examples (left) and the corresponding first approximation (right).

Index j denotes the first predicate/fragment pair which recurs in some following predicate/fragment pair (the first base case). The precedent $j-1$ predicate/fragment pairs do not recur. n is the interval of the recurrence. For Example 3 we have $j = 2$ and $n = 1$.

Inductive Inference. If $k - j \geq 2n$ then we *inductively infer* that the recurrence relations hold for all $i \geq j$.

In Example 3 we have $k - j = 2 \geq 2 = 2n$ and hence induce that the relations hold for all $i \geq 2$.

The generalized recurrence relations may be used to compute new approximations of the assumed target function. The mth *approximating function*, $m \geq j$, is defined as

$$F_m(x) = (p_1(x) \to f_1(x), \ldots, p_m(x) \to f_m(x), T \to \omega)$$

where the p_i, f_i with $j < i \leq m$ are defined in terms of the generalized recurrence relations and where ω means *undefined*. Consider the following *complete partial order* over partial functions, which is well known from denotational semantics:

$$F(x) \leq_F G(x) \text{ iff } F(x) = G(x) \text{ for all } x \in Dom(F)\,.$$

Regarding this order, the set of approximating functions builds a chain. The assumed target function $\mathbf{F}$ is defined as the supremum of this chain.

Now the hypothesized target function is defined, in terms of recurrence relations. In his synthesis theorem and its corollaries, Summers shows how a function defined this way can be expressed by a recursive program.[1]

Theorem 1 ([37])**.** *If* $\mathbf{F}$ *is defined in terms of recurrence relations as in* (1) *for* $j \leq i \in \mathbb{N}$ *then the following recursive program is identical to* $\mathbf{F}$*:*

$$\begin{aligned}
F(x) &= (p_1(x) \to f_1(x), \ldots, p_{j-1}(x) \to f_{j-1}(x), \\
&\qquad T \to G(x)) \\
G(x) &= (p_j(x) \to f_j(x), \ldots, p_{j+n-1}(x) \to f_{j+n-1}(x), \\
&\qquad T \to C[G(\sigma(x))])\,.
\end{aligned}$$

[1] This works, in a sense, reverse to interpreting a recursively expressed function by the partial function given as the fixpoint of the functional of the recursive definition. In the latter case we have a recursive program and want to have the particular partial function computed by it—here we have a partial function and want to have a recursive program computing it.

Example 4. The recurrence relations from Example 3 with $i \geq 2$ define the function $\mathbf{F}$ to be the *Init*-function. According to the synthesis theorem, the resulting program is:

$$\begin{aligned}
F(x) &= (atom(cdr(x)) \to nil, T \to G(x)) \\
G(x) &= \,(atom(cddr(x)) \to cons(car(x), nil), \\
&\qquad\qquad T \to cons(car(x), G(cdr(x))))\,.
\end{aligned}$$

Introducing Additional Variables. It may happen that no recurrent differences can be found between a chain of fragments and/or predicates. In this case, the fragments/predicates may be generalized by replacing some common subterm by an additional variable. In the generalized fragment/predicate chain recurrent differences possibly exist.

3.2 Early Variants and Extensions

Two early extensions are described. A broader survey of these and other early extensions can be found in [36].

3.2.1 BMWK—Extended Forms of Recurrences

In Summers' approach, the condition for deriving a recursive function from detected differences is that the differences hold—starting from an initial index j and for a particular interval n—recurrently along fragments and predicates with a constant context $C[\,]$ and a constant substitution σ for x. The BMWK[2] algorithm [14] generalizes these conditions by allowing for contexts and substitutions that are different in each difference. Then a found sequence of differences originates a sequence of contexts and substitutions each. Both sequences are considered as fragments of new *subfunctions*. The BMWK algorithm is then recursively applied to these new fragment sequences, hence features the automatic introduction of (necessary) subfunctions.

Furthermore, Summers' ad-hoc method to introduce additional variables is systematized by computing *least general generalization (lgg)* [31] of successive fragments.

3.2.2 Biermann et al—Pruning Enumerative Search Based on Recurrences within Single Traces

Summers objective was to avoid search and to justify the synthesis by an explicit inductive inference step and a subsequent proven-to-be-correct program construction step. This

[2] This abbreviates *Boyer-Moore-Wegbreit-Kodratoff.*

could be achieved by a restricted program schema and the requirement of a well chosen set of I/O-examples.

On the contrary, Biermann's approach [3] is to employ traces (fragments) to speed up an exhaustive enumeration of a well-defined program class, the so-called *regular* LISP *programs*. Biermann's objectives regarding the synthesis were

1. *convergence* to the class of regular LISP programs,
2. convergence on the basis of *minimal input information*,
3. robust behavior on different inputs.

Particularly 2 and 3 are contradictory to the recurrence detection method—by 2 Biermann means that no synthesis method exists which is able to synthesize every regular LISP program from fewer examples and by 2 he means that examples may be chosen randomly.

3.3 From LISP to Term Rewriting Systems

At the beginning of Sec. 3.1 we stated the LISP primitives as used in programs induced by Summers' method (as well as by BMWK and Biermann's method). This selection is crucial for the first step, the deterministic construction of first approximations, yet not for the generalization step. Indeed, the latter is independent from particular primitives, it rather relies on matching (sub)terms over arbitrary first-order signatures. Two recent systems inspired by Summers' recurrence detection method use *term rewriting systems* over first-order signatures to represent programs. Special types of TRSs can be regarded as (idealized) functional programs.

A *term rewriting system (TRS)* is a set of directed equations or *(rewrite) rules*. A rule is a pair of first-order terms $\langle l, r\rangle$, written $l \to r$. The term l is called *left-hand side (lhs)*, r is called *right-hand side (rhs)* of the rule.

We get an *instance* of a rule by applying a substitution σ to it: $l\sigma \to r\sigma$. The instantiated lhs $l\sigma$ is called *redex* (*red*ucible *ex*pression). *Contracting* a redex means replacing it by its rhs. A *rewrite step* consists of contracting a redex within an arbitrary context: $C[l\sigma] \to C[r\sigma]$. The *one-step rewrite relation* $\to$ of a rule is defined by the set of its rewrite steps. The *rewrite relation* $\overset{*}{\to}$ of a rule is the reflexive transitive closure of $\to$. The rewrite relation of a TRS R, $\to_R$, is the union of the rewrite relations of all its rules.

3.3.1 IGOR1—Inducing Recursive Program Schemes

The system IGOR1 [18] induces *recursive program schemes (RPSs)*. An RPS is a special form of TRS: The signature is divided into two disjoint subsets $\mathcal{F}$ and $\mathcal{G}$, called *unknown* and *basic* functions, respectively; rules have the form $F(x_1, \ldots, x_n) \to t$ where $F \in \mathcal{F}$ and the x_i are variables, and there is exactly one rule for each $F \in \mathcal{F}$.

IGOR1's program schema is more general than Summers' in that recursive subfunctions are found automatically with the restriction that (recursive) calls of defined functions may not be nested in the rhss of the equations. Furthermore, additional parameters are introduced systematically.

(Mutually) recursive RPSs do not terminate. Their standard interpretation is the infinite term defined as the limit $\lim_{n\to\infty, F(\boldsymbol{x})\overset{n}{\to}t} t$ where F denotes the main rule of the RPS. One gets finite approximations by replacing infinite subterms by the special symbol Ω, meaning *undefined*. Certainly, such an infinite tree and its approximations contain recurrent patterns because they are generated by *repeatedly* replacing instances of lhss of the rules by instances of rhss. IGOR1 takes a finite approximation of some (hypothetical) infinite tree as input, discovers the recurrent patterns in it, and builds, based on these recurrences, an RPS R such that the input is a finite approximation of the infinite tree of R.

Example 5. For a simple example without subfunctions (the Init function again), consider the finite approximation of some unknown infinite term:

$$\begin{aligned}&\boldsymbol{\mathit{if}}(\mathit{atom}(\mathit{cdr}(\ \ x\ \ )), \mathit{nil},\\ &\quad \mathit{cons}(\mathit{car}(\ \ x\ \ ),\\ &\qquad \boldsymbol{\mathit{if}}(\mathit{atom}(\mathit{cdr}(\ \ \mathit{cdr}(x)\ \ )), \mathit{nil},\\ &\qquad\quad \mathit{cons}(\mathit{car}(\ \ \mathit{cdr}(x)\ \ ),\\ &\qquad\qquad \boldsymbol{\mathit{if}}(\mathit{atom}(\mathit{cdr}(\ \ \mathit{cdr}(\mathit{cdr}(x))\ \ )), \mathit{nil},\\ &\qquad\qquad\quad \mathit{cons}(\mathit{car}(\ \ \mathit{cdr}(\mathit{cdr}(x))\ \ ),\\ &\qquad\qquad\qquad \Omega)))))).\end{aligned}$$

At the path from the root to Ω, where the latter denotes the unknown infinite subterm of the infinite target term and hence, which has been generated by an unknown recursive RPS, we find a recurring sequence of if-cons pairs. This leads to the hypothesis that a replacement of the lhs of a recursive rule by its rhs has taken place at the if-positions. The term is divided at these positions leading to three segments (assume, the break-positions are replaced by Ω). An approximation of the assumed rhs is computed as the lgg of the segments: $\mathit{if}(\mathit{atom}(\mathit{cdr}(x)), \mathit{nil}, \mathit{cons}(\mathit{car}(x), \Omega))$.

The Ω denotes the still unknown recursive call. The non-equal parts of the segments, which are replaced by the variable x in the lgg, are highlighted by extra horizontal space in the term. These parts must have been generated by the substitution $\{x \leftarrow \mathit{cdr}(x)\}$ in the recursive call. Denoting the induced function by F, it is now correctly defined as

$$F(x) \to \mathit{if}(\mathit{atom}(\mathit{cdr}(x)), \mathit{nil}, \mathit{cons}(\mathit{car}(x), F(\mathit{cdr}(x)))).$$

Different methods to construct a finite approximation as first synthesis step have been proposed. In [18], an extension of Summers' first step is described. Examples need not be linearly ordered and nested `if-then-else`-conditionals are used instead of the McCarthy conditional. In [34], *universal planning* is proposed as first step.

3.4 IGOR2—Combining Search and Analytical Techniques

All methods based on Summers' seminal work described so far suffer from strong restrictions regarding their general program schemas, the commitment to a small fixed set of primitives, and, at least the early methods, to the requirement of linearly ordered I/O-examples.

The system IGOR2 [17] aims to overcome these restrictions, but not at the price of falling back to generate-and-test search (cp. Sec. 5). IGOR2 conducts a search in program space, but the transformation operators are data-driven and use techniques such as matching and least generalizations, similar to the methods described so far. In contrast to generate-and-test search, only programs being correct with respect to the I/O-examples in a particular sense (but possibly unfinished) are generated. This narrows the search tree and makes testing of generated programs unnecessary.

Programs (as well as I/O-examples and background knowledge) are represented as *constructor (term rewriting) systems (CSs)*. CSs can be regarded as an extension of RPSs: The function sets $\mathcal{F}$ and $\mathcal{G}$ are called *defined functions* and *constructors*, respectively. The arguments of a defined function symbol in a lhs need not be variables but may be terms composed of constructors and variables and there may be several rules for one defined function. This extension corresponds to the concept of *pattern matching* in functional programming. One consequence of the CS representation is that I/O-examples themselves already constitute "programs", CSs. Hence, rewriting outputs into fragments to get a first approximation (Sec. 3.1.1) is not necessary anymore.

IGOR2 is able to construct complex recursive CSs containing several base- and (mutually) recursive rules, automatically identified and introduced recursive subfunctions, and complex compositions of function calls. Several interdependent functions can be induced in one run. In addition to I/O-examples, background knowledge may be provided.

3.5 Discussion

Summers' important insights were first, how the algebraic properties of data-structures can be exploited to construct program fragments and predicates without search and second, that fragments (and predicates) for different I/O-pairs belonging to one recursively defined function share recurrent patterns that can be used to identify the recursive definition. Obviously, it is necessary for recurrence detection that I/O-examples are not randomly chosen but that they consist of the first $k \in \mathbb{N}$ examples regarding the underlying order on S-expressions, i.e., that they are *complete* up to some level.

If the general schema of inducible functions becomes more complex, e.g., if subfunctions can be found automatically, and/or if background knowledge is allowed, then search is needed. IGOR2 shows that Summers' ideas for generalization can be integrated into search operators.

Search is also needed if the goal is to induce programs based on minimal sets of randomly chosen examples. In this case, the recurrence detection method cannot be applied. Biermann's method shows that it is possible for particular program classes to use fragments as generated in Summers' first step to constrain an exhaustive search in program space.

4. Inductive Logic Programming

Inductive Logic Programming (ILP) [26, 28] is a branch of machine learning [23]—intensional concept descriptions are learned from (counter-)examples, called *positive and negative examples*. The specificity of ILP is its basis in computational logic: First-order clausal logic is used as uniform language for hypotheses, examples, and background knowledge, semantics of ILP is based on entailment, and inductive learning techniques are derived by inverting deduction.

Horn clause logic together with resolution constitutes the (Turing-complete) programming language PROLOG. Program synthesis is therefore principally within the scope of ILP and has been regarded as one application field of ILP [26]. One of the first ILP systems, MIS [35], is an automatic programming/debugging system. Today, ILP is concerned with (relational) data-mining and knowledge discovery and program synthesis does not play a role anymore.

4.1 Preliminaries

An *atom* is a predicate symbol applied to arguments, a literal is an atom or negated atom. A *clause* is a (possible empty) disjunction of literals, a *Horn clause* is a clause with at most one positive literal, a *definite clause* is a clause with exactly one positive literal. A *definite program* is a finite set of definite clauses. A definite clause C consisting of the positive literal A and the negative literals $\neg B_1, \ldots, \neg B_n$ is equivalent to $B_1 \wedge \ldots \wedge B_n \rightarrow A$, written $A \leftarrow B_1, \ldots, B_n$.

4.2 Overview

In the definite setting, hypotheses and background knowledge are definite programs, examples are ground atoms. The following two definitions state the ILP problem with respect to the so-called *normal semantics*.[3]

Definition 2. Let Π be a definite program and E^+, E^- be positive and negative examples. Π is

complete with respect to E^+ iff $\Pi \models E^+$,
consistent with respect to E^- iff $\Pi \not\models e$ for every $e \in E^-$,
correct with respect to E^+ and E^- iff it is complete with respect to E^+ and consistent with respect to E^-.

Definition 3. Given

- a set of possible hypotheses (definite programs) $\mathcal{H}$,
- positive and negative examples E^+, E^-,
- consistent background knowledge B (i.e., $B \not\models e$ for every $e \in E^-$) such that $B \not\models E^+$,

find a hypothesis $H \in \mathcal{H}$ such that $H \cup B$ is correct with respect to E^+ and E^-.

Entailment ($\models$) is undecidable in general and for Horn clauses, definite programs, and between definite programs

[3] There is also a *non-monotonic* setting in ILP where hypotheses need not entail positive examples but only state true properties. This is useful for *data mining* or *knowledge discovery* but not for program synthesis, so we do not consider it here.

and single atoms in particular. Thus, in practice, different decidable (and preferably also efficiently computable) relations, which are sound but more or less incomplete, are used. We say that a hypothesis *covers* an example if it can be proven true from the background knowledge and the hypothesis. That is, a hypothesis is regarded correct if it, together with the background knowledge, covers all positive and no negative examples. Two commonly used notions are:

Extensional coverage. Given a clause $C = A \leftarrow B_1, \ldots, B_n$, a finite set of ground atoms B as background knowledge, positive examples E^+, and an example e, C *extensionally covers* e iff there exists a substitution θ such that $A\theta = e$ and $\{B_1, \ldots, B_n\}\theta \subseteq B \cup E^+$.

Intensional coverage. Given a hypothesis H, background knowledge B, and an example e, $H \cup B$ *intensionally covers* e iff e can be proven true from $H \cup B$ by applying some terminating theorem proving technique, e.g., depth-bounded SLD-resolution.

Example 6. As an example for extensional coverage, suppose $B = \emptyset$ and $E^+ = \{\ \mathit{Init}([c],[\,]), \mathit{Init}([b,c],[b]), \mathit{Init}([a,b,c],[a,b])\ \}$. The recursive clause $\mathit{Init}([X|Xs],[X|Ys]) \leftarrow \mathit{Init}[Xs, Ys]$ extensionally covers the positive example $\mathit{Init}([b,c],[b])$ with $\theta = \{X \leftarrow b, Xs \leftarrow [c], Ys \leftarrow [\,]\}$.

Both extensional and intensional coverage are sound. Extensional coverage is more efficient but less complete. As an example for the latter, suppose the positive example $\mathit{Init}([c],[\,])$ is missing in E^+ in Example 6. Then the stated recursive clause together with the base clause $\mathit{Init}([X],[\,])$ still intensionally covers $e = \mathit{Init}([b,c],[b])$ yet the recursive clause *does not* extensionally cover e anymore. Obviously, extensional coverage requires that examples (and background knowledge) are complete up to some complexity (cp Sec. 3.5). Another problem with extensional coverage is that if two clauses each do not cover a negative example, both together possibly do.

Extensional and intensional coverage are closely related to the general ILP algorithm (Algo. 1) and the covering algorithm 2 as well as to the generality models θ-subsumption and entailment as described below (Sec. 4.3), respectively.

ILP is considered a search problem. Typically, the search operators to compute new candidate programs are based on the dual notions of *generalization* and *specialization* of programs respectively clauses.

Definition 4. A program Π is *more general than* a program Φ iff $\Pi \models \Phi$. Φ is said to be *more specific than* Π.

This structure of the program space provides a way for pruning. If a program is not consistent then all generalizations are also not consistent and therefore need not be considered. This dually holds for non-completeness and specializations. Algorithm 1 shows a generic ILP algorithm. Most ILP systems are instances of it.

Algorithm 1: A generic ILP algorithm.

```
Input: $B$, $E^+$, $E^-$
Output: A definite program $H$ such that $H \cup B$ is
        correct with respect to $E^+$ and $E^-$
Start with some initial (possibly empty) hypothesis $H$
repeat
    if $H \cup B$ is not consistent then specialize $H$
    if $H \cup B$ is not complete then generalize $H$
until $H \cup B$ is correct with respect to $E^+$ and $E^-$
return $H$
```

A common instance is the *covering algorithm* (Algo. 2). The individual clauses of a program are generated independently one after the other. Hence, the problem space is not the program space (*sets* of clauses) but the clause space (*single* clauses). This leads to a more efficient search.

Algorithm 2: The covering (typically interpreted extensionally) algorithm.

```
Input and Output as in Algorithm 1
Start with the empty hypothesis $H = \emptyset$
repeat
    Add a clause $C$ not covering any $e \in E^-$ to $H$
    Remove all $e \in E^+$ covered by $C$ from $E^+$
until $E^+ = \emptyset$
return $H$
```

Entailment ($\models$) as well as θ-subsumption (Sec. 4.3.1) are *quasi-orders* on sets of definite programs and clauses, resp. We associate "more general" with "greater". The operators carrying out specialization and generalization are called *refinement operators*. They map clauses to sets of (refined) clauses or programs to sets of (refined) programs. Most ILP systems explore the problem space mainly in one direction, either from general to specific (*top-down*) or the other way round (*bottom-up*). The three well-known systems FOIL [32] (top-down), GOLEM [27] (bottom-up), and PROGOL [25] (mixed) are instantiations of the covering algorithm.

Example 7. For an example of the covering algorithm, let B and E^+ be as in Example 6 and E^- all remaining instantiations for the "inputs" $[c], [b,c], [a,b,c]$, e.g., $\mathit{Init}([b,c],[c])$. Let us assume that a (base-)clause $\mathit{Init}([X],[\,])$ is already inferred and added and hence, the covered example $\mathit{Init}([c],[\,])$ is deleted from E^+. Assume, our instantiation of the covering algorithm is a top-down algorithm. This means, each clause is found by starting with a (too) general clause and successively specializing it until no negative examples are covered anymore. Let us start with the clause $\mathit{Init}([X|Xs], Ys) \leftarrow$. It covers all remaining positive but also all corresponding negative examples; it is too general. Applying the substitution $\{Ys \leftarrow [X|Ys]\}$ specializes it to $\mathit{Init}([X|Xs],[X|Ys]) \leftarrow$. This excludes some negative examples (e.g., $Init([b, c], [c])$). Adding the literal $Init(Xs, Ys)$ to the body again specializes the clause to $Init([X|Xs], [X|Ys]) \leftarrow Init(Xs, Ys)$. All remaining positive examples are still covered but no negative example is covered anymore. Hence, the clause is added and the algorithm returns the two inferred clauses as solution.

Both specializations were refinements under θ-subsumption (Sec. 4.3.1, "Refinement Operators").

4.3 Generality Models and Refinement Operators

Instead of entailment ($\models$), θ-subsumption is often used in ILP as generality model. It is incomplete with respect to $\models$ but decidable, simple to implement, and efficiently computable. If we have background knowledge B, then we are not simply interested in whether a clause C is more general than a clause D but in whether C together with B is more general than D (together with B). This is captured by the notions of *relative* (to background knowledge) entailment respectively θ-subsumption.

4.3.1 Refinement under (Relative) θ-subsumption

Definition 5. Let C and D be clauses and B a set of clauses.

C θ-subsumes D, written $C \succeq D$, iff there exists a substitution θ such that $C\theta \subseteq D$.

C θ-subsumes D relative to B, written $C \succeq_B D$, if $B \models C\theta \rightarrow D$ for a substitution θ.

A Horn clause language quasi-ordered by θ-subsumption with an additional bottom element is a lattice. This does not generally hold for relative subsumption. Least upper bounds are called *least general generalizations (lgg)* [31]. Lggs and greatest lower bounds are computable and hence may be used for generalization and specialization. though they do not properly fit into our general notion of refinement operators because they neither map single clauses to sets of clauses nor single programs to sets of programs.

A useful restriction is to let background knowledge be a finite set of ground literals. In this case, lggs exist under subsumption relative to B and can be reduced to (non-relative) lggs. The bottom-up system GOLEM uses this scenario.

In general, (relative) θ-subsumption is sound but not complete. If $C \succeq D$ ($C \succeq_B D$) then $C \models D$ ($C \cup B \models D$) but not vice versa. For a counter-example of completeness let $C = P(f(X)) \leftarrow P(X)$ and $D = P(f(f(X))) \leftarrow P(X)$ then $C \models D$[4] but $C \not\succeq D$. As the example indicates, the incompleteness is due to *recursive* rules and therefore especially critical for *program synthesis*.

Refinement Operators. A specialization operator refines a clause by

- applying a substitution for a single variable or
- adding one most general literal.

A generalization operator uses inverse operations.

[4] D is simply the result of self-resolving C.

Application of these operators is quite common in ILP, e.g., in the systems MIS, FOIL, GOLEM, and PROGOL.

4.3.2 Refinement under (Relative) Entailment

Due to the incompleteness of θ-subsumption regarding recursive clauses, refinement under (relative) entailment has been studied. *Relative* entailment is defined as follows:

Definition 6. Let C and D be clauses and B a finite set of clauses. Then *C entails D relative to B*, denoted $C \models_B D$, if $\{C\} \cup B \models D$.

Neither lggs nor greatest specializations exist in general for Horn clause languages ordered by (relative) entailment.

Refinement Operators. Roughly speaking, entailment is equivalent to resolution plus θ-subsumption. This leads to specialization operators under (relative) entailment. Objects of refinement under entailment are not single clauses but *sets* of clauses, i.e., programs. A specialization operator under entailment refines a definite program by

- Adding a resolvent of two clauses or
- adding the result of applying the θ-subsumption specialization operator to a clause or
- deleting a clause.

4.4 Automatic Programming Systems

The three general-purpose systems FOIL, GOLEM, PROGOL are successful in learning non-recursive concepts from large data sets, yet have problems to learn recursive programs: Due to their use of the covering approach (extensional coverage), they need complete example sets and background knowledge to induce recursive programs. Since they (at least FOIL and GOLEM) explore (i) only the θ-subsumption lattice of clauses and (ii) do this greedily, correct clauses may be passed. Finally, their objective functions in the search for clauses is to cover as many as possible positive examples. Yet base clauses typically cover only few examples such that these systems often fail to induce correct base cases.

Hence ILP systems especially designed to learn *recursive* programs have been developed. They address different issues: Handling of random examples, predicate invention, usage of general programming knowledge, and usage of problem-dependent knowledge of the user, which goes beyond examples. A comprehensive survey of automatic programming ILP systems can be found in [8].

Inverting entailment by structural analysis. Several systems—CRUSTACEAN [1], CLAM [33], TIM [12], MRI [9]—address the issue of inducing *recursive* programs from *random* examples by inverting entailment based on structural analysis, similar to Sec. 3, instead of searching in the θ-subsumption lattice. These systems also have similar restrictions regarding the general schema of learnable programs. However, some of them can use background knowledge; MRI can find more than one recursive clause.

Top-down induction of recursive programs. Top-down systems can principally—even if they explore the θ-subsumption clause-lattice only—generate arbitrary (in particular all recursive) Horn clauses.[5] Thus, if a top-down covering system would use intensional instead of extensional coverage, it could principally induce recursive programs from *random* examples. Certainly, this would require to find clauses in a particular order—base clauses first, then recursive clauses, only depending on base clauses and themselves, then recursive clauses, only depending on base clauses, the previously generated recursive clauses, and themselves, and so on. This excludes programs with mutually interdepending clauses. The system SMART [24] is based on these ideas. It induces programs consisting of one base clause and one recursive clause. Several techniques to sensibly prune the search space allows for a more exhaustive search than the greedy search applied by FOIL, such that the incompleteness issue of θ-subsumption-based search is weaken.

The system FILP [2] is a covering top-down system that induces *functional* predicates only, i.e., predicates with distinguished input- and output parameters, such that for each binding of the input parameters exactly one binding of the output parameters exists. This makes negative examples unnecessary. FILP can induce multiple interdependent predicates/functions where each may consist of several base- and recursive clauses. Hence, intensional coverage is not assured to work. FILP starts with a few randomly chosen examples and tries to use intensional covering as far as possible. If, during the intensional proof of some example, an instance of the input parameters of some predicate appears for which an output is neither given by an example nor can be derived intensionally, then FILP queries for this "missing" example and thereby completes the example set as far as needed.

Using programming knowledge. Flener argued, in several papers, for the use of program schemas that capture general program design knowledge like divide-and-conquer, generate-and-test, global-search etc., and has implemented this in several systems. He distinguishes between schema-*based* systems inducing programs of a system-inherent schema only and schema-*guided* systems, which take schemas as dynamic, problem-dependent, additional input and thus are more flexible. Flener's DIALOGS [6] system uses schemas and strong queries to restrict the search space and thereby is able to efficiently induce comparatively complex programs including predicate invention.

Jorge and Brazdil have—besides for *clause structure grammars* defining a program class and thus similar to schemas as dynamic language-bias—argued for so called *algorithm sketches*. An algorithm sketch is problem-dependent algorithm knowledge about the target function and provided by the user in addition to examples. This idea is implemented in their SKIL and SKILIT systems [13].

[5] Hence, although θ-subsumption is incomplete with respect to entailment due to recursive clauses, every clause, in particular the recursive clauses, can be generated by refinement based on θ-subsumption—if one searches top-down starting from the empty clause or some other clause general enough to θ-subsume the desired clauses.

4.5 Discussion

Compared to the classical approaches in Sec.3 (except for IGOR2), ILP has broadened the class of inducible relations by allowing for background knowledge, using particular search methods and other techniques (Sec. 4.4).

Shapiro [35] and Muggleton and De Raedt [26] argued for clausal logic as universal language in favor to other universal formalisms such as Turing machines or LISP. Their arguments are: (i) Syntax and semantics are closely and in a natural way related. Hence if a logic program makes errors, it is possible to identify the erroneous clause. Furthermore, there are simple and efficient operations to manipulate a logic program with predictable semantic effects (cp. Sec. 4.3.1). Both is not possible for, say, Turing machines. (ii) It suffices to focus on the logic of the program, control is left to the interpreter. In particular, logic programs (and clauses) are *sets* of clauses (and literals), order does not matter.

The first argument carries over to other declarative formalisms such as equational logic, term rewriting, and functional logic programming (FLIP [5] is an IPS system in this formalism). The second argument also carries over to some extent, declarative programming all in all shifts the focus off control and to logic. Yet in this generality it only holds for non-recursive programs or ideal, non-practical, interpreters. For the efficient interpretation of recursive programs however, order of clauses in a program and order of literals in a clause matters. Hence we think that declarative, (clausal- and/or equational-)logic-based formalisms are principally equally well suited for IPS.

Logic programs represent general relations. (Partial) functions are special relations—their domains are distinguished into source and target (or: a functional relation has input and output parameters) and they are single-valued (each instantiation of the input parameters implies a unique instantiation of the output parameters). Regarding functional- and logic programming, there is another difference: Functional programs are typically typed, i.e., their domain is partitioned and inputs and outputs of each function must belong to specified subsets, whereas logic programs are typically untyped. Interestingly, all three "restrictions" of functions compared to relations have been shown to be advantageous from a learnable point of view in ILP. The general reason is that they restrict the problem space such that search becomes more efficient and fewer examples are needed to describe the intended function. In particular, no negative examples are needed since they are implicitly given by the positive ones.

ILP is built around the natural generality structure of the problem space. Regarding *functional* relations, we observe an "oddity" of this structure. For definite programs, "more general", with respect to the minimal Herbrand model, means "more atoms". If the relation is a *function*, an additional ground atom must have a different instantiation of the input parameters compared to all other included atoms. Thus, "more general" in the case of definite programs representing functions reduces to "greater domain". In other words: *All functions with the same domain are incomparable with respect to generality*. Since most often one is interested in total functions, generality actually provides *no structure at all* of the space of possible solutions.

5. Functional Generate-and-Test Approaches

The functional IPS methods in this third block have in common that their search is generate-and-test based. I/O-examples are not used as a means to *construct* programs but only to *test* generated programs.

5.1 Genetic Programming

Genetic programming (GP) [20], like other forms of *evolutionary algorithms* is inspired by biological evolution. GP systems maintain *populations* of candidate solutions, get new ones by stochastical methods like *reproduction*, *mutation*, *recombination/crossover*, and *selection*, and thereby try to maximize *fitness*. Evolutionary search can be useful when the problem space is too broad to conduct an exhaustive search and simultaneously nothing or few is known about the *fitness landscape*, i.e., when it is not possible to construct sensible heuristics. The randomness of the search cares for a widespread exploration of the problem space which is guided by the fitness measure. On the other side, this "chaotic" search in a space with unknown properties makes it difficult to give any guaranties regarding solutions and leads to only approximated solutions. A GP problem is specified by *fitness cases* (e.g., example inputs of the target function), a fitness function, and primitives to be used in evolved expressions. There are no predefined goal criteria or preference biases in GP systems. The search is completely guided by the fitness function that is to be maximized.

Data structures and recursion do not play a predominant role in GP. A typical evolved program is an arithmetic expression or a propositional formula. Koza and his colleagues [21] integrated recursion into GP. One of the major issues is the handling of non-terminating programs. As a generate-and-test approach, GP relies on testing evolved candidate programs against the given examples. If non-termination may appear then a runtime limit is applied. This raises two problems if non-terminating programs are frequently generated: (i) The difficulty of assigning a fitness value to an aborted program and (ii) the runtime uselessly consumed by evaluating non-terminating programs. Wong and Mun [38] deal with this problem by a meta-learning approach to decrease the possibility of evolving non-terminating programs.

Others try to avoid non-termination completely: In her system POLYGP [39], Yu integrates *implicit recursion* through the use of user-provided higher-order functions. Kahrs [15] evolves *primitive recursive* functions over the natural numbers. Binard and Felty [4] evolve programs in SYSTEM F, a typed lambda calculus where only total recursive functions are expressible. The primitive recursive functions are contained as proper subclass.

Hamel and Shen [10] have developed a method lying in the intersection of ILP, GP and algebraic specification. They evolve (recursive) algebraic specifications, i.e., equational theories over many-sorted signatures, using GP search methods. Instead of providing a fitness function, a target theory is, as in ILP, specified by positive and negative facts—ground equations in this case. Additionally, a background theory may be provided. The fitness function to be maximized is derived from such a specification. Candidate theories satisfying more positive facts, excluding more negative facts, and being of smaller syntactical complexity are preferred.

5.2 ADATE

The ADATE system [29], to our knowledge the most powerful inductive programming system regarding inducible programs, is an evolutionary system in that it maintains a population of programs and performs a greedy search guided by a fitness function. Yet unlike GP, it is especially designed to evolve *recursive* programs and applies sophisticated program transformation operators, search strategy, and program evaluation functions to this end.

Programs are represented in ADATE-ML, a subset of STANDARD ML. Programs are rated according to a user-provided output evaluation function, user provided preference biases, and syntactical and computational complexity.

5.3 Systematic Enumeration of Programs

Two further recent methods, MAGICHASKELLER [16] and the software testing system G∀ST [19] essentially systematically enumerate programs of a certain class.

MAGICHASKELLER uses higher-order functions as background knowledge. Katayama argues that by using higher-order functions, programs can be represented in a compact form and by using strong typing, the problem space is narrowed such that a simple brute-force enumeration of programs could make sense. He furthermore considers MAGICHASKELLER as a base-line which could be used to evaluate the performance of more sophisticated methods. As a first result, Katayama compares MAGICHASKELLER and POLYGP for the problems *Nth*, *Length*, and *Map*, and states that POLYGP, in contrast to MAGICHASKELLER, needs different higher-order functions for each of these problems, needs several runs to find a solution, needs additional parameters to be set, and, nevertheless, consumes more time to induce a solution.

5.4 Discussion

One general advantage of generate-and-test methods is their greater flexibility, in at least to aspects: First regarding the problem space—there are no *principle* difficulties in enumerating even very complex programs. Second regarding the form of the incomplete specification. Whereas the search operators of an analytical technique depend on the specification (e.g., I/O-examples) such that different forms of specification need different search operator techniques, the search is more independent from the specification in generate-and-test methods such that more expressive forms of specification can easily be integrated. In particular, fitness functions in GP or the objective function in ADATE are more expressive than I/O-examples since no fixed outputs need to be provided but general *properties* to be satisfied by computed outputs can be specified.

The disadvantage of generate-and-test methods is that they generally generate far more candidate programs until a solution is found and hence need much more time than data-driven methods to induce programs of equal size. Several analytical and generate-and-test systems have been compared empirically in [11]. A further problem is non-termination. As generated programs need to be tested against the provided examples, non-termination is a serious issue. Higher-order functions or formalisms that a-priori only include total functions are helpful to circumvent this problem.

6. Conclusions and Further Research

In the previous sections, we described several approaches and systems to the inductive synthesis of functional and logic programs and discussed pros and cons and relations between them.

One obvious dimension to classify them is the way of how example data is used: As basis to construct candidate solutions (Sec. 3) or to test and evaluate independently generated candidates (Sec. 5). (In ILP, both approaches are found.) The analytical approach tends to be faster because many representable programs are a priori excluded from being generated. On the other side, since it strongly depends on the data and the language bias, it is much less robust and flexible regarding the whole problem specification including types of data, preference-, and language biases. Besides further developing both general approaches separately, we think that examining ways to combine them could be useful to achieve a satisfiable combination of robustness, flexibility, expressiveness, and efficiency. Our system IGOR2 and the well-known ILP system PROGOL indicate the potential of such an integration.

One important topic, that certainly has not received sufficient attention in the context of inductive program synthesis, is learning theory, including models of learning and criteria to evaluate candidate programs. PAC-learning, the predominant learning model in machine learning, is well-suited for restricted representation languages and noisy data, hence approximate solutions. Yet in program synthesis, we have rich representation languages, often assume error-free examples, and want have programs that *exactly* compute an intended function or relation. Moreover, efficiency, not only of the induction process, but of the induced program, becomes an important issue. Muggleton's *U-learning* model[6] captures these needs and is probably a good model or initial point to develop learning models for inductive program synthesis.

There has certainly been significant progress since the beginnings in the seventies. Yet inductive program synthesis still is not yet in a status to be applied to real problems. We think that it is now time for a more target-oriented approach. This does not mean to replacing general approaches by problem-dependent ad hoc techniques. We rather think that identifying and promoting specific application fields and domains could help to spark broader interest to the topic as well as to sensibly identify strengths and weaknesses of existing methods, to extend them and to identify possibilities to integrate them in a useful way.

In the context of software engineering, we think that *test-driven development (TDD)* would be a good starting point to bring IPS to application. The paradigm requires preparing tests "(incompletely) defining" a function *before* coding it. Hence, IPS could smoothly fit in here. Moreover, TDD typically features a strong modularization such that only small entities need to be synthesized.

Within algorithms research, one could try to find (classes) of problems for which "better" than currently known algorithms are expected to exist and to apply IPS methods to them. One such domain are problems in *artificial general intelligence (AGI)*, a research field that again—after established AI is nowadays narrowed to many different specific problems—takes up the original goal of AI of creating artificial agents that reason and act human-like. There is a serious interest in IP in the currently emerging AGI community.

References

[1] D. W. Aha, S. Lapointe, C. X. Ling, and S. Matwin. Inverting implication with small training sets. In *Proceedings of the European Conference on Machine Learning (ECML'94)*, volume 784 of *LNCS*, pages 29–48. Springer-Verlag, 1994.

[2] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, Cambridge, MA, USA, 1995.

[3] A. W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8(8):585–600, 1978.

[4] F. Binard and A. Felty. Genetic programming with polymorphic types and higher-order functions. In *Proceedings of the 10th annual Conference on Genetic and Evolutionary Computation (GECCO'08)*, pages 1187–1194, New York, NY, USA, 2008. ACM.

[5] C. Ferri-Ramírez, J. Hernández-Orallo, and M. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 233–247. Springer-Verlag, 2001.

[6] The 'U' stands for 'universal'.

[6] P. Flener. Inductive logic program synthesis with DIALOGS. In S. Muggleton, editor, *Selected Papers of the 6th International Workshop on Inductive Logic Programming, (ILP'96)*, volume 1314 of *LNCS*, pages 175–198, 1997.

[7] P. Flener and D. Partridge. Inductive programming. *Automated Software Engineering*, 8(2):131–137, 2001.

[8] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming*, 41(2-3):141–195, 1999.

[9] M. Furusawa, N. Inuzuka, H. Seki, and H. Itoh. Induction of logic programs with more than one recursive clause by analyzing saturations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP'97)*, volume 1297 of *LNCS*, pages 165–172. Springer-Verlag, 1997.

[10] L. Hamel and C. Shen. An inductive programming approach to algebraic specification. In *Proceedings of the 2nd Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 3–14, 2007.

[11] M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence*, pages 55–60. Atlantis, 2009.

[12] P. Idestam-Almquist. Efficient induction of recursive definitions by structural analysis of saturations. In *Advances in Inductive Logic Programming*. IOS Press, 1996.

[13] A. M. G. Jorge. *Iterative Induction of Logic Programs*. PhD thesis, Departamento de Ciência de Computadores, Universidade do Porto, 1998.

[14] J.-P. Jouannaud and Y. Kodratoff. Program synthesis from examples of behavior. In A. W. Biermann and G. Guiho, editors, *Computer Program Synthesis Methodologies*, pages 213–250. D. Reidel Publ. Co., 1983.

[15] S. Kahrs. Genetic programming with primitive recursion. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, pages 941–942, New York, NY, USA, 2006. ACM.

[16] S. Katayama. Systematic search for lambda expressions. In M. C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007.

[17] E. Kitzelmann. Analytical inductive functional programming. In *18th International Symposium on Logic-Based Program Synthesis and Transformation, Revised Selected Papers*, volume 5438 of *LNCS*, pages 87–102. Springer-Verlag, 2009.

[18] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[19] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. GAST: Generic automated software testing. In *Implementation of Functional Languages (IFL'02)*, volume 2670 of *LNCS*. Springer, 2003.

[20] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[21] J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[23] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[24] C. R. Mofizur and M. Numao. Top-down induction of recursive programs from small number of sparse examples. In *Advances in Inductive Logic Programming*. IOS Press, 1996.

[25] S. H. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13:245–286, 1995.

[26] S. H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[27] S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsha, 1990.

[28] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNAI*. Springer-Verlag, 1997.

[29] J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55 – 83, 1995.

[30] D. Partridge. The case for inductive programming. *Computer*, 30(1):36–41, 1997.

[31] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[32] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the 6th European Conference on Machine Learning*, LNCS, pages 3–20. Springer-Verlag, 1993.

[33] R. Rios and S. Matwin. Efficient induction of recursive prolog definitions. In *Proceedings of the 11th Conference of the Canadian Society for Computational Studies of Intelligence*, volume 1081 of *LNCS*, pages 240–248. Springer, 1996.

[34] U. Schmid. *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *LNAI*. Springer, Berlin; New York, 2003.

[35] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[36] D. R. Smith. The synthesis of LISP programs from examples: A survey. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 307–324. Macmillan, 1984.

[37] P. D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.

[38] M. Wong and T. Mun. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421–455, 2005.

[39] T. Yu. Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction. *Genetic Programming and Evolvable Machines*, 2(4):345–380, 2001.

Incremental Learning in Inductive Programming

Robert Henderson
University of Edinburgh, UK
robh93@googlemail.com

Abstract

Inductive programming systems characteristically exhibit an exponential explosion in search time as one increases the size of the programs to be generated. As a way of overcoming this, we introduce *incremental learning*, a process in which an inductive programming system automatically modifies its inductive bias towards some domain through solving a sequence of gradually more difficult problems in that domain.

We demonstrate a simple form of incremental learning in which a system incorporates solution programs into its background knowledge as it progresses through a sequence of problems. Using a search-based inductive functional programming system modelled on the MagicHaskeller system of Katayama (2007), we perform a set of experiments comparing the performance of inductive programming with and without incremental learning. Incremental learning is shown to produce a performance improvement of at least a factor of thirty on each of the four problem sequences tested. We describe how, given some assumptions, inductive programming with incremental learning can be shown to have a polynomial, rather than exponential, time complexity with respect to the size of the program to be generated. We discuss the difficulties involved in constructing suitable problem sequences for our incremental learning system, and consider what improvements can be made to overcome these difficulties.

Keywords Inductive programming, Inductive functional programming, Incremental learning

1. Introduction

Inductive Programming (IP) differs from more conventional machine learning techniques in that it features the use of a general, expressive programming language as a space of hypotheses for describing patterns in data. Herein lies both the attraction and the apparent downfall of IP: having such an expressive hypothesis space allows IP to be used to model complex or recursive patterns that simply cannot be represented with the more conventional methods (feedforward neural networks or decision trees, for example). On the other hand, this expressivity also means that IP methods can become intractable very quickly when applied to larger problems. State of the art IP systems such as ADATE (Olsson 1995), Igor II (Kitzelmann 2007), and MagicHaskeller (Katayama 2007) have shown promise on relatively simple arithmetic and list processing problems, but are not currently capable of synthesising the kinds of complex programs that realistic practical applications would demand. See Hofmann et al. (2009) for a recent evaluation of the capabilities of these systems.

How can we solve this dilemma, and get the benefits of a general, expressive hypothesis space as well as a method that is computationally tractable? It has been proposed (Solomonoff 2002; Schmidhuber 2004) that combining IP with *incremental learning* may provide a solution. An incremental learning system is one that can automatically modify its inductive bias towards a given domain through solving a sequence of successively more difficult problems in that domain. In other words, incremental learning is about gaining the expertise required to solve hard problems through the experience of solving easier ones. If successfully equipped with an incremental learning mechanism, a system should be able to learn to solve complex problems without the need for a human expert to hand-code extensive domain-specific knowledge or algorithms into its workings.

In this paper we present experimental evidence that incremental learning is a viable means for producing orders of magnitude performance improvements in IP. We start with a review of previous work in IP that features incremental learning (section 2). We then describe the particular incremental learning mechanism to be evaluated here (section 3), and give an overview of the IP system that was used in our experiments (section 4). We present the experiments themselves along with their results, and give an explanation for these results in the form of a computational complexity argument (section 5). Finally, we discuss the limitations of our chosen incremental learning mechanism, and consider what improvements are required before it can be of practical use (section 6).

2. Previous work

Quinlan and Cameron-Jones (1993) were probably the first to demonstrate a form of incremental learning in an IP context. They showed how their inductive logic programming system, FOIL, was able to solve more than half of the problems in a sequence of 18 textbook logic programming exercises presented to it in order of gradually increasing difficulty. This was made possible by having the system add each solution program to its background knowledge as it went along. It could therefore potentially re-use solutions to earlier problems as primitive elements in the construction of solutions to later problems.

More recently, Schmidhuber et al. (1997) studied an incremental learning mechanism which they termed 'adaptive Levin search'. The idea behind adaptive Levin search is that, in a search-based IP system, the inductive bias can be controlled by weighting the different programming language primitives according to how frequently they should be used. As a system solves a succession of problems, these weights are gradually modified according to how often each primitive actually occurs in solution programs. Thus, the system becomes biased towards re-using primitives that were present in successful programs in the past. Adaptive Levin search was shown to produce some performance improvement on a selection of simple problem sequences.

Schmidhuber (2004) later followed up the work on adaptive Levin search with a fully fledged incremental learning IP system called OOPS. OOPS supported both a weight modification mechanism with a similar role to the one in adaptive Levin search, as well as an ability to invoke chunks of code from past programs in solutions to new problems. However, in the problem sequence that Schmidhuber tested, which involved solving the general 'towers of Hanoi' problem, only the weight modification mechanism was shown to provide a direct performance benefit.

Khan et al. (1998) made a brief study into incremental learning in inductive logic programming, under the name of 'repeat learning'. Using the Progol inductive logic programming system, they demonstrated how helper predicates invented in order to solve one problem may be re-used when constructing the solution to another. They chose a problem domain concerning the inference of the general descriptions of moves in chess.

In this paper, we have chosen to focus on the kind of incremental learning mechanism that was employed in FOIL, that in which a system adds solution programs to its background knowledge as it progresses through a problem sequence. As we shall see, this simple method is remarkably powerful. The main drawback of Quinlan's and Cameron-Jones' short study is that they did not provide a direct comparison between scenarios with and without incremental learning. We shall remedy that with the experiments presented here.

3. Incremental learning mechanism

We aim to give a convincing demonstration of one simple but effective incremental learning mechanism. The mechanism works as follows: a sequence of successively more difficult, but related, problems is presented to an IP system. The system must solve the problems in the order given, and will incorporate each solution program into its object language as a new primitive function (i.e. into its background knowledge) as it goes along. This addition of these new functions to the system's object language is what constitutes the modification of its inductive bias. For an appropriately designed problem sequence, we would expect the time taken for the system to solve whole the sequence, with the help of incremental learning, to be much less than if it were tasked simply with solving the final problem of the sequence in isolation.

One can see how this mechanism might be expected to work effectively by considering how, particularly in functional programming, it is often natural to express the solution to a complex problem in terms of the solutions to one or more simpler problems already solved. This breaks the program down into smaller, more managable units, and is a technique commonly known as *procedural abstraction* when used by human programmers.

4. Implementation

We implemented, for the purpose of this study, a simple brute-force search based IP system modelled on the MagicHaskeller system of Katayama (2007). We shall refer to our implemented system as 'MagicLisper' (it was written in Common Lisp). In this section, we first review MagicHaskeller and explain our reasons for choosing it, then we describe how our system differs from MagicHaskeller in a few respects. We also talk through an example usage of our system on an IP problem.

4.1 Review of MagicHaskeller

MagicHaskeller is a search-based inductive functional programming system that infers programs from input-output training examples. Its main distiguishing feature is the brute-force algorithm that it uses to synthesise solution programs. More or less, it simply generates and tests all possible programs in its object language in order of length, using a breadth-first search, until it finds one that matches the training examples. This is tractable because of two features of MagicHaskeller's object language. Firstly, the language is strongly typed, with only type-consistent programs being considered by the search algorithm. Secondly, recursion is supported not explicitly, but via the use of certain of higher-order primitive operations known as *morphisms* (Augusteijn 1998). These morphisms are essentially generalisations of standard functional programming operations such as *map* and *reduce*, and with them, many useful recursive processes can be expressed concisely. Ultimately, these two features combine to produce a search space that contains rather few obviously useless programs, allowing brute-force search to fare well.

For this investigation into incremental learning, we chose to use a system based on MagicHaskeller for two reasons. Firstly, MagicHaskeller's search algorithm is fast;

synthesising simple recursive programs takes only a matter of seconds. Secondly, the search algorithm is simple and predictable; it is easy to understand exactly why MagicHaskeller succeeds or fails in finding a solution to a given problem, which helps immensely when one is designing problem specifications. It is for this second reason in particular that we chose MagicHaskeller as our base rather than an alternative such as ADATE or Igor II.

4.2 Differences between our system and MagicHaskeller

The object language of MagicLisper has the same form as the 'de Bruijn lambda calculus' language used in the version of MagicHaskeller described in (Katayama 2007). There is one significant structural difference: for the sake of simplicity, MagicLisper's type system does not support parametric polymorphism; instead, every primitive function in its object language has one or more explicit ground types. The default library of primitive functions and constants used by MagicLisper is given in table 1. Also see figure 1 for precise definitions of the morphism primitives.

In this paper we shall use a Lisp-style notation to represent programs. So, for example, the following program (`sum-elems`), which sums the elements of a list, in Haskell notation:

```
(\ a1 -> paralist (\ a2 a3 a4 -> + a4 a2) a1 0)
```

is written in the Lisp notation as:

```
(λ (a1) (paralist (λ (a2 a3 a4) (+ a4 a2)) a1 0))
```

MagicHaskeller searches through programs in order of length, or more precisely, it searches through programs in order of the total number of functor and lexical variable invocations they contain. In MagicLisper, we generalise on this process by requiring that primitive functors each be assigned a numerical weight. Programs are synthesised in order of total weight, this being the sum of the weights of their component functor and lexical variable invocations. Lexical variables always receive a weight of 0.4. The weights of the default primitive functors range between 2.0 and 4.5 (see table 1). As an example of how to calculate the total weight of a program, consider the `sum-elems` program mentioned above, which has a weight of 12:

	`paralist`	`+`	`a4`	`a2`	`a1`	`0`	Total
Weight	4.0	3.4	0.4	0.4	0.4	3.4	12

Note that symbols occuring in lambda parameter lists do not contribute to the calculation.

The weighting feature allows one to manually bias the system towards using certain primitives by assigning them lower weights. This extra flexibility allows our system to potentially handle a larger primitive library than MagicHaskeller, since more rarely used primitives can be given higher weights to minimise their negative impact on the search performance. Note that if one sets all the weights to the same value, our search algorithm reduces to that of MagicHaskeller. In this study, the weights were chosen by hand; however, we note that for a more advanced system it would make sense to have these weights tuned automatically. To justify our choice of weight values, we have tested MagicLisper's performance on a selection of nine non-incremental problems, both with and without the customised weights (table 2). The problems all exhibit a significant increase in solution speed due to the custom weights, ranging from a factor of 2.4 to a factor of 165.7.

MagicLisper does not employ the memoisation or fusion rule optimisations of MagicHaskeller. Finally, MagicLisper requires the user to explicitly specify the maximum number of 'steps' for which to test any candidate solution program on a given training example. Each step corresponds to one evaluation by the interpreter of a sub-expression within a program, and this 'number of steps' is an approximate specification of the maximum time to spend testing each program.

4.3 Example usage of our system

Let us briefly look at MagicLisper in action on a simple problem. Consider the following specification for a function which finds the length of a list:

`()` $\rightarrow$ `0`	[10 steps]
`(8)` $\rightarrow$ `1`	[20 steps]
`(10 4 7 2)` $\rightarrow$ `4`	[50 steps]

To solve this, MagicLisper first determines the type of the program implied by the specification: in this case, it is a function mapping a list of integers to an integer. It then performs an iterative deepening search through the space of programs matching that type; on the nth iteration, it generates and tests programs whose total weight is less than or equal to n. When testing a program, MagicLisper runs it on each training input in turn, for no more than the specified number of steps in each case. The whole search finishes when MagicLisper finds the program with the smallest weight that satisfies all of the training examples, which is in this case:

```
(λ (a1) (paralist (λ (a2 a3 a4) (inc a4)) a1 0))
```

The above program has a weight of 11.6, so is found on the 12th search iteration.

5. Incremental learning experiments

In this section, we describe a set of experiments with MagicLisper that demonstrate the incremental learning mechanism of section 3, that in which solution programs are successively added to the system's object language as new primitives.

5.1 Method and results

We measured the performance of MagicLisper on four problem sequences, both with and without the aid of incremental learning in each case. Full specifications of these problem sequences along with the experimental results are given in figures 2, 3, 4, and 5. Each specification consists of a

Name	Type	Weight
— The empty list —		
`nil`	`list`	2.1
— List operations —		
`cons`	`(λ (int list) list)`	2.1
`car`	`(λ (list) int)`	3.2
`cdr`	`(λ (list) list)`	3.2
— Integer constants —		
`0`	`int`	3.4
`1`	`int`	3.4
— Integer operations —		
`inc`	`(λ (int) int)`	3.4
`dec`	`(λ (int) int)`	3.4
`+`	`(λ (int int) int)`	3.4
`*`	`(λ (int int) int)`	3.4
— If-then-else —		
`if`	`(λ (bool int int) int)`	2.5
`if`	`(λ (bool list list) list)`	2.5
— Boolean constants —		
`t`	`bool`	3.5
`f`	`bool`	3.5
— Boolean operations —		
`not`	`(λ (bool) bool)`	3.5
`and`	`(λ (bool bool) bool)`	3.5
`or`	`(λ (bool bool) bool)`	3.5
— Integer comparions operations —		
`eql`	`(λ (int int) bool)`	2.0
`<`	`(λ (int int) bool)`	2.0
— Morphisms —		
`paranat`	`(λ ((λ (int int) int) int int) int)`	4.0
`paranat`	`(λ ((λ (int list) list) int list) list)`	4.0
`paranat`	`(λ ((λ (int bool) bool) int bool) bool)`	4.0
`paralist`	`(λ ((λ (int list int) int) list int) int)`	4.0
`paralist`	`(λ ((λ (int list list) list) list list) list)`	4.0
`paralist`	`(λ ((λ (int list bool) bool) list bool) bool)`	4.0
`analist`	`(λ ((λ (list) list) list) list)`	4.5

Table 1. The default library of primitive functions and constants used by MagicLisper. The type system consists of: integers (`int`), lists of integers (`list`), and booleans (`bool`). A compound type expression of the form: `(λ (`a b `...)` r`)` represents a function whose argument types are a, b, etc., and whose return type is r. The role of the weights is to bias the system towards using certain primitives more than others when constructing programs; primitives with lower weights are used more frequently (see section 4.2).

```
(define (paranat f n x)
  (if (zero? n)
      x
      (f (- n 1) (paranat f (- n 1) x))))

(define (paralist f lst x)
  (if (null? lst)
      x
      (f (car lst) (cdr lst) (paralist f (cdr lst) x))))

(define (analist f lst)
  (let ((pair (f lst)))
    (if (null? pair)
        '()
        (cons (car pair) (analist f (cdr pair))))))
```

Figure 1. Definitions of MagicLisper's morphism primitives given in the Scheme dialect of Lisp: *natural number paramorphism*, *list paramorphism*, and *list anamorphism*.

Name	Description	Time / s (custom weights)	Time / s (uniform weights)	Speed-up factor
`append`	Appends two lists together.	< 0.1	14.5	> 145.0
`make-list`	Constructs the list of n instances of a given value.	0.1	13.3	133.0
`length`	Finds the length of a list.	0.2	1.9	9.5
`sum-elems`	Finds the sum of the elements in a list.	0.5	19.9	39.8
`evenp`	Tests if a given integer is even.	0.7	1.7	2.4
`nth`	Finds the nth element of a list.	0.9	26.0	28.9
`last-elem`	Finds the last element of a list.	1.5	248.5	165.7
`member`	Tests if a given value is a member of a list.	6.7	> 251.4	> 37.5
`pow`	Raises one integer to the power of another.	9.6	31.2	3.3

Table 2. Some typical problems that MagicLisper can solve without the aid of incremental learning. In each case, between 3 and 5 training examples were given. Solution times were measured in two different scenarios: 'custom weights', in which the lexical variable and primitive weights were set up as described in section 4.2, and 'uniform weights', in which the lexical variable and primitive weights were all set to the value 1. The 'speed-up factor' column gives the proportional increase in speed due to the custom weights: time (uniform weights) divided by time (custom weights). The measurements were made on a 2 GHz Intel Core II Duo desktop PC with 2 GB of RAM running GNU CLISP.

main problem, and a sequence of sub-problems whose solutions may act as building blocks out of which the solution to the main problem can be constructed. For example, in the `sort` problem sequence (figure 4) we tasked our system with inferring an algorithm to sort a list of numbers. Sub-problems included the simpler but related task of taking the smallest element out of a list and bringing it to the front (`extract-least-elem`), and the yet simpler tasks of finding the smallest element in a list (`least-elem`), and of removing a given element from a list (`remove-elem`).

When designing the problem sequences, we used our knowledge of how one might implement the solution programs by hand in order to choose appropriate sub-problems. We also used some degree of trial and error in tweaking the problem sequences until incremental learning worked effectively (for example, `remove-first-block` was originally the first stage in our design for the `block-lengths` problem sequence; we added an extra stage, `car-p`, when it became apparent that our system was taking too long to solve `remove-first-block` from the default starting conditions). For now, let us emphasise the point that readily comprehensible and effective problem breakdowns often exist. In the next section (6) we shall consider in detail the issue of how much human effort is required to produce these problem breakdowns, as well as what ways can be developed to reduce or remove the need for this human effort.

For every problem and sub-problem, in order to obtain some guarantee that the program found was indeed the correct general solution, we checked it against a set of test examples. When designing our problem specifications, if any

deref-list: *dereferences a list of indices into another list.*	
— Training examples —	
`(), (7) → ()`	[20 steps]
`(0), (6) → (6)`	[50 steps]
`(1 0 2), (8 6 4 5) → (6 8 4)`	[200 steps]
— Test examples —	
`(3 2 2 1 3 4 0 5), (77 42 3 −10 8 61) → (−10 3 3 42 −10 8 77 61)`	
`(8 4 7), (9 5 2 5 8 4 1 9 1 7) → (1 8 9)`	

Incremental specification

1.

nth: *returns the nth element of a list.*	
— Training examples —	
`0, (5) → 5`	[15 steps]
`1, (8 6) → 6`	[30 steps]
`3, (4 10 77 34 58) → 34`	[150 steps]
— Test examples —	
`8, (8 4 9 3 7 1 9 2 5 4 7) → 5`	
`4, (11 23 45 15 27 89 102 56) → 27`	

2. **deref-list**

Results

Stage	Time / s	Depth	Solution
nth	0.9	12	(λ (a1 a2) (car (paranat (λ (a3 a4) (cdr a4)) a1 a2)))
deref-list	3.6	13	(λ (a1 a2) (paralist (λ (a3 a4 a5) (cons (nth a3 a2) a5)) a1 nil))
Total	**4.5**		

Non-incremental: TIMEOUT (950.2 seconds, depth 18)

Figure 2. The `deref-list` problem sequence: specification and results.

reverse: *reverses a list.*	
— Training examples —	
`() → ()`	[20 steps]
`(8) → (8)`	[40 steps]
`(3 7) → (7 3)`	[150 steps]
`(9 4 7 1) → (1 7 4 9)`	[800 steps]
— Test examples —	
`(2 9 1 7 −3 4 8 9 10 12) → (12 10 9 8 4 −3 7 1 9 2)`	
`(6 4 5 2 1 1 1 8 2) → (2 8 1 1 1 2 5 4 6)`	

Incremental specification

1.

append-elem: *appends an element to the end of a list.*	
— Training examples —	
`8, () → (8)`	[15 steps]
`4, (9) → (9 4)`	[30 steps]
`7, (3 8 1) → (3 8 1 7)`	[100 steps]
— Test examples —	
`6, (4 7 1 3 9 8 6) → (4 7 1 3 9 8 6 6)`	
`3, (8 8 8 8 8) → (8 8 8 8 8 3)`	

2. **reverse**

Results

Stage	Time / s	Depth	Solution
append-elem	1.0	12	(λ (a1 a2) (paralist (λ (a3 a4 a5) (cons a3 a5)) a2 (cons a1 nil)))
reverse	0.1	10	(λ (a1) (paralist (λ (a2 a3 a4) (append-elem a2 a4)) a1 nil))
Total	**1.1**		

Non-incremental: SOLUTION FOUND (569.6 seconds, depth 19)

(λ (a1) (paralist (λ (a2 a3 a4) (paralist (λ (a5 a6 a7) (cons a5 a7)) a4 (cons a2 nil))) a1 nil))

Figure 3. The `reverse` problem sequence: specification and results.

sort: *sorts a list of integers in ascending order.*	
— Training examples —	
`() → ()`	[30 steps]
`(7) → (7)`	[100 steps]
`(4 2) → (2 4)`	[500 steps]
`(9 8 7) → (7 8 9)`	[2000 steps]
`(3 2 3 2 3) → (2 2 3 3 3)`	[10000 steps]
— Test examples —	
`(10 6 −30 7 2 5 −2 3 1 6 4) → (−30 −2 1 2 3 4 5 6 6 7 10)`	
`(1 1 1 8 6 8 6 4 3 3 1 1) → (1 1 1 1 1 3 3 4 6 6 8 8)`	
`(10 2 105 −78 46 45 23) → (−78 2 10 23 45 46 105)`	

Incremental specification

1.

remove-elem: *removes the first instance of a given element from a list.*	
— Training examples —	
`6, (6) → ()`	[15 steps]
`7, (8 7) → (8)`	[30 steps]
`3, (3 3) → (3)`	[30 steps]
`10, (2 4 10 7 2 1) → (2 4 7 2 1)`	[200 steps]
— Test examples —	
`43, (9 56 43 2 7) → (9 56 2 7)`	
`8, (6 8 4 8 2 8) → (6 4 8 2 8)`	
`9, (7 5 2 9) → (7 5 2)`	

2.

min: *returns the smaller of two integers.*	
— Training examples —	
`2, 1 → 1`	[10 steps]
`3, 10 → 3`	[10 steps]
`7, 7 → 7`	[10 steps]
— Test examples —	
`−5, −10 → −10`	
`−3, 20 → −3`	
`27, 27 → 27`	

3.

least-elem: *returns the smallest element in a list of integers.*	
— Training examples —	
`(3) → 3`	[20 steps]
`(8 4 7) → 4`	[100 steps]
`(9 6 2 9 2) → 2`	[200 steps]
— Test examples —	
`(10 7 45 5 7 8) → 5`	
`(77 34 59 34 208) → 34`	

4.

extract-least-elem: *brings the smallest element to front of a list of integers.*	
— Training examples —	
`(8) → (8)`	[50 steps]
`(10 4) → (4 10)`	[200 steps]
`(8 6 2 7 2 5) → (2 8 6 7 2 5)`	[2000 steps]
— Test examples —	
`(3 2 1 2 3) → (1 3 2 2 3)`	
`(54 70 14 59 14 20) → (14 54 70 59 14 20)`	

5. **sort**

Results

Stage	Time / s	Depth	Solution
remove-elem	7.8	14	(λ (a1 a2) (paralist (λ (a3 a4 a5) (if (eql a3 a1) a4 (cons a3 a5))) a2 a2))
min	0.0	7	(λ (a1 a2) (if (< a2 a1) a2 a1))
least-elem	0.7	13	(λ (a1) (paralist (λ (a2 a3 a4) (min a4 a2)) a1 (car a1)))
extract-least-elem	3.3	14	(λ (a1) (cons (least-elem a1) (remove-elem (least-elem a1) a1)))
sort	4.5	14	(λ (a1) (analist (λ (a2) (paralist (λ (a3 a4 a5) (extract-least-elem a5)) a2 a2)) a1))
Total	**16.3**		

Non-incremental: TIMEOUT (586.6 seconds, depth 19)

Figure 4. The `sort` problem sequence: specification and results.

failure occurred at the testing stage, we added new training examples and re-ran the experiment. For the final specifications given in the figures, every solution program has passed all of its test examples.

We recorded the times taken for MagicLisper to solve the stages of each sequence. Total times was determined by adding these values together. Following each sub-problem in a sequence, the inferred solution program was added to the library of primitives and assigned a weight of 2.5, 2.5, 3.5, or 3.0 in the case of problem sequences `deref-list`, `reverse`, `sort`, and `block-lengths` respectively. The library of primitives was reset to its default state between problem sequences. We also tested how our system fared when solving each main problem on its own with the default primitive library, i.e. without incremental learning. We allowed at least 500 seconds for every problem; if this time limit was exceeded then the computation was aborted after allowing for the current search iteration to finish, and 'TIMEOUT' was indicated in the results table. Also given in each results table are the search depths, in units of program weight, at which any solution was found or a timeout occurred, as well as the solution programs themselves. The experiments were performed on a 2 GHz Intel Core II Duo desktop PC with 2 GB of RAM running GNU CLISP.

block-lengths: *replaces all blocks of consecutive identical elements in a list with their lengths.*	
— Training examples —	
`() → ()`	[50 steps]
`(8) → (1)`	[200 steps]
`(7 6) → (1 1)`	[700 steps]
`(8 8 3 4) → (2 1 1)`	[5000 steps]
`(6 5 5 4) → (1 2 1)`	[5000 steps]
— Test examples —	
`(7 7 7 7 7 5 5 5 7 7 2 2 4 9 9 9) → (5 3 2 2 1 3)`	
`(5 8 8 4 9 1 2 1 2) → (1 2 1 1 1 1 1 1)`	
`(0 0 0 0 0 7 0 0 0 5 5 5 5 5 5) → (5 1 3 6)`	

Incremental specification

1.

car-p: *tests whether an object is the first element of a list.*	
— Training examples —	
`0, () → f`	[15 steps]
`1, () → f`	[15 steps]
`4, (4) → t`	[15 steps]
`5, (2) → f`	[15 steps]
`8, (8 2) → t`	[15 steps]
`7, (6 2 7) → f`	[15 steps]
— Test examples —	
`7, (8 7 7 6 4 7) → f`	
`3, (3 8 1 4) → t`	

2.

remove-first-block: *removes the first block of consecutive identical elements from a list.*	
— Training examples —	
`(8) → ()`	[30 steps]
`(4 6) → (6)`	[100 steps]
`(1 3 1 3) → (3 1 3)`	[400 steps]
`(9 9 8 6 9 3) → (8 6 9 3)`	[1000 steps]
`(5 5 5 5 4 9) → (4 9)`	[1000 steps]
— Test examples —	
`(7 7 7 7 4 4 3 3 7 8 8 7 2 2) → (4 4 3 3 7 8 8 7 2 2)`	
`(1 6 5 1 2 2 2) → (6 5 1 2 2 2)`	
`(9 9 9 9) → ()`	

3.

length: *finds the length of a list.*	
— Training examples —	
`() → 0`	[10 steps]
`(8) → 1`	[20 steps]
`(10 4 7 2) → 4`	[50 steps]
— Test examples —	
`(8 4 7 3 2 9 1 1 2) → 9`	
`(92 −8 7 83 24) → 5`	

4.

length-first-block: *finds the length of the first block of consecutive identical elements in a list.*	
— Training examples —	
`(8) → 1`	[50 steps]
`(4 6) → 1`	[100 steps]
`(9 9 8 6 9 3) → 2`	[1000 steps]
— Test examples —	
`(3 3 3 3 8 7 6 3 4 5) → 4`	
`(5 5 5 5 5 5 5 2 2) → 7`	

5.

convert-first-block-to-length: *replaces the first block of consecutive identical elements in a list with its length.*	
— Training examples —	
`() → ()`	[20 steps]
`(8) → (1)`	[100 steps]
`(7 6) → (1 6)`	[400 steps]
`(8 8 3 4) → (2 3 4)`	[2000 steps]
`(5 5 5 3) → (3 3)`	[2000 steps]
— Test examples —	
`(8 8 8 6 6 6 6) → (3 6 6 6 6)`	
`(4 1 5 4 2 2) → (1 1 5 4 2 2)`	

6. **block-lengths**

Results

Stage	Time / s	Depth	Solution
car-p	0.4	11	`(λ (a1 a2) (paralist (λ (a3 a4 a5) (eql a3 a1)) a2 f))`
remove-first-block	0.5	12	`(λ (a1) (paralist (λ (a2 a3 a4) (if (car-p a2 a3) a4 a3)) a1 a1))`
length	0.2	12	`(λ (a1) (paralist (λ (a2 a3 a4) (inc a4)) a1 0))`
length-first-block	6.9	15	`(λ (a1) (length (paralist (λ (a2 a3 a4) (cdr a4)) (remove-first-block a1) a1)))`
convert-first-block-to-length	4.8	14	`(λ (a1) (paralist (λ (a2 a3 a4) (cons (length-first-block a1) (remove-first-block a1))) a1 a1))`
block-lengths	0.1	9	`(λ (a1) (analist (λ (a2) (convert-first-block-to-length a2)) a1))`
Total	**12.9**		

Non-incremental: TIMEOUT (561.8 seconds, depth 19)

Figure 5. The `block-lengths` problem sequence: specification and results.

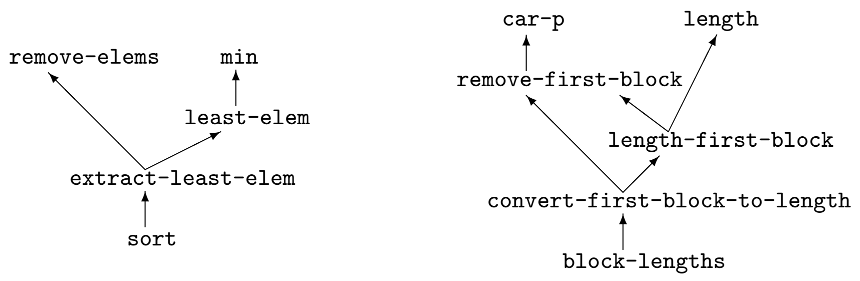

Figure 6. Dependency graphs for the solutions to the `sort` and `block-lengths` problem sequences. Each arrow $x \longrightarrow y$ means 'program x invokes program y'.

5.2 Analysis

The total times taken for our system to solve the incremental specifications ranged between 1 and 16 seconds. On the other hand, all of the non-incremental scenarios took more than 500 seconds, with a solution only being found at all in the case of `reverse`. This amounts to an increase in speed due to incremental learning of at least a factor of thirty in every case.

On inspection of the solution programs for the incremental sequences, we see that the majority of programs do indeed invoke earlier solutions, as expected. Indeed, for the longer sequences `sort` and `block-lengths` we can visualise a graph of dependencies between the solution programs (figure 6).

The timing results indicate that, at worst, incremental learning can greatly improve the performance of IP, while, at best, it is able to make otherwise intractable problems tractable. To see why this should be the case, consider the following computational complexity argument. Assuming that it takes constant time to generate and test each program, then the time taken for MagicLisper's search algorithm to solve a given problem will be proportional to the total number of programs generated. We expect this to be approximately $\mathrm{O}[b^d]$, where b is the search branching factor, roughly proportional to the size of the primitive library, and d is the search depth of the lowest-weight solution program that exists for the problem. Now, if we make an assumption that with incremental learning we can always divide a problem into sub-problems whose solution depths are bounded by a constant d_0, then the time taken to solve the problem in incremental stages is no more than $\mathrm{O}[n(b_0+n\Delta b)^{d_0}]$, where n is the number of stages, b_0 is the branching factor of the default primitive library and Δb is the increase in the branching factor that occurs each time we add a new primitive. Let us also assume that the number of stages required to satisfactorily break down a problem is roughly proportional to the depth of the lowest-weight solution program that we'd get if the problem were solved non-incrementally, in other words, $n = kd$. This gives us a time taken of $\mathrm{O}[kd(b_0 + kd\Delta b)^{d_0}]$, or simply $\mathrm{O}[d^{d_0+1}]$ with respect to d, if the problem is solved incrementally, compared with $\mathrm{O}[{b_0}^d]$ if it is solved non-incrementally. In this way, IP with incremental learning can allow a system solve, in polynomial time, problems that take exponential time with non-incremental IP.

6. Limitations and further work

The main contribution of this paper has been to demonstrate a simple, working methodology for incremental learning in IP. This methodology involved equipping a brute-force search based IP system with an ability to reuse solution programs by adding them to its primitive library. We showed that this mechanism can be effective by demonstrating its use on four problems, each of which had been broken down into an appropriate sequence of sub-problems. Our IP system was able to solve the problems orders of magnitude more quickly when making using of the incremental sequences than when simply solving the main problems in isolation.

In this section we address the limitations of our simple incremental learning methodology; in particular we talk about the difficulties involved in constructing problem sequences. We consider how to overcome these limitations, and discuss how, by eliminating the need for problem sequences to be designed by a human expert, we aim to enable a much more useful, autonomous form of incremental learning.

6.1 Limitations of the simple methodology

The main drawback of the simple incremental learning methodology presented in this paper is the significant amount of human effort and expertise required to design effective problem sequences. Based on our experience designing problem sequences for MagicLisper, we feel that the need for this effort and expertise is largely due to what we shall call 'brittleness' in the system's learning mechanism. In other words, problem specifications must obey certain conditions in order for learning to work, and they 'break easily', i.e. if these conditions are not met perfectly, then the system will fail to find a solution at all.

One source of brittleness in our mechanism is the fact that solutions to sub-problems are only useful if a solution to the main problem can be expressed in terms of them directly. It is not enough for a sub-problem simply to be related to the main problem, for example if their solutions would share some common structure. In consequence, the success or failure of incremental learning is very sensitive to the exact choice and order of sub-problems. Often, the only way to predict if a particular sub-problem will be effective is to use ones knowledge of how one might implement the target program by hand; in other words, using the IP system does not save one much effort over hand-coding the program. Our methodology suffers from brittleness in two other ways too. Firstly, the IP system will not tolerate any error or noise in the training examples. Secondly, if any of the step counts associated with the training examples are too low, the system will again completely fail to find a solution.

6.2 Overcoming the limitations

Our simple methodology seems capable of scaling up to relatively complex problems, but at the cost of a degree of human effort expended in designing problem sequences at least as great as would be required to code the solutions by hand. In this subsection we discuss how to eliminate the three sources of 'brittleness' described in the last subsection, with the aim of developing a mechanism that is flexible enough to perform incremental learning over loosely constructed sets of problems, rather than precisely constructed sequences.

The need to specifiy step counts with training examples should be the easiest limitation to overcome. In MagicHaskeller, it is already unnecessary to specify step counts, because the system simply tests all programs until termination, relying on the fact the primitive library belies the possibility of infinite loops. However, we don't expect this approach to remain feasible as we start to generate more complex programs, because the number of programs that run for a long time before termination will become much larger. Instead, we propose using an algorithm like 'Levin search' (Schmidhuber 2004), in which the iterative deepening nature of our IP search is extended so as to automatically re-test programs for longer and longer step counts as the search progresses.

The need for a solution to a main problem to be expressible directly in terms of solutions to sub-problems could be overcome as follows. Suppose that we modify our incremental learning mechanism such that, instead of adding actual solution programs to the primitive library, it attempts to derive re-usable procedural abstractions from groups of solution programs, and then adds these abstractions as the new primitives. The potential re-usabilility of a procedural abstraction can be measured objectively using a principle of 'minimum description length': if a procedure, when reused in multiple solution programs, serves to reduce the combined size of these programs by more than its own size, then we can deem it a useful abstraction. Though the best way to discover candidate abstractions is an open question, it would seem a reasonable starting point to try a brute-force search. This method of incremental learning would be much more adaptable and generic than our original mechanism, in that it should be able to extract useful inductive bias from almost any kind of shared structure or commonality between solution programs. We know of at least one previous implementation of a similar idea: the 'Duce' system (Muggleton 1987) can discover abstractions that encapsulate shared structure among groups of statements in propositional logic.

To overcome the lack of toleration of errors or noise in the training examples, we feel that the most satisfactory solution will ultimately be to reformulate our IP methodology within a probabilistic framework. In such a framework, a program would no longer describe a deterministic mapping from inputs to outputs, rather it would represent a conditional probability distribution over the set of possible outputs given the inputs. Such a reformulation is highly desirable if our aim is to develop a machine learning technique of practical use, since real-world data is usually noisy. Indeed, the development of probabilistic frameworks for IP is an active area of research, particularly within inductive logic programming (De Raedt and Kersting 2004).

In overcoming the above limitations, our eventual goal is to produce an IP methodology capable of performing incremental learning simply from a corpus of data, without the need for that data to be organised into problem sequences by a human expert. To see how this might work, first consider how a system could perform incremental learning if provided with a large bank of related problems of various difficulties, in no particular order. Such a system could repeatedly scan through the problems, briefly attempting to solve each as it goes. Some of the problems might be easy enough to solve immediately, and the system could then use the solutions of these to derive procedural abstractions which it would add to its primitive library. On the next scan through the problem bank, these new primitives should enable the system to solve some problems that were previously out of its reach. Ideally, the process iterates until most of the problems are solved. Consider next how one might extend this idea in order to create a system capable of automonomously learning a model for a complex environment or corpus of data. In a such a situation, it might often be the case that various parts of the environment or corpus can be described by simple models. By analogy with the 'bank of problems' scenario, one may imagine an incremental learning system that initially looks for these simple models, adds abstractions derived from those models to its background knowledge, then searches for more complex models, and so on. We may think of this process as an automation of the scientific method.

7. Conclusion

In this paper, we have demonstrated a simple but effective incremental learning mechanism for an inductive programming system. It works by having the system incorporate solution programs into its object language as new primitive functions as it progresses through a sequence of problems. The mechanism is capable of producing orders of magnitude improvements in problem solving performance, but at the expense of considerable human effort spent in designing appropriate problem sequences. However, we have sketched a number of possible improvements to the mechanism which should reduce or remove much of the need for this human guidance. Our aim is that this methodology can eventually be developed into a powerful generic machine learning technique by which a system can learn a model of a large, complex dataset in an autonomous fashion.

Acknowledgments

Thank you to my MSc supervisor Michael O'Boyle, for his support and encouragement on this project.

References

L. Augusteijn. Sorting morphisms. In *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1998.

L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In *ALT*, volume 3244 of *LNCS*, pages 19–36. Springer, 2004.

M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence (AGI-09)*, 2009.

S. Katayama. Systematic search for lambda expressions. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007.

K. Khan, S. Muggleton, and R. Parson. Repeat learning using predicate invention. In *Inductive Logic Programming*, volume 1446 of *LNCS*, pages 165–174. Springer, 1998.

E. Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

S. Muggleton. Duce, an oracle based approach to constructive induction. In *IJCAI-87*, pages 287–292, 1987.

J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55 – 83, 1995.

J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proceedings of the 6th European Conference on Machine Learning*, LNCS, pages 3–20. Springer-Verlag, 1993.

J. Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

R. J. Solomonoff. Progress in incremental machine learning. Given at *NIPS Workshop on Universal Learning Algorithms and Optimal Search, Dec. 14, 2002, Whistler, B.C., Canada.*, 2002. URL `http://world.std.com/~rjs/pubs.html`.

Enumerating Well-Typed Terms Generically

Alexey Rodriguez Yakushev[1] Johan Jeuring[2,3]

[1]Vector Fabrics B.V., Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
[2]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[3]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

alexey.rodriguez@gmail.com johanj@cs.uu.nl

Abstract

We use generic programming techniques to generate well-typed lambda terms. We encode well-typed terms by generalized algebraic datatypes (GADTs) and existential types. The Spine approach (Hinze et al. 2006; Hinze and Löh 2006) to generic programming supports GADTs, but it does not support the definition of generic producers for existentials. We describe how to extend the Spine approach to support existentials and we use the improved Spine to define a generic enumeration function. We show that the enumeration function can be used to generate the terms of simply typed lambda calculus.

1. Introduction

This paper discusses the problem of given a type, generate lambda terms of that type. There exist several algorithms and/or tools for producing lambda terms given a type (Augustsson 2005; Katayama 2005; Koopman and Plasmeijer 2007). The approach discussed in this paper uses generic programming techniques on Generalized Algebraic Datatypes (GADTs) and existentials to enumerate well-typed lambda terms. The enumeration function is much simpler than previous work, and the main problem lies in making generic programming techniques available for GADTs and existentials in functions that produce values of a particular datatype, such as an enumeration function.

Since their introduction to Haskell, Generalized Algebraic Datatypes (GADTs) (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) are often used to improve the reliability of programs. GADTs encode datatype invariants by type constraints in constructor signatures. With this information, the compiler rejects values for which such invariants do not hold during type-checking. In particular, GADTs can be used to model sets of well-typed terms such that values representing ill-typed terms cannot be constructed. Other applications of GADTs include well-typed program transformations, implementation of dynamic typing, staged computation, ad-hoc polymorphism and tag-less interpreters.

Given the growing relevance of GADTs, it is important to provide generic programming support for generalized datatype definitions. The generation of datatype values using generic programming is of particular interest. Generic value generation has been used before to produce test data, which can be used to check the validity of program properties (Koopman et al. 2003). In generic value generation, the datatype definition acts as a specification for test data. However, this specification is often imprecise, since it gives rise to either values that do not occur in practice, or, worse, ill-formed values (for example, a program fragment with unbound variables). For this reason, QuickCheck (Claessen and Hughes 2000) allows the definition of custom generators.

GADT definitions may specify types more precisely than normal datatypes. In the case of well-typed terms, the constraints in the datatype definition describe the formation rules of a well-typed value. It follows that a generic producer function on a GADT might produce values that are better suited for testing program properties. For example, it should be possible to use a generic value generation function with a GADT encoding lambda calculus, in order to produce a well-typed lambda term with which a tag-less interpreter can be tested.

The *spine* view (Hinze et al. 2006; Hinze and Löh 2006), which is based on "Scrap Your Boilerplate" (Lämmel and Peyton Jones 2003), is the only approach to generic programming in Haskell that supports GADTs. The main idea behind the spine view is to make the application of a data constructor to its arguments explicit. The spine view represents a datatype value by means of two cases: the representation of a datatype constructor, and the representation of the application to constructor arguments. A generic function can then be defined by case analysis on the spine view. Hinze and Löh (2006) describe how to use the spine view to represent GADT values and define generic functions to consume and produce such values.

Besides GADT definitions, our definition of well-typed terms uses existentially quantified type variables. In particular, the type of expression application is that of the function return type. The argument type is hidden from the application type and is therefore existentially quantified. Under certain conditions, the spine view supports the definition of generic functions that *consume* existentially typed values. Unfortunately, it cannot be used to define a generic function that *produces* them. It follows that the spine view cannot in general be used to define generic enumerators for well-typed terms.

This paper extends the spine view to allow the use of producer functions on existential types. We make the following contributions:

- We show how to support existential types systematically within the spine view. We extend the spine view to encode existentially quantified type variables explicitly. This enables the definition of generic functions that perform case analysis on such types. As a consequence, the extended spine view supports the definition of generic producers that work on existential types. We demonstrate the increased generality by defining generic serialization and deserialization for existential types and GADTs.
- We define a generic enumeration function that can be used with GADTs and existential types. This function can be used to enumerate the well-typed terms represented by a GADT. Consider a GADT that represents terms in the simply typed lambda calculus. The enumeration of terms with type $\mathsf{Expr}\ ((\mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c})$ yields the term that corresponds to function

composition. The enumeration function requires explicit support for existential types in producers. For that reason it cannot be defined in approaches such as that of Hinze and Löh (2006).

This paper is organized as follows. Section 2 introduces the spine view and gives several examples illustrating why this view is not suitable to define producers for existential types. Section 3 describes our extensions to the spine view, which enable producer support for existential types. Section 4 uses the extended spine view to define a generic enumeration function. The enumeration function is then used to produce well-typed lambda calculus terms. Section 5 discusses related work, and section 6 concludes.

2. The spine view

The spine view was introduced by Hinze et al. (2006). This view supports the definition of generic functions that consume (such as *show* or *eq*) or transform (such as *map*) datatype values. We introduce the spine view using the generic show function as an example. This function prints the textual representation of a value based on the type structure encoded by the view. To implement this function, we need case analysis on types to implement type-dependent behavior.

2.1 Case analysis on types

The spine view uses GADTs to implement case analysis on types. We define a type representation datatype where each constructor represents a specific type:

```
data Type :: ∗ → ∗ where
  Int     :: Type Int
  Maybe   :: Type a → Type (Maybe a)
  Either  :: Type a → Type b → Type (Either a b)
  List    :: Type a → Type [a]
  ( :→ )  :: Type a → Type b → Type (a → b)
```

An overloaded function can be implemented by performing case analysis on types. To perform case analysis on types we pattern match on the type representation values. The GADT pattern matching semantics (Peyton Jones et al. 2006) ensures that the type variable a is refined to the target type of the matched constructor:

```
show :: Type a → a → String
show Int          n         = showInt n
show (Maybe a)    (Just x)  = paren ("Just"  • show a x)
show (Maybe a)    Nothing   = "Nothing"
show (Either a b) (Left x)  = paren ("Left"  • show a x)
show (Either a b) (Right y) = paren ("Right" • show b y)
show (List a)     ((:) x xs) = paren ("(:)"
                               • show a x
                               • show (List a) xs)
show (List a)     []        = "[]"
```

This function prints a textual representation of a datatype value. Note that we choose to print lists in prefix syntax rather than the usual Haskell notation. The operator (•) separates two strings with a white space, and *paren* prints parentheses around a string argument.

2.2 The spine representation of values

Type representations can be used to implement overloaded functions, but such functions are not generic. The user needs to define new *show* cases for every datatype added to the program. To define generic functions, we make use of the spine view.

The spine view represents all datatype values by means of two cases: a constructor and the application of a (partially applied) constructor to an argument. This is embodied in the Spine datatype:

```
data Spine :: ∗ → ∗ where
  Con   :: ConInfo a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
infixl 0 :◇:
```

The *Con* case of the spine view stores a value constructor of type a together with additional information including the constructor name, the fixity, and the constructor tag. This additional information is stored in the datatype ConInfo. The application case (:◇:) consists of a functional value Spine that consumes a-values, and the argument a paired with its type representation in the datatype Typed. We show Typed, and a simplified ConInfo containing only the constructor name below:

```
data ConInfo a = ConInfo { conName :: String, conVal :: a }
data Typed a   = (:▷:)   { val :: a, rep :: Type a }
```

To write a generic function, we first convert a value to its Spine representation. We show how to perform this conversion using the type-indexed function *toSpine*:

```
toSpine :: Type a → a → Spine a
toSpine Int          x          = Con (conint x x)
toSpine (Maybe a)    (Just x)   = Con (conjust Just) :◇: x :▷: a
toSpine (Maybe a)    Nothing    = Con (connothing Nothing)
toSpine (Either a b) (Left x)   = Con (conleft Left) :◇: x :▷: a
toSpine (Either a b) (Right y)  = Con (conright Right) :◇: y :▷: b
toSpine (List a)     ((:) x xs) = Con (concons (:))
                                    :◇: x :▷: a :◇: xs :▷: List a
toSpine (List a)     []         = Con (connil [])
```

Because we reuse the constructor information in later sections of the paper, we define ConInfo values separately. We give some examples below:

```
conint :: Int → a → ConInfo a
conint i     = ConInfo (showInt i)

connothing, conjust :: a → ConInfo a
connothing = ConInfo "Nothing"
conjust    = ConInfo "Just"
```

In summary, to enable generic programming using the spine view, we define a GADT for type representations, the Spine datatype, and conversions from datatype values to their spine representations. The conversions for datatypes are written only once, and then the same conversion can be reused for different generic functions. The conversion to the spine representation is regular enough that it can be automatically generated from the syntax trees of datatype declarations[1]

Equipped with the spine representation, we can write a number of generic functions. For example, this is the definition of generic *show*:

```
show :: Type a → a → String
show rep x = paren (gshow (toSpine rep x))

gshow :: Spine a → String
gshow (Con con)     = conName con
gshow (con :◇: arg) = gshow con • show (rep arg) (val arg)
```

This function is a simplified variant of the *show* function defined in the Haskell prelude. All datatype values are printed uniformly: constructors are separated from the arguments by means of the • operator, and parentheses are printed around fully applied constructors.

[1] At the time of writing, Template Haskell cannot handle GADT declarations. Our prototype generates the spine representation for a GADT using a manually constructed declaration syntax tree instead of parsing the GADT declaration and processing it via Template Haskell.

2.3 Transformer functions

The spine view also supports the definition of generic transformer functions. Examples of such functions include incrementing all Int values in a tree, and applying a function to all nodes of a specific type in a tree.

To write such a function, we need to convert the spine representation back to the represented value *after* it has been traversed and transformed. This is achieved by the *fromSpine* function:

```
fromSpine :: Spine a → a
fromSpine (Con con)     = conVal con
fromSpine (con :◇: arg) = fromSpine con (val arg)
```

See Hinze et al. (2006) for examples of transformer functions using the spine view.

2.4 A view for producers

It is impossible to write *read*, the inverse to *show* using the current Spine datatype. We could for example use the following type for *read*:

```
read :: Type a → String → [(a, String)]
```

This function produces all possible parses of type a (paired with unused input) from a representation for the type a and an input string. To write such a generic function, we would need a spine representation to guide the parsing process. Unfortunately, a representation Spine a cannot be used for this purpose. A value of Spine a represents a particular value of type a (for example, a singleton list) rather than the full datatype structure (a description of the cons and nil constructors and their arguments). To enable a generic definition of generic read and other producer generic functions, Hinze and Löh (2006) introduce the type spine view. This view describes all values of a rather than a particular one.

```
type TypeSpine a = [Signature a]
data Signature :: ∗ → ∗ where
  Sig   :: ConInfo a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
infixl 0 :⊕:
```

Here we again have two cases, one for encoding a constructor and another for the application of a (partially) applied constructor to an argument. The application case contains only a type representation and no argument value anymore. A value of TypeSpine a is a list of constructor signatures representing all constructors of the represented datatype. The type-indexed function *typeSpine* produces the type spine representations of all datatypes on which generic programming is to be used.

```
typeSpine :: Type a → TypeSpine a
typeSpine Int          = [Sig (conint i i)
                          | i ← [minBound . . maxBound]]
typeSpine (Maybe a)    = [Sig (connothing Nothing)
                          , Sig (conjust Just) :⊕: a]
typeSpine (Either a b) = [Sig (conleft Left)  :⊕: a
                          , Sig (conright Right) :⊕: b]
typeSpine (List a)     = [Sig (connil [ ])
                          , Sig (concons (:)) :⊕: a :⊕: List a]
```

The generic parsing function, *read*, builds a parser that deserializes a value of type a:

```
read :: Type a → Parser a
```

For the purposes of this paper, we assume that Parser is an abstract parser type with a monadic interface, with some standard derived functions:

```
return       :: a → Parser a
(≫=)         :: Parser a → (a → Parser b) → Parser b
ap           :: Parser (a → b) → Parser a → Parser b
(≫)          :: Parser a → Parser b → Parser b
noparse      :: Parser a
alternatives :: [Parser a] → Parser a
readInt      :: Parser Int
lex          :: Parser String
token        :: String → Parser ()
readParen    :: Parser a → Parser a
```

The definition of generic read uses *readInt* to read an integer value. For other datatypes, we make parsers for each of the constructor representations and merge all the alternatives in a single parser.

```
read :: Type a → Parser a
read Int = readInt
read rep = alternatives [readParen (gread conrep)
                         | conrep ← typeSpine rep]
```

The generic parser of a constructor is built by induction on its signature representation. The base case (*Sig*) tries to recognize the constructor name and returns the constructor value. The application case parses the function and argument parts recursively and the results are combined using monadic application:

```
gread :: Signature a → Parser a
gread (Sig c)       = token (conName c) ≫ return (conVal c)
gread (con :⊕: arg) = gread con ‘ap‘ read arg
```

In the definition of generic read, we could also have used parser combinators based on an applicative interface (McBride and Paterson 2007) instead of a monadic one. For an example of parser combinators with an applicative interface see Swierstra and Duponcheel (1996). In Section 3.1 we show that existentially typed values cannot be parsed using purely applicative parser combinators, because generic read on existentials makes essential use of bind (≫=).

2.5 Generalized algebraic datatypes

Recall that generalized algebraic datatypes are datatypes to which type-level constraints are added. Such constraints can be used to encode invariants that datatype values must satisfy. For example, we can define a well-typed abstract syntax tree by having the syntactic categories of constructs in the target type of constructors:

```
data Expr :: ∗ → ∗ where
  EZero   :: Expr Int
  EFalse  :: Expr Bool
  ESuc    :: Expr Int  → Expr Int
  ENot    :: Expr Bool → Expr Bool
  EIsZero :: Expr Int  → Expr Bool
```

We have constants for integer and boolean values, and operators that act on them.

GADTs can easily be represented in the spine view. For instance, the definition of *toSpine* for this datatype is as follows:

```
toSpine :: Type a → a → Spine a
. . .
toSpine (Expr Int)  EZero       = Con (conezero    EZero)
toSpine (Expr Bool) EFalse      = Con (conefalse   EFalse)
toSpine (Expr Int)  (ESuc e)    = Con (conesuc     ESuc)
                                  :◇: e :▷: Expr Int
toSpine (Expr Bool) (ENot e)    = Con (conenot     ENot)
                                  :◇: e :▷: Expr Bool
toSpine (Expr Bool) (EIsZero e) = Con (coneiszero EIsZero)
                                  :◇: e :▷: Expr Int
```

This definition requires the extension of Type with the representation constructors *Expr* and *Bool*. Generic functions defined on the spine view, such as generic show, can now be used on Expr.

What about generic producer functions? These can be used on Expr too, because it is also possible to construct datatype representations for GADTs in the type spine view:

```
typeSpine :: Type a → TypeSpine a
...
typeSpine (Expr Int)  = [Sig (conezero    EZero)
                        ,Sig (conesuc     ESuc) :⊕: Expr Int]
typeSpine (Expr Bool) = [Sig (conefalse   EFalse)
                        ,Sig (conenot     ENot) :⊕: Expr Bool
                        ,Sig (coneiszero EIsZero) :⊕: Expr Int]
```

To parse boolean expressions, we invoke the generic read function as follows:

```
readBoolExpr :: Parser (Expr Bool)
readBoolExpr = read (Expr Bool)
```

The parser for integer expressions would use a different argument for *Expr*. In this example, we are assuming that the expression to be parsed is always of a fixed type. A more interesting scenario would be to leave the type of the GADT unspecified and let it be dynamically determined from the parsed value. This would be useful if the programmer wants to parse some well-typed expression regardless of the type that the expression has.

A possible solution to parsing a GADT without specifying its type argument would be to existentially quantify over that argument in the result of the parsing function. Next, we discuss how the spine view deals with existential types.

2.6 Existential types and consumer functions

In Haskell, existential types are introduced in constructor declarations. A type variable is existentially quantified if it is mentioned in the argument type declarations but omitted in the target type. For example, consider dynamically typed values:

```
data Dynamic :: * where
  DynVal :: Type a → a → Dynamic
```

The type variable a in the declaration is existentially quantified. It is used to hide the type of the a-argument used when building a Dynamic value. The type a is kept abstract when pattern matching a Dynamic value, but by case analyzing the type representation it is possible to dynamically recover the type a. Thus, statically, Dynamic values all have the same type, but, dynamically, the type distinction can be recovered and acted upon.

To represent dynamic values in Spine, we add type representations for Type itself and Dynamic. Hence, we add the following two constructors to Type:

```
data Type :: * → * where
  ...
  Type    :: Type a → Type (Type a)
  Dynamic :: Type Dynamic
```

Now, Dynamic values may be represented as follows by the spine view:

```
toSpine :: Type a → a → Spine a
...
toSpine Dynamic (DynVal rep val) = Con (condynval DynVal)
                                   :◇: rep :▷: Type rep
                                   :◇: val :▷: rep
```

While Dynamic values may be easily represented, this is not the case for all datatypes having existential types. Recall that in a spine representation, every constructor argument is paired with its type representation in the datatype Typed. In general, in constructors having existential types, it may not be possible to build such a pair because the representation of the existential type may be missing. The constructor *DynVal* is a special case, because it carries the representation type of the existential a. For an example where the representation of a constructor with existential types is not possible, consider adding an application constructor to the expression datatype:

```
data Expr :: * → * where
  ...
  EApp :: Expr (a → b) → Expr a → Expr b
```

and consider the corresponding *toSpine* alternative:

```
toSpine :: Type a → a → Spine a
...
toSpine (Expr b) (EApp fun arg) = Con (coneapp EApp)
                                  :◇: fun :▷: Expr (a :→ b)
                                  :◇: arg :▷: Expr a
```

This code is incorrect due to the unbound variable *a* which stands for the existential representation. The conclusion here is that the spine view can be used on an existential type, as long as the constructor in which it occurs carries a type representation for it.

2.7 Existential types and producer functions

The view for producer functions, the type spine view, cannot represent existential types equally well as the spine view. For instance, consider how to generate such a representation for dynamic values:

```
typeSpine :: Type a → [Signature a]
...
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a]
```

What should the representation *a* be? There are two options, we either fix it to a single type representation or we range over all possible type representations. Choosing one type representation would be too restrictive, because *read* would only parse dynamic values of that type and fail on any other type. We try the second option:

```
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a
                    | a ← types]
```

This code does not yet have the behavior we desire. For *typeSpine* to be type-correct, *types* must return a list of representations all having the same type. Because Type is a singleton type (each type has only one value), *types* returns a single type representation. We would like *types* to generate a list of all possible type representations, but different type representations have different types. Therefore, *types* should return representations whose represented type is existentially quantified. To this end, we define the type of boxed type representations:

```
data BType = ∀a.Boxed (Type a)
applyBType :: (∀a.Type a → c) → BType → c
applyBType f (Boxed a) = f a
```

Now we can define the type spine of dynamic values, for which we assume a list of boxed representations (*types*):

```
types :: [BType]
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a
                    | Boxed a ← types]
```

The boxed representations are used to construct a list of constructor signatures that represent a dynamic value of the corresponding type. There are infinitely many type instances of polymorphic types, therefore there are infinitely many Dynamic constructor representations. An infinite type spine is not a desirable representation to work with. The *read* function would try to parse the input using every Dynamic constructor representation. If there is a correct parse, parsing would eventually succeed with one of the representations. However, if there is no correct parse, parsing would not terminate. Moreover, this representation precludes implementing more efficient variants of parsing.

Infinite type spine representations for datatypes with existentials make the use of generic producers on such datatypes unpractical. Before describing a modified type spine view that solves this problem, we explore a couple of non-generic examples to motivate our design decisions.

We start with the parser definition for Dynamic values. In the code above, we are able to parse any possible dynamic value because there are *DynVal* constructor signatures for all possible types. For each signature, we build a parser that parses the corresponding type representation and a value having that type.

Now, rather than parsing the two arguments of the constructor *DynVal* independently, we introduce a dependency on representations. First, we parse the type representation for the existential. Then, we use it to build a parser of the corresponding type and parse the second argument. In this way, we no longer need to have an infinite representation of types because we obtain the representation of the existential during the parsing process:

```
read :: Type a → Parser a
...
read Dynamic = do
  Boxed a ← readType
  value   ← read a
  return (DynVal a value)
```

To this end, we use a function that parses type representations. Because the result may be of an arbitrary type, *readType* produces a representation that is boxed:

```
readType :: Parser BType
```

We defer the presentation of *readType* to Section 3.4.

The same technique can be used to parse any constructor having an existential type. For example, the definition for parsing expression applications is as follows:

```
read (Expr b) = do
  Boxed a ← readType
  fun     ← read (Expr (a :→ b))
  arg     ← read (Expr a)
  return (EApp fun arg)
```

In this example, the type representation that is parsed is used to build the type representations for the two remaining arguments.

These two examples show that constructors with existential types must be handled differently from other constructors. In such constructors, the constructor argument representations depend on the type representation of the existential type. In our examples, this dependency is witnessed by the dynamic construction of parsers based on the type representation that was previously parsed.

3. An improved Spine view: support for existential types

We start this section by showing how to extend the spine view for producers to represent existential types explicitly. Then, we show why this extension is also necessary for the consumer spine view.

3.1 The existential case for producer functions

We have learned two things from the *read* examples for constructors with existential types. First, we need a way to represent existential variables explicitly, so that generic functions can handle existential type variables specifically. And second, there is a dependency from constructor arguments on the existential variable. For example, we can only parse the function and argument parts of an expression application, if we have already parsed the existential type representation. We modify the type spine view to accommodate these two aspects. We extend the constructor signatures in this view with a constructor to represent existential quantification: *AllEx*. The dependency of type b on an existential type a is made explicit by means of a function from type representations of type a to representations of b.

```
data Signature :: ∗ → ∗ where
  Sig   :: a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
  AllEx :: (∀a.Type a → Signature b) → Signature b
```

Interestingly, the type variable a is universally rather than existentially quantified. Why is this the case? The type spine view represents all possible values of a datatype, therefore the existential variable must range over all possible types. This also explains the name of the constructor *AllEx*, which stands for all existential type representations.

There is another modification to the type spine view. The *Sig* constructor no longer carries constructor information. Instead, this information is stored at the top-level of the representation:

```
type TypeSpine a = [ConInfo (Signature a)]
```

This change is not strictly necessary but it is convenient. Suppose that the constructor information is still stored in *Sig*. Now, applications that need to perform a pre-processing pass using constructor information (for example, for more efficient parsing) would be forced to apply the function in *AllEx* only to obtain the constructor information. Having this information at the top-level, rather than at the *Sig* constructor, avoids the trouble of dealing with *AllEx* unnecessarily.

The function *typeSpine* has to be modified to deal with the new representation:

```
typeSpine :: Type a → TypeSpine a
typeSpine Int          = [ conint i (Sig i)
                         | i ← [minBound..maxBound]]
typeSpine (Maybe a)    = [connothing (Sig Nothing)
                         ,conjust    (Sig Just    :⊕: a)]
typeSpine (Either a b) = [conleft    (Sig Left    :⊕: a)
                         ,conright   (Sig Right   :⊕: b)]
typeSpine (List a)     = [connil     (Sig [])
                         ,concons (Sig (:) :⊕: a :⊕: List a)]
typeSpine Dynamic      =
  [condynval (AllEx (λa → Sig (DynVal a) :⊕: a))]
```

Now let us rewrite the *read* function using the new type spine view. First of all, the constructor is parsed in *read*, because the constructor information is now at the top-level:

```
read :: Type a → Parser a
read Int = readInt
read rep = alternatives [ readParen (conParser conrep)
                        | conrep ← typeSpine rep]
  where conParser conrep = token (conName conrep)
                           ≫ gread (conVal conrep)
```

The function that performs generic parsing is not very different for the first two Signature constructors:

```
gread :: Signature a → Parser a
gread (Sig c)         = return c
gread (con :⊕: arg) = gread con ‘ap‘ read arg
```

The existential case is the most interesting one. We first parse the type representation, and then continue with parsing the remaining part of the constructor.

```
gread (AllEx f) = readType ≫= applyBType (gread ∘ f)
```

This example effectively captures the *read* examples for dynamically typed values and for expression applications. The type representation is used to build the parser for the remaining constructor arguments. This dependency is expressed using the bind operation on parsers (≫=). This means that the definition of generic read for existential types must be based on monadic parser combinators, and therefore applicative parser combinators cannot be used.

There is one function that we use to read type representations:

```
readType :: Parser BType
```

Because type representations are somewhat special, we deal with them separately in Section 3.4.

3.2 Choice in the representation of existentials

There a choice in the representation of existential quantification. Consider the representation of *DynVal* given above. The function argument of *AllEx* receives a type representation and uses it to build the partially applied constructor value *Sig* (*DynVal a*). This value requires only one more argument which is represented by *a*.

An alternative way to encode *DynVal* is to make all of the constructor arguments explicit:

```
typeSpine :: Type a → TypeSpine a
...
typeSpine Dynamic =
   [condynval (AllEx (λa → Sig DynVal :⊕: Type a :⊕: a))]
```

The two approaches differ in whether a generic function has access to the type representation in the application case (:⊕:). It would seem that the second representation of *DynVal* is more flexible because it would allow the production of values different than *a* for the first argument. However, Type is a singleton type, so the only value (excluding ⊥) that inhabits the type represented by *Type a* is *a* itself. It follows that the second representation of *DynVal* is not an improvement over the first. For this reason, we always choose to expose the representation of an existential by means of the existential case only (*AllEx*).

3.3 The existential case for consumer functions

Producer functions need a modified type spine view (TypeSpine) to handle existential types. Do we need to modify the spine view (Spine) for consumers too? After all, we were able to define *toSpine* for Dynamic using the existing view. There is a good reason why we still need to modify the spine view to handle existentials in an appropriate way. Consider the *read* and *show* functions for example. There is a clear dependency on the representation of existential types during parsing. It is not possible (or at least very impractical) to parse a dynamic value without first having the type representation for it. Therefore, existential type representations should appear earlier than the constructor arguments that depend on it in the text input used for parsing. This means that *show* must pretty print the type representation for the existential before the dependent constructor arguments. However, the current spine view makes this difficult because the representation for the existential may appear in any position.

We solve the problem above making the dependence between existential types and constructor arguments explicit. Like the type spine view, the new constructor encodes the dependency on existentials using a function. The type variable is existentially quantified because in this case we are representing a specific constructor value:

```
data Spine :: ∗ → ∗ where
  Con   :: a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
  Ex    :: Type a → (Type a → Spine b) → Spine b
```

As in the type spine view, the constructor information is lifted out of the *Con* constructor onto the top-level. The new function *toSpine* is as follows:

```
toSpine :: Type a → a → ConInfo (Spine a)
toSpine Int          x          = conint x     (Con x)
toSpine (Maybe a)    (Just x)   = conjust      (Con Just
                                                 :◇: x :▷: a)
toSpine (Maybe a)    Nothing    = connothing   (Con Nothing)
toSpine (Either a b) (Left x)   = conleft      (Con Left
                                                 :◇: x :▷: a)
toSpine (Either a b) (Right y)  = conright     (Con Right
                                                 :◇: y :▷: b)
toSpine (List a)     (x : xs)   =
   concons (Con (:) :◇: x :▷: a :◇: xs :▷: List a)
toSpine (List a)     []         = connil       (Con [])
toSpine Dynamic      (DynVal a x) = condynval  (Ex a dynSig)
   where dynSig a = Con (DynVal a) :◇: x :▷: a
```

The *show* function is modified as follows to use the constructor information that appears at the top-level:

```
show :: Type a → a → String
show rep x = paren (conName spinecon
                      • gshow (conVal spinecon))
   where spinecon = toSpine rep x
```

Generic show does not change much for the two first spine cases:

```
gshow :: Spine a → String
gshow (Con con)     = ""
gshow (con :◇: arg) = gshow con • show (rep arg) (val arg)
```

For the existential case, generic show prints the type representation first and continues printing the remaining constructor values:

```
gshow (Ex a f) = showType a • gshow (f a)
```

The function for printing type representations is explained next:

```
showType :: Type a → String
```

3.4 Handling type representations

In the example above, we have used the function *readType* to parse a type representation. The function *readType* returns a boxed representation since the represented type is dynamically determined during parsing. Unfortunately, it is not easy to define producers that return boxed representations using generic programming. If special care is not taken, such functions may loop when invoked. In the following we describe the problem in more detail and we propose a solution.

3.4.1 Parsing type representations

The obvious way to parse a type representation is to do it generically by using the *read* function. To this end, we use generic read to parse boxed type representations:

```
readType :: Parser BType
readType = read BType
```

Unfortunately, the function given above is non-terminating. First, remember that BType uses existential quantification, and hence its type spine is:

```
typeSpine BType = [conboxed (AllEx (λa → Sig (Boxed a)))]
```

Since the type spine uses an existential case, *gread* would try to parse a BType-value calling *readType* recursively. Therefore, trying to parse a boxed type representation would lead to parsing an existential type, which leads to parsing a boxed type representation and so on.

How can we solve this problem? A desperate solution would be to give up using generic programming in the definition of *readType*. This approach is undesirable because every generic producer would need to have a type representation case. Worse even, every such case would have to handle all type representation constructors. If there are *n* generic functions and *m* represented types, the programmer would need to write $n \times m$ cases. Despite this significant problem, it is worth exploring a non-generic variant of *readType* and try to generalize it.

```
readType = alternatives (map readParen
               [ do token "Int"
                    return (Boxed Int)
               , do token "Maybe"
                    Boxed arg ← readType
                    return (Boxed (Maybe arg))
               , do token "Either"
                    Boxed left  ← readType
                    Boxed right ← readType
                    return (Boxed (Either left right))
               , do token "List"
                    Boxed arg ← readType
                    return (Boxed (List arg))
               ])
```

This example shows that parsing a type representation is no different than parsing a normal datatype in that the type argument of the GADT plays no role here. This example also illustrates the verbosity of writing such boilerplate without using generic programming.

The code of *readType* suggests that we could forget the "GADT-ness" of type representations during parsing. This is the first step we take towards being able to define generic producers for boxed representations, namely, defining the datatype of type codes, a non-GADT companion to type representations:

```
data TCode :: ∗ where
  CInt     :: TCode
  CMaybe   :: TCode → TCode
  CEither  :: TCode → TCode → TCode
  CList    :: TCode → TCode
  CArrow   :: TCode → TCode → TCode
  CType    :: TCode → TCode
  CDynamic :: TCode
  CTCode   :: TCode
```

Besides naming and the absence of a type argument, this datatype is identical to type representations. To make the relation between type codes and type representations precise, we introduce two conversion functions. The first function converts a type representation to a type code, erasing the type information in the process:

```
eraseType :: Type a → TCode
eraseType Int          = CInt
eraseType (Maybe a)    = CMaybe  (eraseType a)
eraseType (Either a b) = CEither (eraseType a) (eraseType b)
eraseType (List a)     = CList   (eraseType a)
eraseType (a :→ b)     = CArrow  (eraseType a) (eraseType b)
eraseType (Type a)     = CType   (eraseType a)
eraseType Dynamic      = CDynamic
eraseType TCode        = CTCode
```

Conversely, we want to be able to convert from a type code to a type representation. Note, however, that the resulting type-index depends on the value of the type code and hence the result is a boxed representation:

```
interpretTCode :: TCode → BType
interpretTCode CInt          = Boxed Int
interpretTCode (CMaybe a)    = applyTCode (Boxed ∘ Maybe) a
interpretTCode (CEither a b) =
  applyTCode (λr → applyTCode (Boxed ∘ Either r) b) a
interpretTCode (CList a)     = applyTCode (Boxed ∘ List) a
interpretTCode (CArrow a b)  =
  applyTCode (λr → applyTCode (Boxed ∘ (r :→ )) b) a
interpretTCode (CType a)     = applyTCode (Boxed ∘ Type) a
interpretTCode CDynamic      = Boxed Dynamic
interpretTCode CTCode        = Boxed TCode

applyTCode :: ∀c.(∀a.Type a → c) → TCode → c
applyTCode f code = applyBType f (interpretTCode code)
```

Using type codes it is now possible to implement parsing of type representations generically. To implement *readType*, we parse a type code value and then we interpret it to obtain a type representation:

```
readType :: Parser BType
readType = read TCode >>= return ∘ interpretTCode
```

Here *TCode* is the type representation for type codes, we do not show the spine and type spine views for this datatype as they are no different from that of other datatypes.

Showing a type representation was no problem previously, we could have written *showType* as follows:

```
showType :: Type a → String
showType a = show (Type a) a
```

However, to remain compatible with *read* we use type codes as the means to pretty print type representations:

```
showType :: Type a → String
showType = show TCode ∘ eraseType
```

Summarizing, *readType* is a special function. It cannot be defined by instantiating *read* to boxed representations. Such an instantiation leads to non-termination because parsing a boxed representation uses the existential case of generic parsing, which in turn makes the recursive call to *readType*. To solve this problem, we defined type codes, a non-GADT analogue of type representations. Non-termination is no longer an issue with type codes. To parse a type code we no longer need to parse existential types, which prevents the recursive call to *readType*. This machinery enables the definition of *readType* as a generic program. This machinery can be reused for other generic producers, for example, see the definition of *enumerateType* in Section 4.

3.5 Equality of type representations

In this section, we have introduced machinery to handle type representations generically, namely type codes and conversion functions between type codes and type representations. In Section 4, we show an advanced GADT example that requires a last piece of machinery: equality on type representations.

Below we show a function which compares two type representations, if the two representations are equal, it returns a proof that the two values represent the same type. First, we introduce the type of type equalities:

```
data TEq :: ∗ → ∗ → ∗ where
  Refl :: TEq a a
```

A value of type TEq a b can be used to convince the type checker that two types a and b are the same at compile time. Since two type representations may not be the same, function *teq* returns the result in a monad:

```
teq :: Monad m ⇒ Type a → Type b → m (TEq a b)
teq Int         Int         = return Refl
teq (Maybe a)   (Maybe b)   = liftM  cong₁ (teq a b)
teq (List a)    (List b)    = liftM  cong₁ (teq a b)
teq (Either a c) (Either b d) = liftM2 cong₂ (teq a b) (teq c d)
teq (Lam a c)   (Lam b d)   = liftM2 cong₂ (teq a b) (teq c d)
teq (a :→ c)    (b :→ d)    = liftM2 cong₂ (teq a b) (teq c d)
teq _           _           = fail "Different reprs"
```

Here, we use *liftM* and *liftM2* to turn congruence functions into functions on monads. Congruence functions are used to lift equality proofs of types to arbitrary type constructors. These are defined as follows:

```
cong₁ :: TEq a b → TEq (f a) (f b)
cong₁  Refl      = Refl
cong₂ :: TEq a b → TEq c d → TEq (f a c) (f b d)
cong₂  Refl        Refl     = Refl
```

3.6 Type codes and dependently typed programming

In the literature of generic programming based on dependent types, sets of types having common structure are modelled by universes (Benke et al. 2003). Values known as universe codes describe type structure and an interpretation function makes the relationship between codes and types explicit.

The generic programming approach that this chapter describes would greatly benefit from the use of dependent types. Our approach is slightly redundant due to the necessity of both type representations and type codes. If we were to revise our approach to use dependent types, the generic machinery would be based on type codes only. Previously, the type representation datatype described the relationship between types and the values that represent them. Using dependent types, this relationship would be defined by interpretation on codes and therefore type representations would not be necessary. Furthermore, producers like *readType* would no longer need to generate type representations. It follows that it would not be possible to accidentally define a non-terminating variant of such producers.

3.7 On the partiality of parsing typed syntax trees

Parsing is necessarily a partial operation. A parser for lists will fail to produce a value if the string to parse is not the textual representation of a list. Generic read is also a partial operation: the constructor names to be recognized in the input depend on the type representation argument of *gread*.

Generalized algebraic datatypes make the behavior of *gread* more interesting. When a GADT is used, the set of constructors to recognize in the input will, in general, be a subset of all constructors in the GADT. For example, *gread* (*Expr Int*) parses all constructors with target type Expr Int but it fails to recognize the constructors *EFalse* and *ENot*. Note that this behavior is closely related to type-checking: what would be type-checking errors in a different context are presented here as parsing errors.

A more interesting case is that of expression application. In this case, a representation for the function argument type is parsed and it steers the parsing of the function and the argument expressions. In this case, a type incompatibility between function and argument would be revealed as a parsing error.

4. Application: enumeration applied to simply typed lambda calculus

Generalized algebraic datatypes can encode sophisticated invariants using type-level constraints. We can combine such precise datatypes with generic producer functions, to generate values that have interesting properties. The example of this section combines a datatype representing terms of the simply typed lambda calculus with a generic function that enumerates all the values of a datatype. Using this function we can, for example, generate the terms that have the type of function composition.

4.1 Representing the simply typed lambda calculus

Terms of the simply typed lambda calculus can be represented as follows:

```
data Lam :: ∗ → ∗ → ∗ where
  Vz  :: Lam a (EnvCons a e)
  Vs  :: Lam a e → Lam a (EnvCons b e)
  Abs :: Lam b (EnvCons a e) → Lam (a → b) e
  App :: Type a → Lam (a → b) e → Lam a e → Lam b e
```

The datatype Lam can be read as the typing relation for the simply typed lambda calculus. A value of type Lam a e represents the typing derivation for a term of type a in an environment e. Environments are encoded by list-like type constructors:

```
data EnvCons a e
data EnvNil
```

Each Lam constructor is a rule of the typing relation. The first constructor (*Vz*) represents a variable occurrence of type a, which refers to the first position of the environment (EnvCons a e). We can build a variable occurrence that refers to a deeper environment position by means of the weakening constructor *Vs*. Lambda abstractions are typed by means of the *Abs* constructor. In this case, a b-expression that is typeable in an environment containing a in the first position can be turned into a lambda abstraction of type a → b. The application constructor is almost like application in our previous example, *EApp*, except that *App* includes a representation for the existential type.

The spine representation for this datatype can be defined as follows:

```
toSpine (Lam a e) Vz = convz (Con Vz)
toSpine (Lam a (EnvCons b e)) (Vs tm) =
    convs (Con Vs :◇: tm :▷: Lam a e)
toSpine (Lam (a :→ b) e) (Abs tm) = conabs (Con Abs :◇: body)
  where body = tm :▷: Lam b (EnvCons a e)
toSpine (Lam b e) (App a tm₁ tm₂) = conapp (Ex a app)
  where app a = Con (App a) :◇: tm₁ :▷: Lam (a :→ b) e :◇: tm₂
                 :▷: Lam a e
```

The type representations are pattern matched in the *Vs* and *Abs* constructors to build the representation in the right hand side. The *App* constructor has an existential type, therefore we use *Ex* in the spine representation. Using the type representation, we can now print lambda terms.

For producer functions, we define the type spine view on Lam as follows:

```
typeSpine (Lam a e) = concat
  [[convz  (Sig Vz) | EnvCons a' e' ← [e], Refl ← teq a a']
  ,[convs  (Sig Vs :⊕: Lam a e') | EnvCons b e' ← [e]]
  ,[conabs (Sig Abs :⊕: Lam b (EnvCons a' e)) | a' :→ b ← [a]]
  ,[conapp
      (AllEx
         (λb → Sig (App b) :⊕: Lam (b :→ a) e :⊕: Lam b e))]
  ]
```

We test whether a constructor signature has the desired target type by performing pattern matching on type representations. The cases *Vz* and *Vs* are only usable if the environment type argument is not empty. Additionally, the target type of *Vz* requires the equality of the type and the first position in the environment. Therefore, the *Vz* case invokes type equality (*teq*) on the term type (*a*) and the type of the first environment position (*a'*). The abstraction constructor

(*Abs*) requires an arrow type, which is checked by pattern matching against an arrow type representation. The application constructor can always be used, because there is no restriction on the target type of *App*.

This type spine representation is more informative and larger than previous examples. The reason is that the GADT type argument is more complex because of the use of type-level environments. Furthermore, the type constraint in *Vz* requires the use of type equality (*teq*). Fortunately, it is possible to generate the type spine representation by induction on the syntax of the datatype declaration. It would be possible to automate this process using external tools such as DrIFT and Template Haskell if these tools supported GADTs.

4.2 Breadth first search combinators

The generic enumeration function generates all possible values of a datatype in breadth first search (BFS) order. The order used in the search corresponds to the search cost of terms generated. The type BFS is used for the results of a breadth first search procedure:

$$\textbf{type}\ \mathsf{BFS\ a} = [[\mathsf{a}]]$$

The type BFS represents a list of multisets sorted by cost. The first multiset contains terms of cost zero, the second contains terms of cost one and so on. Using this datatype, a consumer can inspect the terms up to a certain cost bound and hence the search does not continue if further terms are not demanded. This is useful because the enumeration function returns a potentially infinite list of multisets.

Multiple BFS values can be zipped together by concatenating multisets having terms of equal cost:

```
zip_bfs :: [BFS a] → BFS a
zip_bfs []  = []
zip_bfs xss = if all null xss then [] else
                concatMap head xss' : zip_bfs (map tail xss')
    where xss' = filter (¬ ∘ null) xss
```

It is more convenient to manipulate BFS results using monadic notation. Therefore, we define return and bind on BFS:

```
return_bfs x = [[x]]
(>>=_bfs) :: ∀a b.BFS a → (a → BFS b) → BFS b
(>>=_bfs) xss f =
    foldr (λxs xss → zip_bfs (map f xs ++ [[] : xss])) [] xss
```

Return creates a search result that contains a value of cost zero. Bind feeds the terms found in a search *xss* to a search procedure *f*. The cost of the term passed to *f* is added to the costs of that search procedure. Consider, for example, the search results *aSearch* consisting of the terms $\lambda x\, y \to y$ and $\lambda x\, y \to x$ with costs three and four respectively; and a search procedure that produces a term of cost one by adding an abstraction to its argument:

```
aSearch    = [[],[],[],[Abs (Abs Vz)],[Abs (Abs (Vs Vz))]]
f       tm = [[],[Abs tm]]
```

Then, the expression ($aSearch \gg\!\!=_{bfs} f$) evaluates to the following:

```
[[],[],[],[],[Abs (Abs (Abs Vz))],[Abs (Abs (Abs (Vs Vz)))]]
```

The two terms in the initial search result now have an additional abstraction argument and have costs of four and five respectively.

The cost addition property of bind can be stated more formally as follows:

```
propBind :: BFS a → (a → BFS b) → Bool
propBind xss f = all (all costBind) (costs xss f)
   where costBind (c,(c_xss,c_f)) = c ≡ c_xss + c_f
costs :: BFS a → (a → BFS b) → BFS (Int,(Int,Int))
costs xss f = cost (cost xss  >>=_bfs λ(c_xss,x) →
                    cost (f x) >>=_bfs λ(c_f,y)  →
                    return_bfs (c_xss,c_f))
```

where *cost* annotates each BFS result value with its cost:

```
cost :: BFS a → BFS (Int,a)
cost = zipWith (λsz → map ((,) sz)) [0..]
```

Additionally we use a function that increases the cost of the values found in a search procedure:

```
spend :: Int → BFS a → BFS a
spend n = (!!n) ∘ iterate ([]:)
```

When using a very expensive search procedure, it is useful to increase the cost of terms exponentially:

```
raise :: Int → BFS a → BFS a
raise base xss = traverse 0 xss where
  traverse _   []       = []
  traverse 0   (xs:xss) = [] : xs : traverse 1 xss
  traverse exp (xs:xss) = spend (base^exp − base^(exp−1) − 1)
                                (xs : traverse (exp + 1) xss)
```

For example, *spend* 2 *aSearch* and *raise* 2 *aSearch* evaluate to:

```
[[],[],[],[],[],[Abs (Abs Vz)],[Abs (Abs (Vs Vz))]]
```

and

```
[[],[],[],[],[],[],[],[],[Abs (Abs Vz)],
 [],[],[],[],[],[],[],[Abs (Abs (Vs Vz))]]
```

respectively.

4.3 Generic enumeration

The generic enumeration function returns values of a datatype, classified by cost in increasing order. The cost of a term is the number of datatype constructors used therein (constructors used in type representations are an exception and we discuss them last in the definition of *enumerateType*).

```
enumerate :: Type a → BFS a
enumerate a = zip_bfs [genumerate (conVal s) | s ← typeSpine a]
```

At the top-level, function *genumerate* is invoked on each constructor signature and the resulting search results are zipped together.

The first case of *genumerate* returns the constructor value as the search result assigning it a cost of one. The second case performs search recursively on the function and argument parts and combines the results using BFS monadic application ap_{bfs} :: BFS (a → b) → BFS a → BFS b.

```
genumerate :: Signature a  → BFS a
genumerate   (Sig c)       = spend 1 (return_bfs c)
genumerate   (fun :⊕: arg) =
   genumerate fun `ap_bfs` enumerate arg
```

The third case deals with existential types and hence in our particular application it deals with expression application. This case first enumerates all possible types, and then constructs a constructor signature using *f*, for each type, and enumeration is called recursively:

```
genumerate (AllEx f)   =   enumerateType
                           >>=_bfs genumerate ∘ applyBType f
```

As usual with producer functions, *enumerateType* returns a boxed representation. The enumeration of types is performed on type codes, which *interpretTCode* converts to boxed representations.

```
enumerateType :: BFS BType
enumerateType = raise 4 (enumerate TCode)
                  >>=_bfs return_bfs ∘ interpretTCode
```

For the examples in this paper, we are not interested in values that have very complex existential types. Therefore, we keep their size small by assigning an exponential cost to existentials. This also has the effect of reducing the search space, which makes the generation of interesting terms within small cost upperbounds more likely.

4.4 Term enumeration in action

For convenience, we define a wrapper function to perform enumeration of lambda terms:

```
enumerateLam :: Type a → Int → BFS (Lam a EnvNil)
enumerateLam a cost = take (cost + 1)
                        (enumerate (Lam a EnvNil))
```

Our term datatype can perfectly deal with open terms. But the user interface becomes simpler if only closed terms are provided. Therefore, the wrapper function only generates closed lambda terms.

A direct invocation of the enumeration function will result in an attempt to generate an infinite number of terms. For convenience, our wrapper function takes a cost upperbound that limits the cost of terms that are reported. Because of lazy evaluation, the search procedure stops when all terms within the cost bound are reported. The user may choose to increase the cost upperbound in subsequent invocations if the desired term is not found.

The language that the Lam datatype represents is very simple. There are no datatypes, recursion, and arithmetic operations. For example, we cannot expect the enumeration function to generate the successor or predecessor functions for naturals if functions of the type Int → Int are requested. In principle, it is not difficult to extend the language by adding the appropriate constants to Lam. For example, we could add naturals and arithmetic operations on them. We could also add list constructors and elimination functions and even recursion operators such as catamorphisms and paramorphisms.

However, we can keep our language simple and still generate many interesting terms. We focus our attention to parametrically polymorphic functions. Although we do not model parametric polymorphism explicitly in Lam, such functions are naturally generated when the requested type is an instance of the polymorphic type. For instance, a request with type Int → Int generates the identity function.

To make the intent of generating polymorphic functions more explicit, we define a few types that are uninhabited in the Lam language. These types play the role of type variables in polymorphic type signatures:

```
data A
data B
data C
data D
```

Of course, we also introduce the corresponding type representation constructors:

```
data Type :: ∗ → ∗ where
  ...
  A :: Type A
  B :: Type B
  C :: Type C
  D :: Type D
```

In our first example, we generate the code for the identity function. The type of the identity function is $\forall a.a \to a$, which in our notation translates to $A :\to A$. The function we expect to generate is $\lambda x \to x$, which in Lam is written as *Abs Vz*. This term consists of two constructors, therefore a cost upperbound of two should suffice to generate it. The application *enumerateLam* $(A :\to A)$ 2 results in:

```
[[ ], [ ], [Abs Vz]]
```

It is instructive to sketch the search procedure as it looks for the identity function. First, *enumerate* is called on the identity type with a closed environment. This function calls *genumerate* on all constructor signatures that match the desired type. The two variable cases *Vz* and *Vs* are not considered, because they cannot be used with an empty environment. Application can always be used but recall that it requires an existential type representation, so the cost is at least 5, which is more expensive than the function that we are looking for. The abstraction case matches the identity type so enumeration is called recursively to generate the abstraction body. Now, a term of type A is requested with a type A in the first position in the environment. The case *Vz* matches perfectly with this request so the term *Abs Vz* is returned with cost two.

There are infinitely many lambda calculus terms of a given type when that type is inhabited. A simple way to way to obtain a new term is by creating a redex that reduces to the term that we currently have. For example, we can obtain a new identity function by adding a redex in the function body: $\lambda x \to (\lambda x \to x)\ x$. Can this term be found by our enumeration function? Yes, provided that we increase the cost upperbound to include that of our new term. The new term is essentially two identity functions plus an application constructor, which makes a cost upperbound of nine. We evaluate *enumerateLam* $(A :\to A)$ 9 which yields:

```
[[ ], [ ], [Abs Vz], [ ], [ ], [ ], [ ], [ ], [ ], [Abs (App A (Abs Vz) Vz)]]
```

This example shows that the search space is somewhat redundant. A way to speed up term search would be to avoid the generation of redundant terms by adding constraints to Lam. For example, we could avoid redeces by preventing the generation of abstractions in the left part of applications.

Another interesting example is the generation of the application function. This function has type ∀a b.(a → b) → a → b, which in our notation is written $((A :\to B) :\to A :\to B)$. We evaluate *enumerateLam* $((A :\to B) :\to A :\to B)$ 10 to generate an application function, which results in:

```
[[ ], [ ], [Abs Vz], [ ], [ ], [ ], [ ], [ ], [ ], [ ],
 [Abs (Abs (App A (Vs Vz) Vz))]]
```

These are the encodings for the functions $\lambda x \to x$ and $\lambda x\ y \to x\ y$. The careful reader may wonder why the other identity term $\lambda x \to (\lambda x \to x)\ x$, which has cost 8 in the previous example, is not generated. The answer is that the cost of the term includes that of the type representation used in the application constructor. Since this example has a different type, the type representation would be $A :\to B$ rather than A. It follows that the term $\lambda x \to (\lambda x \to x)\ x$ is not generated because it has a cost of 14.

Our last example is function composition. The type of this function is ∀a b c.(b → c) → (a → b) → a → c. To generate composition, we evaluate

```
last (enumerateLam ((B :→ C) :→ (A :→ B) :→ (A :→ C)) 19)
```

which yields to the encoding of $\lambda x\ y\ z \to x\ (y\ z)$:

```
[Abs (Abs (Abs (App B (Vs (Vs Vz)) (App A (Vs Vz) Vz))))]
```

5. Related work

To the best of our knowledge, only the spine approach (Hinze et al. 2006; Hinze and Löh 2006) enables generic programming on generalized algebraic datatypes in Haskell. This is the approach on which the work in this chapter is based. Because both the spine and the type spine view can encode GADTs, both consumer and producer functions can be defined on such datatypes. Interestingly, to properly support GADTs for producer functions, the approach should also support existential types. For example, when reading a GADT from disk, we may want the GADT type argument to be dynamically determined from the disk contents. Therefore, we would existentially quantify over that argument. However, the spine approach supports existential types for consumers but not for producers.

Generalized algebraic datatypes are inspired by inductive families in the dependent types community. We are aware of two approaches (Benke et al. 2003; Morris 2007) that support definitions by induction on the structure of inductive families. Both approaches make essential use of evaluation on the type level to express the constraints over inductive families. Examples of type families on which generic programming is applied include trees (indexed by their lower and upper size bounds), finite sets, vectors and telescopes. Neither approach gives examples for the support of existential types so it is not clear whether these are supported.

Weirich (2002) proposes a language that provides a construct to perform runtime case analysis on types. In order to support universal and existential quantification, the language includes analyzable type constants for both quantifiers. This approach supports the definition of consumers and producers. Moreover, if the language is extended with polymorphic kinds it supports quantification over arbitrarily kinded types.

Our approach to defining breadth-first search combinators is not novel. Spivey (2000) defines a set of breadth-first search combinators such as monadic join and composition, and proves desirable properties for them. There are many similarities between our work and that of Spivey. It would be interesting to see whether our combinators satisfy the same properties as the combinators proposed by Spivey.

Koopman and Plasmeijer (2007) generate lambda calculus terms by performing systematic enumeration based on a grammar. To reduce the size of the search space, the grammar has syntactic restrictions such as that the applications of certain operands are always saturated, and recursive calls are always guarded by a conditional. The candidate terms are then reported to the user based on whether they satisfy an input-output specification, which is established by evaluation.

Djinn (Augustsson 2005) generates lambda calculus terms based on a user-supplied type. This tool implements the decision procedure for intuitionistic propositional calculus due to Dyckhoff (1992). Similarly, the work of Katayama (2005) makes use of a type inference monad to generate well-typed terms. Later, the candidate terms are evaluated and checked against an input-output specification. As in our approach, Djinn and the approach of Katayama generate only well-typed terms so there is no need for a type checking phase to discard ill-typed terms.

The main difference between the work of Koopman and Plasmeijer (2007) and ours is that our generator is typed-based. It follows that our generator never returns ill-typed terms because the search space is reduced by means of type-level constraints in the GADT. Generating ill-typed terms has advantages. For example, Koopman's approach can generate the Y-combinator. On the other hand, ill-typed terms are usually not desirable, so these have to be discarded through either evaluation (Koopman), which slows down the generation algorithm. In Koopman's work the generation of ill-typed terms is prevented to some extent by the syntactic constraints imposed on the grammar.

A type-based generator, such as Djinn, Katayama's generator and our approach, is able to synthesize polymorphic functions without the need for input-output specifications. Koopman's work, however, cannot generate polymorphic functions based solely on type information.

Djinn supports user-defined dataypes. Katayama and Koopman's generators are able to generate recursive programs. Our approach currently generates programs for a rather spartan language. However, it should be possible to add introduction and elimination constants for (recursive) datatypes, and recursion operators such as catamorphisms and paramorphisms.

Both Djinn and our approach enumerate terms guided by type information. However, the two approaches are very different. Djinn has a carefully crafted algorithm that handles the application of functional values in such a way that it is not necessary to exhaustively enumerate the infinite search space. As a consequence, Djinn is able to detect that a type is uninhabited in finite time. In contrast, our approach produces function applications by means of exhaustive enumeration. First, all the possible types of an argument are enumerated, and, for each of them, function and argument terms are enumerated to construct an application. The good side of an exhaustive approach like ours is that it can generate all possible terms of a given type. For example, it can generate all Church numerals, whereas Djinn only generates those corresponding to zero and one. On the bad side, if unbounded, our approach does not terminate when trying to generate a term for an uninhabited type.

We have not performed a careful performance comparison but we believe that our generator may be the slowest of the approaches considered here. Probably the main culprit for inefficiency is the implementation of the existential case. Currently this case enumerates all possible types, even if no applications for that argument type can be constructed. Ill-typed terms are never generated, but resources are nevertheless consumed when attempting to enumerate terms having possibly uninhabited types. It is difficult to make the algorithm smarter about generating types because, being generic, it does not make assumptions about the particularities of lambda calculus. On the other hand, it is possible to reduce the search space by adjusting the definition of `Lam`. For example, we could forbid the formation of redeces to avoid redundancy of terms, or even adopt the syntactic restrictions used in Koopman's work.

While our approach may be less efficient, it has the virtue of simplicity: the core of the generation algorithm consists of roughly a dozen lines of code and there is no need for an evaluation or a type checking phase. Furthermore, it has the advantage of an elegant separation between the grammar constraints and the formulation of the enumeration algorithm. This allows us to use the enumeration function to generate other languages, whereas the other generators are specific to lambda calculus.

6. Conclusions

We have presented an extension of the spine approach to generic programming, which supports the definition of generic producers for existential types. This extension allows the definition of, for example, generic read for datatypes that use existential quantification.

Our approach opens the way for a new application of generic programming. By taking the standard enumeration generic function and extending it with a case for existentials, we obtain a function that enumerates well-typed terms. For example, we can instantiate enumeration to the GADT that represents terms of the simply typed lambda calculus and use the resulting function to search for terms that have a given type. Such an application was not previously possible because producers that handle existential types could not be generically defined.

Acknowledgments

We are grateful to Andres Löh for the productive discussions that inspired this paper. The ST reading club at Utrecht University gave detailed feedback on an earlier version of this paper. We thank the anonymous reviewers for the useful suggestions and comments. We also thank Lambert Meertens, the careful reader who motivated us to improve the explanations in Section 4.4.

References

Lennart Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. Available from `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`, 2005.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 286–279, 2000.

R Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC'06*, LNCS 4014, pages 180–208, 2006.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *FLOPS'06*, LNCS 3945, 2006.

Susumu Katayama. Systematic search for lambda expressions. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming, TFP 2005*. Institute of Cybernetics, Tallinn, 2005. ISBN 9985-894-88-X.

Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of λ-terms. In *Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, 2007.

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *IFL'02*, volume 2670 of *LNCS*, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007. doi: 10.1017/S0956796807006326.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, 2007.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP'06*, pages 50–61, 2006.

Michael Spivey. Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796800003749.

S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

Stephanie Weirich. Higher-order intensional type analysis. In *In Proc. 11th ESOP, LNCS 2305*, pages 98–114. Springer-Verlag, 2002.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'03*, pages 224–235, New Orleans, January 2003.

Defining Inductive Operators using Distances over Lists [*]

V. Estruch C. Ferri J. Hernández-Orallo M.J. Ramírez-Quintana

DSIC, Univ. Politècnica de València
Cami de Vera s/n, 46020 València, Spain
{vestruch,cferri,jorallo,mramirez}@dsic.upv.es

Abstract

Instance-based learning is one of the most widely-used paradigms in the field of automatic induction. Several reasons back its popularity, among them, we must stand out its capability to cope with different data representations: these methods are designed on the basis of a similarity principle (similar examples should share similar properties) which makes them easily adaptable to different datatypes via redefining the similarity (distance) function. In this sense, multiple distances and similarity functions can be found in the literature.

However, the most notorious downside when speaking of distance-based or similarity-based methods concerns the low expressivity of the models (if any) these methods learn. Decisions are made from expressions such as "example x is more similar or nearer to example y then" which results in little practical knowledge, very specially when structured data is involved. However, in many application areas we require patterns to describe the similarities of the data.

In [Estruch 2008], we have addressed and formalised the problem of turning distance-based methods outputs into comprehensible and consistent patterns. In this work, we first overview our framework and then instantiated it for the case of data represented by lists of symbols.

Keywords inductive operators, induction with distances, list-based representations

1. Introduction

Inductive Programming is concerned with the automated construction of declarative programs from data. We can distinguish several approaches to this problem according to the knowledge representation adopted. For instance, the field known as *Inductive Logic Programming* (ILP) [Muggleton 1999] aims to induce consistent first order theories from data represented as first order objects (atoms or clauses). A natural extension of this comes when we move to higher-order logics [Bowers et al. 2000, Lloyd 2003]. The synthesis of functional programs arises when the training data consists in a sample of inputs and outputs of a evaluation function [Olsson 1995, Schmid 2003]. A more generic framework corresponds to the induction of functional-logic theories. This paradigm centres on performing induction within a formal context that combines the strengths of logic and functional programming [Ferri et al. 2001].

Although declarative languages constitutes an elegant and powerful framework for program synthesis, they show some limitations when the semantic of the data representation does not match the implicit semantic managed by these declarative languages. An example of this is found when working with lists or sequences[1]. From a declarative point of view, lists are recursively defined in terms of a special item (head) and a tail, which is another (sub)list. This perspective makes difficult the search of patterns in data that does not suit this definition. For instance, if we are given the lists $abaca$ and bc, it is not immediate to learn a pattern of the form $*b*c*$ because of the simple fact that the heads of the lists do not match.

Unfortunately, list-based representations appear in many real-world domains, which might put some limits on the applicability of declarative tools. For instance, in bioinformatics, compounds such as amino-acids have a direct representation as sequences of symbols. Furthermore, other much more complex molecules can also be described in terms of sequences by using the so-called 1-D or SMILE representation [Swamidass et al. 2005]. Another example is found in text or web mining where documents are usually transformed into sequences of words. Very common software utilities such as command line completion or orthographic correctors work on lists as well.

At this point, we could wonder if some of the tools employed in inductive programming (generalisation operators) could be upgraded to deal with list-based representations in a more satisfactory way and overcome this limitation. In [Estruch et al. 2005, 2006], we consider the possibility by analysing the relationship between distance and generalisation

[*] This work was partially supported by the EU (FEDER) and the Spanish Government MEC/MICINN, under grant TIN 2007-68093-C02 and the Spanish project "Agreement Technologies"(CONSOLIDER-INGENIO CSD2007-00022).

[1] In this section, the terms *list* and *sequence* will be used indistinctly.

Note that most of the applications that handle sequences usually employ distances in order to find the most similar sequences in data. Distances (and consequently, metric spaces) play an important role in many inductive techniques that have been developed to date. Similarity offers a well-founded inference principle for learning and reasoning since it is commonly assumed that *similar objects have similar properties*. Given the importance of lists as a datatype for knowledge representation, several distances can be found in the literature, being the edit distance [Levenshtein 1966] the best-known. The drawback is that these methods do not infer a model (or patterns) from data as declarative inductive (or more general, symbolic) learners do.

Therefore, if we were able to find out a connection between distance and generalisation we could, on the one hand, define more suitable generalisation operators to work with structured data in general and with lists in particular; and on the other hand, we could come up with induction techniques capable of transforming distance-based method outputs into symbolic models, and consequently, more comprehensible explanations for the user.

Although, there might be many different ways to establish a connection between distance and generalisation, ensuring the consistency between them is compelling one. Note that if the generalisation process is not driven by the distance, this might result in patterns that does not capture the semantic of the distance giving wrong explanations about why objects are similar. Let us see an example of this. If we consider the edit distance over the lists *bbab*, *bab* and *aaba*, we see that the list *ab* is close to the previous lists (distances are 2, 1, and 2 respectively). However, a typical pattern that can be obtained by some model-based methods, **ba**, does not cover the list *ab*. The pattern does cover the list *dededfafbakgagggeewdsc*, which is at distance 20 from the three original lists. The pattern and the distance are up to some point inconsistent since those elements that are most similar to the initial examples are excluded.

Although there are other important works on hybridisation, they tend to ignore the problem of consistency between the semantic of the model learnt and the semantic of the underlying distance. Basically, what we do is to define some simple conditions that a generalisation operator should have in order to behave in a consistent way wrt. a distance. These operators are called distance-based generalisation operators.

In this paper, we address the problem of inducing patterns from lists of symbols embedded in a metric space. In other words, the work we present here can be seen as an instantiation for lists of the general framework aforementioned. This paper is organised as follows. Section 2 contains an overview of our proposal. In Section 3, we analyse how our framework could be used to learn symbolic patterns from lists. To this end, we introduce two different pattern languages $\mathcal{L}_0$ and another more expressive $\mathcal{L}_1$, and study how to define (minimal) distance-based operators in all of them. Finally, conclusions and future work are given in Section 4.

2. Setting

In this section we summarise the main concepts of our framework which integrates distances and generalisation. For a more detailed presentation of it we refer the reader to [Estruch 2008].

The underlying idea in our proposal is that, in order to have a *true* connection between distance and generalisation, the generalisation process have to take the underlying distance into consideration (or at least the two must be consistent). This special relation is formalised through three notions: *reachability*, *intrinsicality* and *minimality*.

Reachability implies that the generalisation of two elements ought to include those paths (a sequence of elements in the metric space) that allow us to reach both elements from each other by making small "steps". The concept of short step must be understood in the sense of the distance.

The second property arises from the observation that the distance between two elements is always given by the length of the shortest paths. Thus, if we want our generalisation to be compatible with the distance, we need the elements belonging to the shortest paths to be covered by the generalisation. This condition is called *intrinsicality*.

The two above properties have been defined for two elements since they are established in terms of the distance which is a binary function. But generalisation operators are not binary, thus for more than two elements, the connection between distance and generalisation turns a bit unclear. It seems that the properties of reachability and intrinsicality must be extended for this generic case. Distance-based algorithms suggest that it would make sense to impose the notion of intrinsicality for *some* pairs of elements. The pairs of elements that will have to comply with the intrinsicality property will be set by a path or connected graph which we will call *nerve*. Furthermore, we obtain with this a more generic notion of reachability since all the elements in the set are reachable from any of them by moving from one element to another through combinations of (intrinsical) paths.

In Figure 1, generalisations $G1$ and $G2$ do not connect the three elements to be generalised. Only the generalisations $G3$ and $G4$ connect the three elements through combinations of straight segments.

Finally, the last property concerns with the notion of *minimality*, which is understood not only in terms of fitting the set (i.e., semantic minimality) but also as the simplicity of the pattern (i.e., syntactic minimality). In Figure 1, $G3$ is an example of a very specific and rather complicated generalisation of A, B and C.

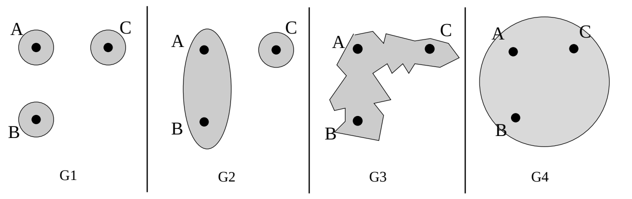

Figure 1. Generalising the elements $E = \{A, B, C\}$. Elements in E are not reachable through a path of segments in generalisations $G1$ and $G2$. For any two elements in E, generalisations $G3$ and $G4$ include a path of segments connecting them.

2.1 Distance-based Inductive Operators

Next, we formally show how the three previous notions are employed in order to define the so-called distance-based generalisation operators.

A generalisation of a finite set of elements $E \subset X$ could be seen as any superset of E in X. Therefore, a generalisation operator (denoted by Δ) simply maps sets of elements E into supersets. As known, this superset can be extensionally or intensionally defined, being the latter one more useful from a predictive/explanatory point of view. Symbolic patterns constitute a widely-spread manner of representing intensional generalisations. For instance, the pattern $a*$ denotes all the lists headed by the symbol a. We denote by $\mathcal{L}$ the pattern language and by $Set(p)$ the set of all the elements in X that the pattern $p \in \mathcal{L}$ represents. For instance, $Set(a*) = \{a, aa, ab, \ldots\}$. If necessary, $\mathcal{L}$ expressiveness can always be increased by combining patterns via logical operators (e.g. pattern disjunction). In this work, disjunction is denoted by the symbol $+$ and the expression $p_1 + p_2$ represents the set $Set(p_1) \cup Set(p_2)$. For simplicity, the pattern $p = p_1 + \ldots + p_n$ will be expressed as $p = \sum_{i=1}^{n} p_i$.

Now, we can already introduce the definition of binary distance-based pattern and binary distance-based generalisation operator.

DEFINITION 1. ***(Binary distance-based pattern and binary distance-based generalisation operator)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, and a set of elements $E = \{e_1, e_2\} \subset X$. We say that a pattern $p \in \mathcal{L}$ is a binary distance-based (db) pattern of E if p covers all the elements between e_1 and e_2*[2]*. Additionally, we say that Δ is a binary distance-based generalisation (dbg) operator if $\Delta(e_1, e_2)$ always computes a binary distance-based pattern.*

As previously said, for the case of more than two elements to be generalised, the concept of "nerve" of a set of elements E is needed to define non-binary *dbg* operators. Informally, a nerve of E is simply a connected[3] graph whose vertices are the elements belonging to E. Observe that if $E = \{e_1, e_2\}$, the only possible nerve is a one-edged graph. Formally,

DEFINITION 2. ***(Nerve function)*** *Let (X, d) be a metric space and let S_G be the set of undirected and connected graphs over subsets of X. A nerve function $N : 2^X \rightarrow S_G$ maps every finite set $E \subset 2^X$ into a graph $G \in S_G$, such that each element e in E is inequivocally represented by a vertex in G and vice versa. We say the obtained graph $N(E)$ is a nerve of E.*

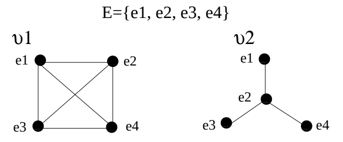

Figure 2. Two nerves for the set E. **(Left)** ν_1 is a complete graph. **(Right)** ν_2 is a 3-star graph.

Some typical nerve functions are the complete graph, and a radial/star graph around a vertex (see Figure 2).

Recall that the nerve corresponds to the notion of reachability and indicates which intermediate elements must be covered by the generalisations. In a more precise way,

DEFINITION 3. ***(Skeleton)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, a set $E \subseteq X$, and ν a nerve of E. Then, the skeleton of E wrt. ν, denoted by $skeleton(\nu)$, is defined as a set which only includes all the elements $z \in X$ between x and y, for every $(x, y) \in \nu$.*

Consequently, we look for generalisations that include the skeleton. From here, we can define the notion of distance-based pattern wrt. a nerve.

DEFINITION 4. ***(Distance-based pattern and distance-based pattern wrt. a nerve ν)*** *Let (X, d) be a metric space, $\mathcal{L}$ a pattern language, E a finite set of examples. A pattern p is a db pattern of E if there exists a nerve ν of E such that $skeleton(\nu) \subset Set(p)$. If the nerve ν is known, then we will say that p is a db pattern of E wrt. ν.*

And, from here, we have:

DEFINITION 5. ***(Distance-based generalisation operator)*** *Let (X, d) be a metric space and $\mathcal{L}$ be a pattern language. Given a generalisation operator Δ, we will say that Δ is a dbg operator if for every $E \subseteq X$, $\Delta(E)$ is a db pattern of E.*

The above definition can be characterised for one nerve function in particular.

DEFINITION 6. ***(Distance-based generalisation operator wrt. a nerve function)*** *Let (X, d) be a metric space and*

[2] Given a metric space (X, d) and two elements $e_1, e_2 \in X$, we say that an element $e_3 \in X$ is between e_1 and e_2, or is an intermediate element wrt. d, if $d(e_1, e_2) = d(e_1, e_3) + d(e_3, e_2)$

[3] Here, the term connected refers to the well-known property for graphs.

$\mathcal{L}$ a pattern language. A generalisation operator Δ is a dbg operator wrt. a nerve function N if for every $E \subseteq X$ then $\Delta(E)$ is a db pattern of E wrt. $N(E)$.

In general it is quite hard to prove that a generalisation operator is *db* wrt. any nerve function. Fortunately, for most of the applications it is enough to exist a particular nerve function wrt. Δ is distance-based. If the nerve is known beforehand, we speak of *distance-based generalisation operators wrt. a nerve function N.*

PROPOSITION 1. *Let $\mathcal{L}$ be a pattern language endowed with the operation $+$ and let Δ^b be a binary dbg operator in $\mathcal{L}$. Given a finite set of elements E and a nerve function N, the generalisation operator Δ_N defined as follows is a dbg operator wrt. N.*

$$\Delta_N(E) = \sum_{\forall (e_1, e_j) \in N(E)} \Delta^b(e_i, e_j)$$

PROOF 1. *It follows from the definition of dbg operator.*

2.2 Minimality

Given the definition of *dbg* operator in the previous section, we can now guarantee that a pattern obtained by a *dbg* operator from a set of elements ensures that all the original elements are reachable inside the pattern through intrinsic (direct) paths. However, the generalisation can contain many other, even distant, elements.

An abstract, well-founded and widely-used principle that connects the notions of fitness and simplicity is the well-known MDL/MML principle [Rissanen 1999, Wallace and Dowe 1999]. According to this principle, in our framework, the optimality of a generalisation will be defined in terms of a cost function, denoted by $k(E, p)$, which considers both the complexity of the pattern p and how well the pattern p fits E in terms of the underlying distance.

From a formal viewpoint, a cost function $k : 2^X \times \mathcal{L} \rightarrow \mathbf{R}^+ \cup \{0\}$ is a mapping where we assume that E is always finite, p is any pattern covering E and $k(E, p)$ can only be infinite when $Set(p) = X$.

As usual in MDL/MML approaches, most of the $k(E, p)$ functions will be expressed as the sum of a complexity (syntactic) function $c(p)$ (which measures how complicated the pattern is) and a fitness function $c(E|p)$ (which measures how the pattern fits the data E). As said, the most novel point here is that $c(E|p)$ will be expressed in terms of the distance employed.

As $c(p)$ measures how complex a pattern is, this function will strongly depend on the sort of data and the pattern space $\mathcal{L}$ we are dealing with. For instance, if the generalisation of two real numbers is a closed interval containing them, then a simple choice for $c(p)$ would be the length of the interval.

As $c(E|p)$ must be based on the underlying distance, a lot of definitions are based on or inspired by the well-known concept of border of a set[4]. But as the concept of border of a set is something intrinsic to metric spaces, several general definitions of $c(E|p)$ can be given independently from the datatype as shown in Table 1.

	$\mathcal{L}$	$c(E\|p)$
1	Any	$\sum_{\forall e \in E} r_e$ $r_e = inf_{r \in \mathbf{R}} B(e, r_e) \not\subset Set(p)$
2	Any	$\sum_{\forall e \in E} r_e$ $r_e = sup_{r \in \mathbf{R}} B(e, r_e) \subset Set(p)$
3	Any	$\sum_{\forall e \in E} min_{e' \in \partial Set(p)} d(e, e')$
4	$Set(p)$ is a bound set	$\sum_{\forall e \in E} min_{e' \in \partial Set(p)} d(e, e')$ $+max_{e'' \in \partial Set(p)} d(e, e'')$

Table 1. Some definitions of the function $c(E|p)$: 1-Infimum of uncovered elements, 2-Supremum of covered elements, 3-Minimum to the border, 4-Minimum and maximum to the border.

Now, we can introduce the definition of minimal distance-based generalisation operator and minimal distance-based generalisation operator relative to one nerve function.

DEFINITION 7. **(Minimal distance-based generalisation operator and minimal distance-based generalisation operator relative to one nerve function** N**)** *Let (X, d) and N be a metric space and a nerve function, and let Δ be a dbg operator wrt. N defined in X using a pattern language $\mathcal{L}$. Given a finite set of elements $E \subset X$ and a cost function k, we will say that Δ is a minimal distance-based generalisation (mdbg) operator for k in $\mathcal{L}$ relative to N, if for every dbg operator Δ' wrt. N,*

$$k(E, \Delta(E)) \leq k(E, \Delta'(E)), \text{for every finite set } E \subset X. \qquad (1)$$

In similar terms, we say that a dbg operator Δ wrt. a nerve function N is a $mdbg$ operator relative to N if the expression (1) holds for every dbg operator Δ' wrt. N.

The previous definition says nothing about how to compute the $mdbg$ operator, and as we will see later, this might be difficult. A way to proceed is to first try to simplify the optimisation problem as much as possible, as the next definition shows:

DEFINITION 8. **(Skeleton generalisation operator wrt. a nerve function** N**)** *Let (X, d) be a metric space and N a nerve function. The skeleton generalisation operator $\bar{\Delta}_N$ is defined for every set $E \subset X$ as follows:*

$$\bar{\Delta}_N(E) = argmin_{\forall p \in \mathcal{L}: skeleton(N(E)) = Set(p)} \; k(E, p)$$

which means the simplest pattern that covers the skeleton of the evidence (given a nerve) and nothing more. Clearly, it is a *dbg* operator because it includes the skeleton, but it might not exist because it cannot be expressed.

The following section is devoted to defining *db* and $mdbg$ operators for the list data type.

[4] Intuitively, if a pattern p_1 fits E better than a pattern p_2, then the border of p_1 (∂p_1) will somehow be nearer to E than the border of p_2 (∂p_2).

3. Inductive Operators for Lists

Lists or sequences is a widely-used datatype for data representation in different fields of automatic induction such as structured learning, bioinformatics or text mining. In this section, we apply our framework to finite lists of symbols by introducing two cost functions and two pattern languages for this sort of data and studying different dbg and $mdbg$ operators for each particular combination of language and cost function. Due to space limitations as well as comprehensibility's sake, we sketch those proofs that are excessively long and would make the reading unnecessarily difficult. If needed, a complete detail of them can be found in [Estruch 2008].

3.1 Metric space, pattern languages and cost functions

Several distance functions for lists have been proposed in the literature. For instance, the Hamming distance defined for equally-length lists in [Hamming 1950], or the distance in [Edgar 1990], defined for infinite-length lists but which can easily be adapted for finite lists.

However, the most widely used distance function for lists is the edit distance (or Levenshtein distance [Levenshtein 1966]), which is the one we are working with. Specifically, we set the edit distance in such a way that only insertions and deletions are allowed (a substitution can be viewed as a deletion followed by an insertion or vice-versa).

Two different pattern languages $\mathcal{L}_0$ (single-list pattern language) and $\mathcal{L}_1$ (multiple-list pattern language) will be introduced in this section. The patterns in $\mathcal{L}_0$ are lists that are built from the extended alphabet $\Sigma' = \{\lambda\} \cup \Sigma \cup V$ where λ denotes the empty list, $\Sigma = \{a, b, c, \ldots\}$ is the alphabet (also called ground symbols) from which the lists to be generalised are defined, and $V = \{V_1, V_2, \ldots\}$ is a set of variables. The same variable cannot appear twice in a pattern. Each variable in a pattern represents a symbol from $\{\lambda\} \cup \Sigma$. Finally, the pattern language $\mathcal{L}_1$ is defined from $\mathcal{L}_0$ by means of the operation $+$ (see Subsection 2.1) and aims to improve the expressiveness of $\mathcal{L}_0$. For instance, if we let $\Sigma = \{a, b\}$, then, the patterns $p_1 = aV_1V_2$ and $p_2 = bV_1V_2b$ belong to $\mathcal{L}_0$ where $Set(p_1) = \{aaa, aab, aba, abb, aa, ab, a\}$ and $Set(p_2) = \{baab, babb, bbab, bbbb, bab, bbb, bb\}$. In other words, the pattern p_1 denotes all those lists headed by the symbol a whose length ranges between 1 and 3. In a similar way, p_2 contains all the lists headed and ended by b whose length ranges between 2 and 4. Likewise, the pattern $p_3 = p_1 + p_2$ belongs to $\mathcal{L}_1$ and $Set(p_3) = Set(p_1) \cup Set(p_2) = \{aaa, aab, aba, abb, aa, ab, a, baab, babb, bbab, bbbb, bab, bbb, bb\}$.

With regard to the cost function, it is convenient to discuss some issues about the computation of the semantic cost function $c(\cdot|\cdot)$ for this particular setting. We will do this by means of an example. Suppose we are given the pattern $p = V_1V_2V_3V_4aV_5V_6V_7V_8$ and the element $e = ccaba$ which is covered by p. The computation of $c(e|p)$ is equivalent to find one of the nearest elements to e, namely e', which is not covered by p. Note that e' is not covered by p when the symbol a does not occur in e' (e.g. $e' = ccb$) or the number of symbols before or after each occurrence of a in e' is greater than 4 (e.g. $e' = ccbbbaba$). From this two possibilities, it is clear in this case that $e' = ccb$ is the nearest element to e not covered by p. This simple example allows us to affirm that the calculus $c(e|p)$ can be as complicated as determining the number of times a sequence s_1 occurs in a sequence s_2. Generally speaking, if s_p is the sequence of ground symbols in a pattern p and e' is the nearest element to e not covered by p, then e' will be a supersequence or a subsequence of e which will be obtained by modifying all the occurrences of s_p in e. Of course, as for the general form $c(E|p)$, this operation must be repeated for all the elements in E.

Therefore, if the learning problem requires the use of a cost function (e.g. because we are interested in minimal generalisations), it might be more convenient to approximate $c(E|p)$, instead of handling the original definition. For instance, we propose a naive but intuitive approximation of c inspired on the one we introduced in [Estruch 2008] for sets:

$$c'(E|p = \sum_{i=1}^{n} p_i) = \begin{cases} |E - E_1| + c(E_1|p_k), \exists p_k = V_1 \ldots V_j \\ \quad \text{and } E_1 = \{e \in E : \text{length of } e \leq j\} \\ |E|, \text{ otherwise.} \end{cases}$$

The justification is as follows. If there exists a pattern $p_k = V_1 \ldots V_j$ in p, then it is immediate that for every element e such that its length l is equal to or less than j, its nearest element not covered by p is, at least, at a distance $j - l + 1$, which is the value computed by $c(e|V_1 \ldots V_j)$. Otherwise, we assume that the nearest element of e is, at least, at a distance of 1. Implicitly, we are assuming that the nearest element to e can be obtained by removing (or adding) one specific ground symbol from (to) e.

The simplicity of $c'(\cdot|\cdot)$ will help us to study and compare the computation of the $mdbg$ in $\mathcal{L}_0$ and $\mathcal{L}_1$. As for $\mathcal{L}_0$, the cost function is directly defined as $k_0(E, p) = c'(E|p)$ (that is, the complexity of the pattern is disregarded). As for $\mathcal{L}_1$, we use $k_1(E, p) = c_1(p) + c'(E|p)$ where $c_1(p)$ measures the complexity of a pattern $p \in \mathcal{L}_1$ by counting both the ground and variable symbols in p.

3.2 Notation and previous definitions

The function $Seq(\cdot)$ defined over a pattern $p \in \mathcal{L}_0$ returns the sequence of ground symbols in p. For example, setting $p = V_1aaV_2b$, then $Seq(p) = aab$. The bar notation $|\cdot|$ denotes the length of a sequence (here a sequence can be an element, a pattern, etc.). For instance, in the previous case, $|p| = 5$. The i-th symbol in a sequence p is denoted by $p(i)$. Following with the example, $p(1) = V_1$, $p(2) = a$, $\ldots, p(5) = b$. Any sequence is indexed starting from 1. The set of all the indices of p is denoted by $I(p)$. Thus, $I(p) = \{1, 2, 3, 4, 5\}$. We sometimes use superscript as a shorthand notation to write sequences and patterns. For instance, $V^5a^3V^2$ is equivalent to $V_1 \ldots V_5aaaV_6V_7$, and

$V^2(ab)^3c$ is the same as $V_1V_2abababc$. Finally, we will often introduce mappings that are defined from one sequence to another. By $Dom(\cdot)$ and $Im(\cdot)$ we denote the domain and the image, respectively, of a mapping.

The first concept that is required is:

DEFINITION 9. **(Maximum common subsequence)** *Given a set of sequences $E = \{e_1, \ldots, e_n\}$, and according to [T.H. Cormen and Stein 2000], the maximum common subsequence (mcs, to abbreviate) is the longest (not necessarily continuous) subsequence of all the sequences in E.*

This concept is already widely used in pattern recognition. Note that the mcs of a group of sequences is not necessarily unique. The following definitions will let us work with the concept of common subsequence in a more algebraic fashion.

DEFINITION 10. **(Alignment)** *Given two elements e_1 and e_2, we say that the mapping $M_{e_2}^{e_1} : I(e_1) \to I(e_2)$ is an alignment of e_1 with e_2 if:*

i) $\forall i \in Dom(M_{e_2}^{e_1}),\ e_1(i) = e_2(M_{e_2}^{e_1}(i))$
ii) $M_{e_2}^{e_1}$ *is a strictly increasing function in* $Dom(M_{e_2}^{e_1})$.

(Remark 1) If $Dom(M_{e_2}^{e_1}) = \emptyset$, we say that $M_{e_2}^{e_1}$ is the empty alignment of e_1 with e_2. Thus, for every pair of elements we can affirm that there is always at least one alignment between them.
(Remark 2) Note that the alignment definition does not exclude the case $e_1 = e_2$.
(Remark 3) We call $e_1(i) = e_2(M_{e_2}^{e_1}(i))$ a (symbol) matching. Thus, $|Dom(M_{e_2}^{e_1})|$ (or equivalently, $|Im(M_{e_2}^{e_1})|$) is the number of matchings between e_1 and e_2 captured by $M_{e_2}^{e_1}$, and the subsequence obtained by considering the i-th symbols of e_1 where $i \in Dom(M_{e_2}^{e_1})$ is the sequence of matchings. For the sake of simplicity, we denote this sequence by $Seq(M_{e_2}^{e_1})$.

DEFINITION 11. **(Optimal alignment)** *Given two elements e_1 and e_2, if $Seq(M_{e_2}^{e_1})$ is a mcs of e_1 and e_2, then we say that $M_{e_2}^{e_1}$ is an optimal alignment.*

Since $I(e_1)$ and $I(e_2)$ are finite sets, an alignment $M_{e_2}^{e_1}$ can be written as a 2 × n matrix where n (which we denote as $Rang(M_{e_2}^{e_1})$) is the number of matchings. Hence,

$$M_{e_2}^{e_1} = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ a_{21} & \ldots & a_{2n} \end{pmatrix}$$

where $e_1(a_{1i}) = e_2(a_{2i})$ for all $1 \leq i \leq n$ (condition i) from Definition 10) and $a_{1i} < a_{1(i+1)}$ and $a_{2i} < a_{2(i+1)}$ for all $1 \leq i \leq (n-1)$ (condition ii) from Definition 10). An element of $M_{e_2}^{e_1}$ placed at row i and column j is denoted by $(M_{e_2}^{e_1})_{ij}$.

Let us illustrate all these ideas by means of an example.

EXAMPLE 1. *Given the elements $e_1 = caabbc$ and $e_2 = aacd$ where $I(e_1) = \{1,2,3,4,5,6\}$ and $I(e_2) = \{1,2,3,4\}$. An alignment $M_{e_2}^{e_1}$ (M in short) is*

$$M = \begin{pmatrix} 2 & 3 & 6 \\ 1 & 2 & 3 \end{pmatrix} \equiv \begin{array}{ccccccc} c & a & a & b & b & c & \\ & a & a & & & c & d \end{array}$$

Note that M satisfies both conditions from Definition 10. Following with M, we have that $Dom(M) = \{2,3,6\}$, $Im(M) = \{1,2,3\}$, $Rang(M) = 3$ and $Seq(M) = aac$. Finally, M is an optimal alignment.

Given that different optimal alignments can be defined over two elements e_1 and e_2, we might be interested in obtaining a concrete optimal alignment. To do this, we define a total order over all of them which lets us formally specify which optimal alignment we want.

DEFINITION 12. ***(Total order for optimal alignments)*** *Given two elements e_1 and e_2 and given the optimal alignments $M_{e_2}^{e_1}$ (M in short) and $N_{e_2}^{e_1}$ (N in short) defined as*

$$M = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ a_{21} & \ldots & a_{2n} \end{pmatrix} \quad N = \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ b_{21} & \ldots & b_{2n} \end{pmatrix}$$

we say that $M < N$ iff $(a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}) <_{LO} (b_{11}, \ldots, b_{1n}, b_{21}, \ldots, b_{2n})$ where $<_{LO}$ is the Lexicographical Order for numerical tuples.

EXAMPLE 2. *Given $e_1 = aab$ and $e_2 = ab$, we define the optimal alignments*

$$M_{e_2}^{e_1} = \begin{pmatrix} 1 & 3 \\ 1 & 2 \end{pmatrix} \quad N_{e_2}^{e_1} = \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix}$$

Then $M_{e_2}^{e_1} < N_{e_2}^{e_1}$.

Every alignment between two elements e_1 and e_2 induces a special pattern p which covers both e_1 and e_2. This pattern is unique and we call it the pattern associated to an alignment.

DEFINITION 13. **(Pattern associated to an alignment and optimal alignment pattern)** *Let e_1 and e_2 be two elements in Σ^* and let $M_{e_2}^{e_1}$ (M in short) be an alignment of e_1 with e_2. We say that a pattern $p \in \mathcal{L}_0$ is a pattern associated to the alignment M (denoted by p_M), if*
i) $Seq(M) = Seq(p)$
ii) the variable symbols in p are distributed as follows (letting $n = Rang(M)$, $l_1 = |e_1|$, $l_2 = |e_2|$):

- *The number of variables in the pattern p before the first ground symbol is equal to*

$$((M)_{11} - 1) + ((M)_{21} - 1)$$

- *The number of variables between whatever two ground symbols $p(i)$ and $p(j)$ $(i < j)$ in $Seq(p)$ such that there does not exists $i < k < j$ where $p(k)$ is a ground symbol, is equal to*

$$((M)_{1(i+1)} - (M)_{1i} - 1) + ((M)_{2(i+1)} - (M)_{2i} - 1)$$

- *The number of variables after the last ground symbol in p is equal to*

$$(l_1 - (M)_{1n}) + (l_2 - (M)_{2n})$$

If $M^{e_1}_{e_2}$ is an optimal alignment of e_1 with e_2, we say that $p_{M^{e_1}_{e_2}}$ is an optimal alignment pattern.

For instance, the pattern associated to the alignment M in Example 1 is $p_M = V_1aaV_2V_3cV_4$, which is an optimal alignment pattern because M is an optimal alignment. Note that if M is the empty alignment then $p_M = V^{l_1+l_2}$ and $Seq(M) = \lambda$.

The alignment and optimal alignment concepts (Definitions 10 and 11) can be easily extended to cope with patterns. Given two patterns p_1 and p_2, $M^{p_1}_{p_2}$ is an alignment of p_1 with p_2 where only matchings between ground symbols are taken into account, that is, $\forall i \in Dom(M^{p_1}_{p_2}), p_1(i) = p_2(M^{p_1}_{p_2}(i))$, $p(i) \in \Sigma$ and $p_2(M^{p_1}_{p_2}(i)) \in \Sigma$. Analogously, $M^{p_1}_{p_2}$ is an optimal alignment if $Seq(M^{p_1}_{p_2})$ is a msc of p_1 and p_2.

To conclude, we introduce a binary bottom-up generalisation operator (called $\uparrow$-transformation) defined over $\mathcal{L}_0$, which allows us to move through the pattern language.

DEFINITION 14. *Given two patterns p_1 and p_2 in $\mathcal{L}_0$ we define the binary mapping*

$$\begin{array}{llll}\uparrow(\cdot,\cdot): & \mathcal{L}_0 \times \mathcal{L}_0 & \rightarrow & \mathcal{L}_0 \\ & (p_1, p_2) & \rightarrow & \uparrow(p_1,p_2) = p, \quad \textit{such that}\end{array}$$

1. *Let $M^{p_1}_{p_2}$ (M in short) be the minimum optimal alignment of p_1 with p_2, then $Seq(p) = Seq(M)$.*
2. *If $Seq(M) = \lambda$ then $p = V^{max\{|p_1|,|p_2|\}}$. Otherwise, the distribution of the variables in p is:*
   - *Before the first ground symbol in p, the number of variable is equal to:*

   $$max\{(M)_{11} - 1, (M)_{21} - 1\}$$

   - *Between two consecutive ground symbols in p, the number of variables is equal to:*

   $$max\{(M)_{1(i+1)} - (M)_{1i} - 1, (M)_{2(i+1)} - (M)_{2i} - 1\}$$

   - *After the last ground symbol in p, the number of variables is equal to (letting $n = Rang(M)$, $l_1 = |p_1|$ and $l_2 = |p_2|$):*

   $$max\{l_1 - (M)_{1n}, l_2 - (M)_{2n}\}$$

EXAMPLE 3. *Given the patterns $p_1 = abcV_1$, $p_2 = V_1abcccV_2$ and $p_3 = dV_1$, then $\uparrow(p_1,p_2) = VabcV^3$ and $\uparrow(p_1,p_3) = V^4$.*

PROPOSITION 2. *For every pair of patterns p_1 and p_2 in $\mathcal{L}_0$, if $p = \uparrow(p_1,p_2)$ then $Set(p_1) \subset (p)$ and $Set(p_2) \subset (p)$.*

PROOF 2. *It directly comes from the definition of the $\uparrow$-transformation.*

Next, we explain how to define dbg operators for the different pattern languages, and we study the possibility of finding $mdbg$ operators for $(\mathcal{L}_0, k_0)$ and $(\mathcal{L}_1, k_1)$.

3.3 Single list pattern language ($\mathcal{L}_0$)

One would expect that if $\Delta(E)$ computes a pattern p such that $Seq(p)$ is a mcs of the lists in E, then $\Delta(\cdot)$ is a dbg operator. However, we find that this operator is not, in general, distance-based. The following example illustrates this:

EXAMPLE 4. *Let $E = \{e_1, e_2, e_3\}$ where $e_1 = c^5a^3b^3$, $e_2 = c^5a^2d^4$ and $e_3 = a^3b^3d^4c^5$ are the elements to be generalised. Initially, we are going to fix a nerve for these elements, namely, the complete nerve (see Figure 3).*

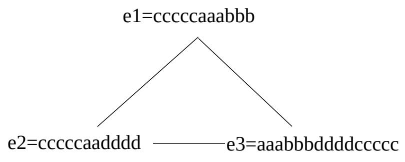

Figure 3. A complete nerve ν for the evidence $E = \{e_1, e_2, e_3\}$.

The pattern $p = V^{10}c^5V^6$ generalises E, and $Seq(p)$ is a mcs of the lists in E. However, this pattern is not a db pattern of E since, for example, the element a^3b^3 (which is between e_1 and e_3) and the element a^2d^4 (which is between e_2 and e_3) are not covered by p. As a matter of fact, no pattern containing the ground symbol c will be db and this result is independent of the nerve chosen.

The explanation for this apparently counterintuitive result is based on how the distance between the different pairs of elements e_i and e_j is calculated. In fact, although all the lists in E have subsequence c^5 in common, this subsequence is never taken into account to compute the distance $d(e_i, e_j)$, for any pair (e_i, e_j) in ν. Therefore, the operator definition we propose next not only uses the concept of mcs but also uses others such as the $\uparrow$-transformation and the concept of nerve which ensures the condition of being db. First, we deal with the binary generalisation operator, and then we extend it for the n-ary case.

In the first stage, for any two elements e_1 and e_2 to be generalised, we need to somehow find out which patterns in $\mathcal{L}_0$ can cover those elements between e_1 and e_2.

PROPOSITION 3. *Given the elements e_1, e_2 and e, if e is between e_1 and e_2, then there exists an optimal alignment pattern p associated to an optimal alignment of e_1 and e_2 such that $e \in Set(p)$.*

PROOF 3. ***(Sketch)** Let $M^{e_1}_{e}$ and $M^{e}_{e_2}$ be the optimal alignments of e_1 with e and e with e_2, respectively. We define the mapping M between e_1 and e_2 as the composition of*

$M_e^{e_1}$ and $M_{e_2}^e$. The goal is to prove first that M is an optimal alignment of e_1 with e_2 and then, see that the associated pattern p_M covers e. For this last step we distinguish two cases: i) M is the empty alignment and consequently $p_M = V^{|e_1|+|e_2|}$. According to Proposition 21 in [Estruch 2008], if e is between e_1 and e_2, then $|e| \leq |e_1| + |e_2|$, hence $e \in Set(p_M)$. ii) M is not empty and we aim to prove that the variable symbols in M are distributed in such a way that we can ensure that $e \in Set(p_M)$.

We will use the proposition above along with the $\uparrow$-transformation to define binary *db* operators.

COROLLARY 1. *Given the elements e_1 and e_2, if $\{p_i\}_{i=1}^n$ is the set of all the optimal alignment patterns of e_1 and e_2, then the generalisation operator defined as follows is db.*

$$\Delta^b(e_1, e_2) = \uparrow (p_1, \uparrow (p_2, \dots \uparrow (p_{n-1}, p_n)) \dots)$$

PROOF 4. *For every optimal alignment pattern, we know from Proposition 2, that*

$$Set(p_i) \subset Set(\Delta^b(e_1, e_2)) \quad (2)$$

Then, from Proposition 3, we can write that

$$\forall \text{ element } e \text{ between } e_1 \text{ and } e_2 \Rightarrow \exists p_i : e \in Set(p_i) \quad (3)$$

Now, combining (2) and (3), we can affirm that

$$\forall \text{ element } e \text{ between } e_1 \text{ and } e_2 \Rightarrow e \in Set(\Delta^b(e_1, e_2)) \quad (4)$$

Hence, the generalisation operator is distance-based.

Next, we extend Corollary 1 for an arbitrary number of elements.

COROLLARY 2. *Given a finite set of elements $E \subset X$ and a function nerve N, the generalisation operator Δ defined in Algorithm 1 (where Δ^b is defined in Corollary 1) is db wrt. N.*

PROOF 5. *For every $(e_i, e_j) \in N(E)$, $Set(\Delta^b(e_i, e_j)) \subset Set(\Delta(E))$ by the definition of the $\uparrow$-transformation. Therefore, for every finite set E, $\Delta(E)$ is distance-based w.r.t. $N(E)$.*

Algorithm 1 returns a pattern p such that $Set(\Delta^b(e_i, e_j)) \subset Set(p)$, for every pair of elements in $N(E)$, by iteratively applying the $\uparrow$-transformation over all the patterns $\Delta^b(e_i, e_j)$. The *else*-block is important since it ensures that $Seq(p) \neq \lambda$, if all the sequences $Seq(\Delta^b(e_1, e_j))$ have a subsequence in common. Let us see an example of this.

EXAMPLE 5. *Given $E = \{e_1, e_2, e_3, e_4\}$ where $e_1 = abc$, $e_2 = cabcd$, $e_3 = c$, $e_4 = cab$ and the nerve $N(E) = \{(e_1, e_2), (e_2, e_3), (e_2, e_4)\}$. The binary distance-based generalisations (lines 5-7 in the algorithm) are:*

$$\begin{array}{lllll} L[0] & = & \Delta^b(e_1, e_2) & = & VabcV \\ L[1] & = & \Delta^b(e_2, e_3) & = & VcabV \\ L[2] & = & \Delta^b(e_2, e_4) & = & V^3cV^4 \end{array}$$

```
Data: $E = \{e_1, \dots, e_n\}$, $\Delta^b$ (binary $dbg$ operator) and
      $\nu$ (a nerve of $E$)
Result: Distance-based pattern of $E$ wrt. $\nu$
1  begin
2      $k \leftarrow 0$;
3      $L \leftarrow [] /* empty\ list */$;
4      for $(e_i, e_j) \in N(E)$ do
5          $L[k] \leftarrow \Delta^b(e_i, e_j)$;
6          $k \leftarrow k + 1$;
7      end
8      $S \leftarrow \{a_i \in \Sigma : \forall 0 \leq j \leq k : a_i \in Seq(L[j])\}$;
9      if $S = \emptyset$ then return $V^{max\{|L[j]| : \forall 0 \leq j \leq k\}}$ ;
10     else
11         $p \leftarrow First(L)$;
12         $Remove(L, p)$;
13         while $L \neq \emptyset$ do
14             Find $p_i \in L$: $\exists a_j \in S, a_j \in Seq(\uparrow (p, p_i))$;
15             $p \leftarrow \uparrow (p, p_i)$;
16             $Remove(L, p_j)$;
17         end
18         return $p$;
19     end
20 end
```

Algorithm 1: An algorithm to compute a *db* pattern of a set of lists E wrt. a nerve ν.

If we applied the $\uparrow$-transformation in any arbitrary order over the set of binary patterns, we could obtain for example:

$$\begin{array}{lll} p & \leftarrow & VabcV \\ p & \leftarrow & \uparrow (p, VcabV) = V^2abV^2 \\ p & \leftarrow & \uparrow (p, V^3cV^4) = V^9 \end{array}$$

However, if the $\uparrow$-transformation is applied as the algorithm indicates (lines 8-17), then $S = \{c\}$ and the patterns would be merged in the following order:

$$\begin{array}{lll} p & \leftarrow & \uparrow VabcV \\ p & \leftarrow & \uparrow (p, V^3cV^4) = V^3cV^4 \\ p & \leftarrow & \uparrow (p, VcabV) = V^3cV^4 \end{array}$$

With regard to the computation of $mdbg$ operators in $(\mathcal{L}_0, k_0)$, the algorithm above always return the $mdbg$. On the one hand, if all the binary patterns have a subsequence in common, the algorithm computes a distance-based pattern p such that $Seq(p) \neq \lambda$ and the function $c'(E|p) = |E|$ which attains a minimum value. On the other hand, the algorithm returns a pattern with variable symbols only, and whose length is the minimum length required to be distance-based. Therefore, p is minimal as well.

3.4 Multiple list pattern language ($\mathcal{L}_1$)

We will define *dbg* operators in $\mathcal{L}_1$ via Δ_N (Proposition 1). The binary operator Δ^b required by Δ_N is the one introduced in Corollary 1. An example of how this operator works is shown below:

EXAMPLE 6. *Given a finite set of elements $E = \{e_1, e_2, e_3, e_4\}$ where $e_1 = a^2b^2d$, $e_2 = da^2c^2$, $e_3 = c^2db^2$ and $e_4 = ad$ and the nerve $N(E) = \{(e_1, e_2), (e_1, e_3), (e_1, e_4)\}$.*

$$\begin{array}{rcccl}\Delta^b(e_1, e_2) & = & p_1 & = & Va^2V^5 \\ \Delta^b(e_1, e_3) & = & p_2 & = & V^5b^2 \\ \Delta^b(e_1, e_4) & = & p_3 & = & VaV^3d\end{array}$$

Finally,

$$\Delta_N(E) = Va^2V^5 + V^5b^2 + VaV^3d$$

Observe that the solution for this example in $\mathcal{L}_0$ is just a pattern consisting of variable symbols only, which shows the utility of $\mathcal{L}_1$. Next, let us see how to obtain $mdbg$ operators in $\mathcal{L}_1$.

Since the only way we know to define a distance-based operator in $\mathcal{L}_1$ consists in fixing a nerve beforehand, it is reasonable to study how to define $mdbg$ operators relative to a nerve function. However, the calculus of the $mdbg$ operator is not easy at all. Basically, the question is whether the $mdbg$ operators relative to a nerve function N can be defined in terms of Δ_N and the $\uparrow$-transformation. However, this result seems hard to be established. On the one hand, we ignore how to explicitly define most of the Δ^b operators (since Corollary 1 only establishes a sufficient condition) and on the other hand, we must take into consideration some inherent limitations of the $\uparrow$-transformation:

1. The mdb pattern might not be found by applying the $\uparrow$-transformation over Δ_N if this one uses the binary operator Δ^b defined in Corollary 1: we will illustrate this by means of an example.

   EXAMPLE 7. *Given the set $E = \{e_1, e_2, e_3\}$, where $e_1 = a_1a_2a_3$, $e_2 = a_1a_6a_7$ and $e_3 = a_2a_4a_5$, and $N(E) = \{(e_1, e_2), (e_1, e_3)\}$. The optimal alignment patterns which are associated to (e_1, e_2) and (e_1, e_3), respectively, are a_1V^4 and Va_2V^3. Then a_1V^4 is a db pattern of (e_1, e_2) (since it is the only optimal alignment pattern) and Va_2V^3 is a db pattern of (e_2, e_3) (since it is the only optimal alignment pattern). Hence, the pattern $p = a_1V^4 + Va_2V^3$ is db w.r.t. $N(E)$. However, the pattern $p' = a_1V^4 + a_2V^3$ is distance-based (the only element between e_1 and e_2, which is not covered by a_2V^3, is $a_1a_2a_4a_5$ but this is covered by a_1V^4) but $Set(p') \not\subset Set(p)$. The mdb pattern for E will have $|p'|$ or even fewer symbols and this will never be achieved by the $\uparrow$-transformation over the optimal alignment patterns.*

   Therefore, given that Δ^b is defined from the concept of optimal alignment patterns and Δ_N is defined from Δ^b, it is not possible that the $mdbg$ operator can be expressed in terms of the $\uparrow$-transformation and Δ_N.

2. The $mdbg$ pattern might not be found by applying the $\uparrow$-transformation over $skeleton(N(E))$: from the previous point, we could think that the mdb pattern cannot be found because the optimal alignment patterns are excessively general. However, if it was so, it would mean that starting the search from something extremely specific, namely the skeleton, the mdb pattern should be found. However, this is not true as the next example reveals:

   EXAMPLE 8. *Given $E = \{e_1, e_2, e_3, e_4, e_5\}$ where $e_1 = ac^3b^2$, $e_2 = ab^2$, $e_3 = ab^2ce$, $e_4 = d$ and $e_5 = fgh$ and the nerve depicted below:*

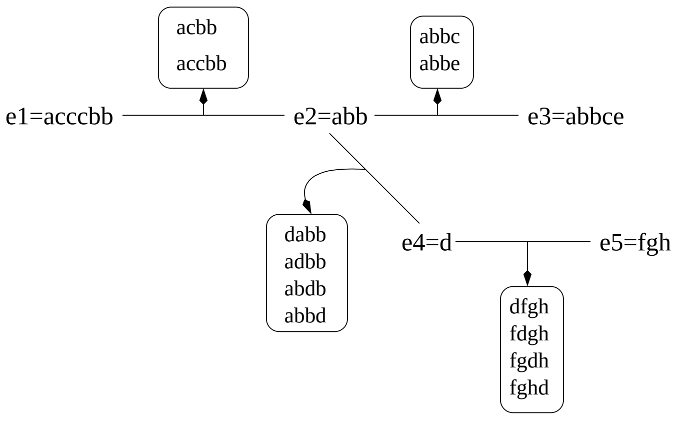

Figure 4. A naive generalisation of the set E w.r.t. the nerve N(E). Circled elements are the intermediate elements.

If we group the elements according to its similarity and then apply the $\uparrow$-transformation over the different groups, the pattern obtained would attain a lower value for $k_1(E, \cdot)$. Taking this strategy into account, we can distinguish several meaningful grouping criteria. For instance, those elements which contain the subsequence abb (G_1) and those which do not (G_2). That is,

$$\begin{array}{rcl}G_1 & = & \{ac^3b^2, acb^2, ac^2b^2, ab^2, \ldots, ab^2d\} \\ G_2 & = & \{dfgh, fdgh, fgdh, fghd\}\end{array}$$

In this particular case, it does not matter how the elements in the groups are ranked in order to apply the $\uparrow$-transformation since the final result remains invariable. Thus, we can write

$$p_1 = \uparrow(G_1) + \uparrow(G_2) = VaV^3bVbV^2 + VfVgVhV$$

For any other binary splitting, we would have elements having no subsequence in common in the same group (e.g. abb and $dfgh$). The shortest patterns would be

$$\begin{array}{rcl}p_2 & = & aV^3b^2V^2 + V^4 \\ p_3 & = & V^6\end{array}$$

Using three groups, another interesting possibility can be explored. For instance, $G_1 = \{fgh\}$, those elements containing the subsequence d (G_2) and the remaining ones

(G_3). Depending on the order of the elements in G_2 we could obtain by applying the $uparrow$-transformation.

$$\begin{aligned} p_4 &= V^5 + aV^3b^2V^2 \\ p_5 &= V^3dV^3 + aV^3b^2V^2 + fgh \end{aligned}$$

Finally, it is not worth using more than three groups because of the excessive length of the pattern obtained. Evaluating the different patterns, we have that:

$$\begin{aligned} k_1(E,p_1) = c(p_1) + c'(E|p_1) = 17 + 5 = 22 \\ k_1(E,p_2) = c(p_2) + c'(E|p_2) = 12 + 10 = 22 \\ k_1(E,p_3) = c(p_3) + c'(E|p_3) = 6 + 17 = 23 \\ k_1(E,p_4) = c(p_4) + c'(E|p_4) = 13 + 13 = 26 \\ k_1(E,p_5) = c(p_5) + c'(E|p_5) = 18 + 5 = 23 \end{aligned}$$

But the following patterns are also distance-based for E:

$$\begin{aligned} p_6 &= V^3cV^2 + V^4 \\ p_7 &= aV^5 + V^4 \end{aligned}$$

where

$$\begin{aligned} k_1(E,p_6) = c(p_6) + c'(E|p_6) = 10 + 10 = 20 \\ k_1(E,p_7) = c(p_7) + c'(E|p_7) = 10 + 10 = 20 \end{aligned}$$

However, neither p_6 nor p_7 can be derived from a $\uparrow$-transformation since this tends to extract the longest common subsequence. Observe that all the elements which have the subsequence c or a also contain the subsequence abb in common.

From this previous analysis, we can conclude that the $\uparrow$-transformation is not enough in itself to explore the search space. We need a generalisation tool which is not based on the concept of the longest common subsequence. For this purpose, we introduce the so-called inverse substitution.

DEFINITION 15. ***(Inverse substitution)*** *Given a pattern p in $\mathcal{L}_0$ or in $\mathcal{L}_1$ an inverse substitution σ^{-1} is a set of indices where each index denotes a ground symbol in p to be changed by a variable. Thus, $p\sigma^{-1}$ represents the new pattern which is obtained by applying σ^{-1} over p.*

Basically, an inverse substitution just changes ground symbols by variables. For example, given $p = VaabV$ and $\sigma^{-1} = \{2, 4\}$ then $p\sigma^{-1} = V^2aV^2$. Now, we are in conditions to introduce the next proposition:

PROPOSITION 4. *Given a finite set of elements $E = \{e_1, \ldots, e_n\}$ and a nerve function N. If we set $S = skeleton(N(E))$ then there exists a partition P of the set S and a collection of inverse substitutions $\{\sigma_1^{-1}, \ldots, \sigma_n^{-1}\}$ such that the pattern*

$$p = \sum_{\forall P_i = \{e_{k_i}\}_{k_i=1}^{m_i} \in P} \uparrow (\{e_{k_i}\sigma_{k_i}^{-1}\}_{k_i=1}^{m_i})$$

is a mdb pattern of E relative to $N(E)$.

PROOF 6. ***(Sketch)***. *We can assume that there exists a pattern $p = \sum_{i=1}^{n} p_i$ such that $k(E,p)$ attains a minimum value. The pattern p induces a partition of $E = \cup E_i$ in such a way that $e_i \in E_i$ iff $e_i \in Set(p_i)$. Next, we remove repeated elements in the different E_i in order to make sure that the subsets E_i are pairwise disjoints. Finally, the proposition can be proved using the concepts of inverse substitution and $\uparrow$-transformation over the partition we have set.*

This latter proposition leads to an exhaustive search algorithm in order to compute the $mbdg$ operator. This algorithm turns out to be useless in general due to the size of the search space (the number of different possibilities for the partition of $skeleton(N(E))$ and substitutions). In fact, for a particular version of $\mathcal{L}_1$, we have proved that this optimisation problem is NP-Hard (see [Estruch 2008]).

Hence, the other option is to approximate the calculus of the mdb patterns. To do this, we use a greedy search schema driven by the cost function. That is, for each iteration, the $\uparrow$-transformation is applied over the pair of patterns that reduces th cost function most. This idea is formalised in the Algorithm 2 and illustrated in Example 9.

```
Input: $E = \{e_1, \ldots, e_n\}$, $\Delta^b$ (binary $dbg$ operator)
       and $N$ (nerve function)
Output: A pattern which approximates a $mdb$ pattern
        of $E$ w.r.t. $N(E)$
1  $\tilde{\Delta}_N(E)$
2  begin
3      $k \leftarrow 1$;
4      for $(e_i, e_j) \in N(E)$ do
5          $p_k \leftarrow \Delta^b(e_i, e_j)$;
6          $k \leftarrow k + 1$;
7      end
8      $p = \sum_{k=1}^{n} p_k$;
9      do
10         $k_p \leftarrow k_1(E, p)$;
11         $p' \leftarrow argmin\{k_1(E, p_{ij}) : \forall 1 \leq i, j, \leq$
           $n, p_{ij} = \uparrow (\{p_i, p_j\}) + (p - p_i - p_j)\}$;
12         $k'_p \leftarrow k_1(E, p')$;
13         if $k_{p'} < k_p$ then $p \leftarrow p'$;
14     while $k_{p'} < k_p$
15     return $p$;
16 end
17 //The notation $p - p_i - p_j$ employed in the algorithm
   means all the patterns in $p$ except $p_i$ and $p_j$.;
```

Algorithm 2: A greedy algorithm which approximates the $mdbg$ operator.

EXAMPLE 9. *Let E and $N(E)$ be the set of examples and the nerve employed in Example 6. Remember that,*

$$\begin{array}{rcl} p_1 = \Delta^b(e_1, e_2) & = & Va^2V^5 \\ p_2 = \Delta^b(e_1, e_3) & = & V^5b^2 \\ p_3 = \Delta^b(e_1, e_4) & = & VaV^3d \end{array}$$

and

$$p = Va^2V^5 + V^5b^2 + VaV^3d$$

see lines 4-8 *in the algorithm. Next, we have to apply the* $\uparrow$*-transformation over each pair of binary generalisations and we choose the one which attains a lower value of* $k_1(E, \cdot)$ *(see lines* 9-14*). In our case, we must consider two possibilities:*

$$\begin{array}{rclcl} p_1 & = & \uparrow (Va^2V^5, V^5b^2) + VaV^3d & = & V^8 + VaV^3d \\ & = & V^8 & & \\ p_2 & = & \uparrow (Va^2V^5, VaV^3d) + V^5b^2 & = & VaV^6 + V^5b^2 \end{array}$$

Since $k_1(E, p_2) = 19$ *is less than* $k_1(E, p_1) = 27$*, we choose the pattern* p_2*. The process stops when the pattern cannot be further improved. Note that the next iteration leads to*

$$\uparrow (VaV^6, V^5b^2) = V^8$$

which performs worse than p_2*. Therefore, the algorithm returns* p_2*.*

4. Conclusions and Future Work

This work is based in our approach for a correct integration of distance-based methods with symbolic inductive learners [Estruch et al. 2005, 2006]. This proposal relies on the novel concept of (minimal) distance-based generalisation operator, which aims to induce consistent (minimal) patterns from data embedded in a metric space.

However, the main contribution that we present here, consists in studying how to apply our framework in order to infer consistent symbolic patterns from a particular structured data type (lists) and a distance function (edit distance). More concretely, we have seen how to define (minimal) distance-based generalisation operators for this domain.

To do this, we have introduced two different pattern languages $\mathcal{L}_0$ and $\mathcal{L}_1$. The first language is made up of patterns which consist of finite sequences of ground and variable symbols. The language $\mathcal{L}_1$ extends $\mathcal{L}_0$ in that the disjunction of patterns is permitted. Additionally, we have defined a cost function for each language in order to study the minimality of the patterns we can obtain.

We have proved that for more than two sequences, the widely-used concept of *maximum common subsequence* does not necessarily lead to distance-based generalisation operators. In order to obtain this sort of operators, we need to introduce a new concept: namely, the concept of sequence associated to an optimal alignment. This kind of sequences leads to certain patterns that when combined, allows us to define distance-based operators. As for the minimality of these operators, we have shown this is a computational hard problem in $\mathcal{L}_1$. For this reason, we have introduced a greedy search algorithm which allows us to approximate minimal generalisations.

Further work refers to the following questions. One is about the computational complexity of the greedy search algorithm which approximates minimal patterns. This has a quadratic complexity with the number of subpatterns in the pattern obtained by Proposition 1. Unfortunately, this operation still has a high cost, if we want to run our algorithm over large data sets. Thus, it would be convenient to try other heuristics with a lower complexity but that ensure a good approximation. Another one is devoted to the pattern languages that have been investigated. Note that both $\mathcal{L}_0$ and $\mathcal{L}_1$ are subfamilies of the regular languages. A very interesting line of work would consist in extending all the results presented in this paper in order to include pattern representations based on other more expressive subfamilies of regular languages. By doing this, we could obtain not only new grammar inference algorithms but also new grammar learners that would ensure the consistency of the inferred model wrt. the underlying distance, something which does not happen when traditional grammar learners are applied.

References

A.F. Bowers, C. G. Giraud-Carrier, and J. W. Lloyd. Classification of individuals with complex structure. In *Proc. of the 17th International Conference on Machine Learning (ICML'00)*, pages 81–88. Morgan Kaufmann, 2000.

G. A. Edgar. *Measure, Topology and Fractal Geometry*. Springer-Verlag, 1990.

V. Estruch. Bridging the gap between distance and generalisation: Symbolic learning in metric spaces. PhD Thesis, DSIC-UPV `http://www.dsic.upv.es/ vestruch/thesis.pdf`, 2008.

V. Estruch, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Distance based generalisation. In *Proc. of the 15th Int. Conf. on ILP*, volume 3625 of *LNCS*, pages 87–102, 2005.

V. Estruch, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Minimal distance-based generalisation operators for first-order objects. In *In Proc. of the 16th Int. Conf. on ILP*, pages 169–183, 2006.

C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In H. Kuchen and K. Ueda, editors, *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2001. ISBN 3-540-41739-7.

R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal.*, 26(2):147–160, 1950.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady.*, 10:707–710, 1966.

J. W. Lloyd. Learning comprehensible theories from structured data. In *Advanced lectures on machine learning*, pages 203–225. Springer-Verlag, 2003.

S. H. Muggleton. Inductive logic programming: Issues, results, and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296, 1999.

R. Olsson. Inductive functional programming using incremental program transformation. *Artifificial Intelligence*, 74(1):55–81, 1995. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(94)00042-Y.

J. Rissanen. Hypothesis selection and testing by the MDL principle. *The Computer Journal*, 42(4):260–269, 1999.

U. Schmid. *Inductive synthesis of Functional Programs-Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*. Springer, 2003.

S.H. Swamidass, J. Chen, J. Bruand, P. Phung, L. Ralaivola, and P. Baldi. Kernels for small molecules and the prediction of mutagenecity, toxicity and anti-cancer activity. *Bioinformatics*, 21:359–368, 2005.

R. Rivest T.H. Cormen, C. Leiserson and C. Stein, editors. *Introduction to Algorithms*. The MIT Press, 2000.

C. S. Wallace and D. L. Dowe. Minimum Message Length and Kolmogorov Complexity. *Computer Journal*, 42(4):270–283, 1999.

Porting IgorII from MAUDE to HASKELL

Introducing a System's Design

Martin Hofmann Emanuel Kitzelmann Ute Schmid

Cognitive Systems Group, University of Bamberg

{martin, emanuel, ute}.{hofmann, kitzelmann, schmid}@uni-bamberg.de

Abstract

This paper describes our efforts and solutions in porting our IP system IGOR 2 from the termrewriting language MAUDE to HASKELL. We describe how, for our purpose necessary features of the homoiconic language MAUDE can be simulated in HASKELL using a stateful monad transformer. With our new implementation we are now able to use higher-order context during our synthesis and extract information from type classes useable as background knowledge. Keeping our new implementation as close as possible to our old, we could keep all features of our system.

Keywords Inductive Programming, Homoiconic Languages, MAUDE, HASKELL, IGOR 2, System Design

1. Introduction

Inductive programming (IP) dares to tackle a problem as old as programming itself: Help the human programmers with their task of creating programs, solely using evidence of an exemplary behaviour of the desired program. Contrary to deductive program synthesis, where programs are generated from an abstract, but complete specification, inductive program synthesis is concerned with the synthesis of programs or algorithms from incomplete specifications, such as input/output (I/O) examples. Focus is on the synthesis of *declarative*, i.e., logic, functional, or functional logic programs. The aims of IP are manyfold. On the one hand, research in IP provides better insights in the cognitive skills of human programmers. On the other hand, powerful and efficient IP systems can enhance software systems in a variety of domains—such as automated theorem proving and planning—and offer novel approaches to knowledge based software engineering such as model driven software development or test driven development, as well as end user programming support in the XSL domain [Hofmann 2007].

Beginnings of IP research addressed inductive synthesis of functional programs from small sets of positive I/O examples only [Biermann et al. 1984]. One of the most influential classical systems was THESYS [Summers 1977] which synthesised linear recursive LISP programs by rewriting I/O pairs into traces and folding of traces based on recurrence detection. Currently, induction of functional programs is covered by the analytical approaches IGOR 1 [Kitzelmann and Schmid 2006], and IGOR 2 [Kitzelmann 2007] and by the evolutionary/generate-and-test based approaches ADATE [Olsson 1995] and MAGICHASKELLER [Katayama 2007].

Analytical approaches work example-driven, so the structure of the given I/O pairs is used to guide the construction of generalised programs. They are typically very fast and can guarantee certain characteristics for the generated programs such as minimality of the generalisation w.r.t. to the given examples and termination. However they are restricted to programs describeable by a small set of I/O pairs.

Generate-and-test based approaches first construct one or more hypothetical programs, evaluate them against the I/O examples and then work on with the most promising hypotheses. They are very powerful and usually do not have any restrictions concerning the synthesisable class of programs, but are extremely time consuming.

Two decades ago, some inductive logic programming (ILP) systems were presented with focus on learning recursive logic *programs* in contrast to learning classifiers: FFOIL [Quinlan 1996], GOLEM [Muggleton and Feng 1990], PROGOL [Muggleton 1995], and the interactive system DIALOGS [Flener 1996]. Synthesis of functional logic programs is covered by the system FLIP [Hernández-Orallo and Ramírez-Quintana 1998].

IP can be viewed as a special branch of machine learning because programs are constructed by inductive generalisation from examples. Therefore, as for classification learning, each approach can be characterised by its restriction and preference bias [Mitchell 1997]. However, IP approaches cannot be evaluated with respect to some covering measure or generalisation error since (recursive) programs must treat *all* I/O examples correctly to be an acceptable hypothesis.

The task of writing programs writing programs—pardon the pun—is *per se* reflexive, so it is virtually self-suggesting to use reflexive, also called homoiconic languages. Unfortunately only a few homoiconic languages are declarative and adequate for IP, e.g. LISP and MAUDE. Nevertheless, they lack interesting features like polymorphic types with type classes or higher-order functions. State-of-the-art functional languages with a large community and good library support as e.g. HASKELL do not provide reflexive features, though.

Nevertheless, we value the pros of a state-of-the-art functional language more and so grasp the nettle and build our own homoiconic support. This paper describes our efforts and solutions in porting our IP system IGOR 2 from the termrewriting language MAUDE to HASKELL facing problems in simulating reflexive properties. This is done mainly to overcome MAUDE's restricted higher-order context, but also to use information about type classes as background knowledge. IGOR 2's key features are kept unchanged. They are

- termination by construction,
- handling arbitrary user-defined data types,
- utilisation of arbitrary background knowledge,
- automatic invention of auxiliary functions as sub programs,
- learning complex calling relationships (tree- and nested recursion),
- allowing for variables in the example equations,
- simultaneous induction of mutually recursive target functions.

Furthermore it provides insights in less theoretical but more pragmatic implementation details of the systems. The next Section 2 gives an overview of the theory behind IGOR 2 and its strong linkage to MAUDE, and in Section 3 we describe the library specification of our new implementation in HASKELL. We conclude with an outlook on future work in Section 5.

2. IGOR 2 and MAUDE

IGOR 2's [Kitzelmann 2008] main objective is to overcome the strong limitations—only a small fixed set of primitives and no background knowledge, strongly restricted program schemas, linearly ordered I/O examples—of the classical analytical approach but not for the price of a generate-and-test search. This is realized by integrating analytical techniques into a systematic search in the program space. A prototype is implemented in MAUDE.

2.1 The IGOR 2-Algorithm

We only sketch the algorithm here. For a more detailed description see [Kitzelmann 2008].

IGOR 2 represents I/O examples, background knowledge, and induced programs as *constructor (term rewriting) systems (CSs)* over many-sorted (typed) first-order signatures. Signatures for CSs are the union of two disjoint subsignatures called *defined function symbols* and *constructor symbols*, respectively. Terms containing only constructor symbols (and variables) are called *constructor terms*. A CS is then a set of directed equations or rules of the form $F(p_1, \ldots, p_n) \to t$ where F is a defined function symbol and the p_i are constructor terms. This corresponds to *pattern matching* over user-defined datatypes in functional programming. A CS is evaluated by term rewriting. Terms that are not rewritable—these include, in particular, all constructor terms—are called *normal forms*. For CSs representing I/O examples or background knowledge hold the additional restriction that right-hand sides (rhss) are constructor terms. This particularly means that also background knowledge must be provided in an extensional form.

In order to construct *confluent* CSs, i.e., CSs with *unique* normal forms, IGOR 2 assures that patterns of rules belonging to one defined function are *disjoint*, i.e., do not unify. IGOR 2's inductive bias is—roughly speaking—to prefer CSs with fewer disjoint patters, i.e., CSs that partition the domain into fewer subsets. With respect to this preference bias, IGOR 2 starts with one *initial rule* per target function. An initial rule is the *least general generalisation*—with respect to the subsumption order $t \geq t'$ (t *subsumes* or *is more general than* t'), if there exists a substitution σ with $t\sigma = t'$—of the provided I/O examples. Initial rules entail the I/O examples with respect to equational reasoning and are *correct with respect to the I/O examples* in this sense. However, an initial rule may contain variables in its right-hand side (rhs) not occurring in its left-hand side (lhs), i.e. pattern. We call such variables *unbound* and rules and their rhs containing them, *open*. Unbound variables may be instantiated arbitrarily within rewriting such that CSs containing open rules do not represent *functions*. Hence, CSs are transformed during the search by taking an open rule r out of a CS and replacing it by a set of new rules R such that (i) either the unbound variables are eliminated in the rhs of r in R or r is completely discarded from R, and (ii) the resulting CS is still correct with respect to the I/O examples and equational reasoning. Different sets R may be possible as replacements for an open rule, i.e., a refinement operator takes an open rule r and yields a set of sets R of rules. In one search step, an open and best rated CS with respect to the preference bias and one open rule from it is chosen. Then all refinement operators are applied to r yielding a set of sets of rules each. The union of these sets is the set of possible replacements R of r. Now r is replaced in each CS containing it by each possible R. A goal state is reached if all best rated CSs are closed. This set constitutes the solution returned by IGOR 2.

There are three transformation operators: (i) The I/O examples belonging to the open initial rule are partitioned into subsets and for each subset, a new initial rule (with a more specific pattern than the original rule) is computed. (ii) The open rhs is replaced by a (recursive) call to a defined function. The arguments of the call may again contain calls to defined functions. Hence, computing the arguments is considered as a new subproblem. (iii) If the open rhs has a constructor as root, i.e., does not consist of a single unbound variable, then all subterms containing unbound variables are treated as subproblems. A new auxiliary function is introduced for each such subterm.

Splitting an open rule. The first operator partitions the I/O examples belonging to a rule into subsets such that the patterns of the resulting initial rules are disjoint more specific than the pattern of the original rule. Finding such a partition is done as follows: A position in the pattern p with a variable resulting from generalising the corresponding subterms in the subsumed example inputs is identified. This implies that at least two of the subsumed inputs have different constructor symbols at this position. Now all subsumed inputs are partitioned such that all of them with the same constructor at this position belong to the same subset. Together with the corresponding example outputs this yields a partition of the example equations whose inputs are subsumed by p. Since more than one position may be selected, different partitions leading to different sets of new initial rules may result.

For example, let

$$\begin{array}{lcl} reverse([]) & = & [] \\ reverse([X]) & = & [X] \\ reverse([X,Y]) & = & [Y,X] \end{array}$$

be some examples for the *reverse*-function. The pattern of the initial rule is simply a variable Q, since the example input terms have no common root symbol. Hence, the unique position at which the pattern contains a variable and the example inputs different constructors is the root position. The first example input consists of only the constant [] at the root position. All remaining example inputs have the list constructor *cons* as root. Put differently, two subsets are induced by the root position, one containing the first example, the other containing the two remaining examples. The least general generalisations of the example inputs of these two subsets are [] and $[Q|\,Qs]$ resp. which are the (more specific) patterns of the two successor rules.

Introducing (Recursive) Function Calls and Auxiliary Functions. In cases (ii) and (iii) help functions are invented. This includes the generation of I/O-examples from which they are induced. For case (ii) this is done as follows: Function calls are introduced by matching the currently considered outputs, i.e., those outputs whose inputs match the pattern of the currently considered rule, with the outputs of any defined function. If all current outputs match, then the rhs of the current unfinished rule can be set to a call of the matched defined function. The argument of the call must map the currently considered inputs to the inputs of the matched defined function. For case (iii), the example inputs of the new defined function also equal the currently considered inputs. The outputs are the corresponding subterms of the currently considered outputs.

For an example of case (iii) consider the last two *reverse* examples as they have been put into one subset in the previous section. The initial rule for these two examples is:

$$reverse([Q|Qs]) = [Q2|Qs2] \quad (1)$$

This rule is unfinished due two the two unbound variables in the rhs. Now the two unfinished subterms (consisting of exactly the two variables) are taken as new subproblems. This leads to two new examples sets for two new help functions $sub1$ and $sub2$:

$$\begin{array}{lclllcl} sub1([X]) & = & X & & sub2([X]) & = & [] \\ sub1([X,Y]) & = & Y & & sub2([X,Y]) & = & [X] \end{array}$$

The successor rule-set for the unfinished rule contains three rules determined as follows: The original unfinished rule (1) is replaced by the finished rule:

$$reverse([Q|Qs]) = [sub1([Q|Qs] \mid sub2[Q|Qs]]$$

And from both new example sets an initial rule is derived.

Finally, as an example for case (ii), consider the example equations for the help function $sub2$ and the generated unfinished initial rule:

$$sub2([Q|Qs] = Qs2 \quad (2)$$

The example outputs, $[], [X]$ of $sub2$ match the first two example outputs of the *reverse*-function. That is, the unfinished rhs $Qs2$ can be replaced by a (recursive) call to the *reverse*-function. The argument of the call must map the inputs $[X], [X,Y]$ of $sub2$ to the corresponding inputs $[], [X]$ of *reverse*, i.e., a new help function, $sub3$ is needed. This leads to the new example set:

$$\begin{array}{lcl} sub3([X]) & = & [] \\ sub3([X,Y] & = & [X] \end{array}$$

The successor rule-set for the unfinished rule contains two rules determined as follows: The original unfinished rule (2) is replaced by the finished rule:

$$sub2([Q|Qs] = reverse(sub3([Q|Qs]))$$

Additionally it contains the initial rule for $sub3$.

2.2 IGOR 2's use of MAUDE's Term Rewriting and Homoiconic Capabilities

In the functional subpart of MAUDE, a module essentially defines an order-sorted signature[1] Σ, a set of variables X, and a term rewriting system over Σ and X. Hence, IGOR 2's I/O examples, background knowledge, and induced programs are valid and evaluateable MAUDE modules. Since I/O examples, background knowledge, and induced CSs are input and output respectively, i.e., *data* for IGOR 2, we need some *homoiconic* capabilities: A MAUDE

[1] Order-sorted signatures are a non-trivial extension of many-sorted signatures. In an order-sorted signature, the sorts partially ordered into sub- and supersorts.

program (IGOR 2) needs to handle MAUDE programs as data. This is facilitated by MAUDE's meta-level. For all constructs of MAUDE modules—signatures, terms, equations, and complete modules—sorts and constructors to represent them are implemented in the META-LEVEL module and its submodules in MAUDE's standard library. Furthermore, functions to transform terms etc. to their meta-representation—`upTerm`, `upEqs`, and `upModule`—are predefined there. Meta-represented terms, equations, modules and so on are *terms* of types `Term`, `Equation`, `Module` etc. and may be rewritten by a MAUDE program like any other term.

Let us examine in some more detail, how terms and equations are meta-represented in MAUDE: Constants and variables are meta-represented by quoted identifiers containing name and type of the represented constant or variable. E.g., `upTerm(nil)` where `nil` is a constant of sort `List` yields the constant `'nil.List` of sort `Constant` which is a subsort of `Term` and `upTerm(X)` where `X` is a variable of sort `List` yields the constant `'X:List` of sort `Variable` which is also a subsort of `Term`. Other terms are represented by a quoted identifier as root and a list of meta-terms in brackets as arguments. E.g., `upTerm(Reverse(nil))` yields the term `'Reverse['nil.List]` of sort `Term`.

The constructor in mixfix notation for representing an equation is `eq_=_[_]` . where the first two _ may take a term in meta-representation each (the rhs and lhs of the equation) and the third _ an attribute set (belonging to an equation). The resulting term is of sort `Equation`.

Now consider a MAUDE module `M` containing the two equations

```
eq reverse(nil) = nil .
eq reverse(cons(X,nil)) = cons(X,nil) .
```

where X is a variable of sort `Item`. Applying `upEqs('M, false)` then yields:

```
eq 'Reverse['nil.List] = 'nil.List [none] .
eq 'Reverse['cons['X:Item,'nil.List]] =
       'cons['X:Item,'nil.List] [none] .
```

This is a term of the sort `EquationSet`.

Also concepts of rewriting, e.g., matching and substitutions, are implemented for the meta-level. For example,

```
metaMatch(upModule('M,false), 'X:List,
          'cons['Y:Item,'nil.List], nil, 0)
```

yields the term

```
'X:List <- 'cons['Y:Item,'nil.List]
```

of sort `Assignment` which is a subsort of `Substitution`.

3. IGOR 2 in HASKELL

As LISP, MAUDE is a dynamically typed, homoiconic language. This means that (i) the majority of its type checking is done at run-time so type information is available at this point, and, as seen in the previous section, (ii) it supports treating 'code as data' and vice versa 'data as code' very well. This is quite useful for program synthesis, because the data structure to represent hypotheses about possible programs can directly be treated as code and evaluated, and of course the other way round too. Any piece of code can be lifted into a data structure and be modified. Furthermore, names of functions or data type constructors can be reified, so the interpreter's symbol table is accessable at runtime. This makes it possible get the constructors of an arbitrary data type or the type of a function at run-time without much effort.

From the viewpoint of IP, HASKELL has on this matter its weak spot. As a typical statically typed language, types are only necessary until type checking is done. Once a piece of code has passed the type checker, type information can safely be dropped. Although this improves efficiency for compiled programs, when doing program synthesis, this information is necessary though. Lifting code to a meta-level and back, as done with MAUDE's `upXYZ` functions is only available quite restricted. Also reification cannot be done so easily since again, there is no access to the symbol table after type checking. There are various library extensions for HASKELL especially for GHC, to alleviate these problems, e.g. Template Haskell (TH) [Sheard and Jones 2002] for compile-time metaprogramming and `Data.Dynamic` and `Data.Typeable` to allow for dynamic typing. Why they are not useful for us though, we will explain soon.

Usually, in HASKELL expressions are represented as an algebraic data type:

```
data Exp
  = VarE Name
  | ConE Name
  | LitE Lit
  | AppE Exp Exp
```

Template Haskell's dual quasi-quoting (`[| |]`) and splicing (`$`) operators would provide us with the means to transform code into such an algebraic data type and these expressions back into code, similar to MAUDE's `upXYZ` functions. So `[|1|]` would be `LitE (IntegerL 1)` inside the TH's `Q` monad and `$(LitE (IntegerL 1))` would be replaced by the `Integer` value `1` by the compiler. However, this is only done at compile-time and without types of the quoted code itself. This simply comes from TH's use case to be able to write code-generating macros, so the purpose of quoting and splicing is really to coerce expressions into real code at compile-time and evaluate it at run-time instead of having an algebraic representation of that code at run-time.

Similarly, the dynamic typing library extension of HASKELL is not appropriate for us, too. Its main idea is by creating a type class `Typeable` to be able to compare the type of arbitrary and unknown values. For example the function `toDyn :: Typeable a => a -> Dynamic` from `Data.Dynamic`. Without knowing the type of an arbitrary value, but being a member of `Typeable`, a representation of its type can be created and e.g. compared. However, in our case we are not interested in a type representation of an expression, but of the type representation of an expression when interpreted as code.

In the rest of this section we will look at the HASKELL-specific details of the new IGOR 2 implementation.

3.1 Expressions, Types, and Terms

Finally, there is nothing else for us but to write our own expression type and tag it with an also algebraic representation of its underlying type.

```
type Name = String
data TExp
 = TVarE Name Type
 | TConE Name Type
 | TLitE Lit Type
 | TAppE TExp TExp Type
 | TWildE Type
data Lit
 = CharL Char
 | IntL Int
 | StringL String
```

So a typed expression is either a variable, a constant, a literal, or an application of them. For simplicity let a `Name` be just a `String`. Neglecting the types for the moment, the expression `(:) 1 ((:) 2 [])`[2] would be represented as follows:

```
TAppE (TAppE (TConE ":")
             (TLitE (IntL 1)))
      (TAppE (TAppE (TConE ":")
                    (TLitE (IntL 2)))
             (TConE "[]"))
```

The algebraic data type of a type looks similar, where a type is either a type variable, a type constant, an arrow, or an application of them.

```
type Cxt = [Type]
data Type
  = ForallT [Name] Cxt Type
-- variables in scope, class context, type
  | VarT Name
  | ConT Name
  | ArrowT
  | AppT Type Type
```

Additionally, there is a forall type, allowing us to restrict a type variable to a certain type class. As a short example, the type `(Show a)::` `a -> [Int]` is represented as the following algebraic expression:

```
ForallT ["a"] [AppT (ConT "Show")
                    (VarT "a")]
         (AppT (AppT ArrowT (VarT "a"))
               (AppT ListT (ConT "Int")))
```

[2] aka `1:2:[]` or `[1,2]`

For our convenience, we also create the `class Typed` to easily have access to a type of an expression or the like.

```
class Typed t where
  typeOf :: t -> Type
instance Typed TExp where
  -- omitted
```

For `TExp`, the function `typeOf` is just a projection on the last argument, i.e. the type of an expression constructor.

To work with `TExp` and `Type` in the sense of terms we make them all instances of a `class Term` which provides the basis for fundamental operations on terms. The function `sameSymAtRoot` compares two term only at their root symbol, `subterms` returns all immediate subterms of a term and `root` is the inverse of it such that `root t (subterms t)= t`. The functions `isVar`, `toVar`, and `fromVar` provide a type independent way to check for variables, access their name and create a variable from a name.

```
class (Eq t) => Term t where

  sameSymAtRoot :: t -> t -> Bool
  subterms      :: t -> [t]
  root          :: t -> ([t] -> t)
  isVar         :: t -> Bool
  toVar         :: t -> Name -> t
  fromVar       :: t -> Name

instance Term Type where
  -- omitted
instance Term TExp where
  -- omitted
```

Both, `Types` and `TExp` are instances of the `class Term`.

3.2 Specification Context

Up to now, we have seen how to represent expressions and types, but as mentioned earlier, this is not sufficient, since synthesis of a program takes place in a certain context. A small specification, which is itself a HASKELL module, could e.g. look like the following listing.

```
module FooMod where

data Peano = Z | S Peano
 deriving (Eq, Ord)

count :: [a] -> Peano
count []    = Z
count [a]   = S Z
count [a,b] = S S Z
```

Such a given specification is parsed and the IO examples for `count` are translated into `TExp`-expressions. Furthermore, all data type definition with their constructors and types have to be stored in a record modelling the context of this specification, i.e. all types and functions which are in scope. Since the standard Prelude is assumed to be allways in scope, their types and constructors are included statically. We use a named record for managing the context, where each field in this record is a Map from Data.Map storing the relevant key value pairs.

```
import qualified Data.Map as M

data SynCtx = SCtx
  { sctx_types    :: (M.Map Name Type)
  -- function name -> its type
  , sctx_ctors    :: (M.Map Name Type)
  -- constructor name -> its type
  , sctx_classes  :: (M.Map Name [Name])
  -- class name -> its superclasses
  , sctx_members  :: (M.Map Name [Name])
  -- class name -> member functions names
  , sctx_instnces :: (M.Map Type [Name])
  -- type -> classes
  , sctx_typesyns :: (M.Map Type Type)
}deriving(Show)
```

It is common practise to hide the relevant plumbing of stateful computation inside a state monad [Wadler 1992], and so do we. While we are at it we can start stacking monads with monad transformers [King and Wadler 1992] and add error handling. Later we will go on in piling monads, and because this is the bottom one it is self-evident to the add the error monad here. Our context monad now looks as follows with an accessor function lookIn for our convenience.

```
type C a = StateT SynCtx (ErrorT String a)

(lookIn) :: (Ord a) =>
        a -> (SynCtx -> M.Map a b) -> C b
(lookIn) n f = gets f >>= \m ->
    maybe (fail "Not in context!")
          return
          (M.lookup n m)
```

The function lookIn can now be used, preferably infix, wherever we need information about names or types. For example with **"Peano"** lookIn sctx_classes we get the names of the classes Peano is an instance of, here [**"Eq"**,**"Ord"**].

3.3 Using Terms

The cornerstones of our synthesis algorithm are unification and anti-unification. Due to our type-tagged expression, computing the *most general unifier* or the *least general generalisation* of two terms will become stateful, when considering polymorphic types with type classes. Not only the terms have to be unified or generalised, but with respect to their types. For this purpose we create the classes Unifiable and Antiunifiable and make both TExp and Type instances of them.

Substitutions which replace variables by terms are essential when unifying or antiunifying terms. Let a Substitution be a list of pairs, such that the variable with the name on the left side is replaced by the term on the right side of the pair. Then we define our unification monad U t again as a monad transformer as follows.

```
type Substitution t = [(Name,t)]
nullSubst = []

type U t = StateT (Substitution t) C ()
```

Note that the last argument of StateT is the unit type. Consequently, a computation inside U t has no result, or put differently, the result is the state itself, i.e. the substitution which is modified on the way. Therefore, when computing the *most general unifier* (mgu) or the substitution with which two terms match matchingS, unify and match respectively are executed in the U t monad with the empty substitution as initial state. As result the final state is returned.

```
class (Term t) => Unifiable t where

 unify     :: t -> t -> U t

 mgu       :: t -> t -> C (Substitution t)
 mgu x y = execStateT (unify x y) nullSubst

 match     :: t -> t -> U t

 matchingS :: t -> t -> C (Substitution t)
 matchingS x y =
        execStateT (match x y) nullSubst

 equal     :: (Unifiable t) =>
                 t -> t -> C Bool
 equal y x = matchingS x y >> return . null
        `catchError` \_ -> return False
```

Remember that we stacked the U t monad on top of our context monad C which supports error handling. So if two terms do not unify or match respectively, then fail is invoked inside C, otherwise a potentially empty substitution is returned inside C. The function matchingS returns the substitution that matches the first term on the second term and equal returns True if the computation inside U t succeeds with an empty substitution, False otherwise.

The class Antiunifier looks similar, but instead of a Substitution it uses the data type VarImg as state. VarImg stores a list of terms, i.e. the so called image, together with the variable subsuming these terms.

```
type VarImg t = [([t],Name)]
nullImg = []

type AU t = StateT (VarImg t) C t
```

However, unlike in the U t monad, there is a result of a computation in the AU t monad: The *least general generalisation* of the given terms. With the function antiunify we throw the state away and return the result of the monadic computation.

```
class (Term t) => Antiunifiable t where

    aunify           :: [t] -> AU t

    antiunify        :: [t] -> C t
    antiunify t      =
      runStateT (aunify t) nullImg
```

The types TExp and Type are now added as instances to these type classes. We omit the concrete implementations, since they are straight forward following the structure of the algebraic data types. All that is left to say that two TExps only unify/match/antiunify if and only if their types unify/match/antiunify.

3.4 Rules, Hypotheses, and other Data Types

Now let us introduce the basic data types for the synthesis.

First of all we have a Rule, with a list of TExps on the left-hand side (lhs) and one TExp on the right-hand side (rhs).

```
data Rule = R { lhs :: [TExp]
              , rhs :: TExp }
```

Usually we are talking about a certain rule, a rule covering some I/O examples of a specific function. Therefore we need to store information about this specific function and the covered I/O examples together with the Rule in a covering rule CovrRule.

```
data CovrRule = CR
    { name  :: Name
    , rule  :: Rule
    , covr  :: [Int] }
```

The accessor functions name, rule, and covr return the name of the function, the rule itself, and the indices of the covered I/O examples. A CovrRule makes therefore only sense, when there is something the indices refer to. The data structure IOData answers this purpose. It is more or less a map, relating function names to list of rules, i.e. the I/O examples. Let for simplicity be IOData just a synonym.

```
type IOData = M.Map Name [Rule]
```

The indices in a CovrRule are just the position of rules in the list stored under a name. The indices should not be visible outside IOData. For this purpose there are a couple of functions to create and modify CovrRule referring to a certain IOData. We refrain from the concrete implementations here.

```
getAll :: Name -> IOData
       -> Maybe [CovrRule]
getNth :: Name -> IOData
       -> Int -> Maybe CovrRule
```

As the names suggest, getAll is simply a lookup and returns just a list of covering rules, each covering one I/O pair, and getNth just picks the n^{th} of all. The following functions are used to breakup and fuse covering rules. So breakup returns a list of covering rule, each covering one I/O pair of those covered by the original one, and fuse is the inverse of it, fusing many covering rules into one which covers all their I/O pairs.

```
breakup :: CovrRule -> IOData -> [CovrRule]
fuse    :: [CovrRule] -> C CovrRule
```

We have to be inside the C monad for fusing, because we need to antiunify the rules to be covered.

Hypotheses are the most fundamental data record storing a list of open covering rules, the closed rules as a list of declarations Decl, for each function one, and all calling dependencies between all functions to prevent the system to generate non-terminating programs.

```
type Decl = (Name,[Rule])
data Hypo  = HH { open      :: [CovrRule]
                , clsd      :: [Decl]
                , callings :: CallDep }
```

The basic idea behind calling dependencies is that if function f calls function g, then f depends on g ($f \to g$). The argument(s) of a call could either increase, decrease or remain in size, thus the dependency could be of either type LT, EQ, or GT ($\xrightarrow{<}$, $\xrightarrow{=}$, $\xrightarrow{>}$).

Calling dependencies are transitive, so if $f \to g$ and $g \to h$ then also $f \to h$. The kind of the transitive dependency has the maximal type of all compound dependencies with the obvious ordering LT $<$ EQ $<$ GT.

If already a calling dependency $f \to g$ exists, the following possibilities for g calling f are allowed:

$$
\begin{aligned}
f \xrightarrow{>} g &\Rightarrow g \text{ is not allowed to call } f \\
f \xrightarrow{=} g &\Rightarrow g \xrightarrow{<} f \\
f \xrightarrow{<} g &\Rightarrow g \xrightarrow{<} f \text{ or } g \xrightarrow{=} f \\
f = g &\Rightarrow f \xrightarrow{<} f
\end{aligned}
$$

If there is no such calling dependency, all possibilities are allowed. To check, whether a call is admissible and to get all allowed possible calls two functions exist.

```
admissible :: (Name,Ordering,Name)
           -> CallDep
           -> Bool
allowedCalls :: Name
             -> CallDep
             -> M.Map Name [Ordering]
```

The first one checks if the given (new) calling dependency is admissible, and the second returns for each function in a CallDep which additional calls to it are allowed. If a function is not mentioned in the Map returned by allowedCalls, anything goes.

3.5 Comparing Rules and Hypotheses

To compare rules and hypotheses to decide which to process we establish the class Rateable with the member function `rate` which returns for each member an `Int` value inside `C`.

```
class Rateable r where
   rate :: r -> C Int
```

Hypotheses should be rated with regard to their number of different partition, i.e. patterns on the left-hand side of all their rules that do not match any other pattern. This is motivated by some kind of Occam's razor, preferring programs with few rules.

```
instance Rateable Hypo where
   rate h       =  numberOfPartitions h

numberOfPartitions :: Hypo -> RatingData
numberOfPartitions h =  liftM length $
        foldM leastPatterns $ allRules h
  where
  leastPatterns [] p       = return [p]
  leastPatterns (p1:ps) p2 = do
  p1gtp2 <- match `on` lhs p1 p2
  p2gtp1 <- match `on` lhs p2 p1
  if p1gtp2 then return (p2:ps)
     else if p2gtp1 then return (p1:ps)
            else liftM (p1:)
                       (leastPatterns ps p2)
```

Covering rules are rated with regard to the longest chain of function calls they are in, so preferring rules causing less nested function calls. To compute the length of this longest path in the call dependencies, always a `CallDep` is required.

```
instance Rateable (CallDep,CovrRule) where
   rate (cd,cr)  =  return.length (
       longestPath (name cr) cd)
```

3.6 The Synthesis Monad

For searching a space of hypotheses we need to maintain a data structure representing this search space. In each step, the best hypothesis w.r.t to a certain heuristic is selected and from it an appropriate rule, again w.r.t an *a priori* defined heuristic is chosen. Refining one rule results in multiple sets of rules, because multiple refinement operators are used and each operator may result itself in multiple rules.

So let r be a rule and $\rho_1 \ldots \rho_n$ are refinement operators, then are $\rho_i(r)$ the rules resulting in applying ρ_i to r. If R is the set of all rules occurring in any hypothesis h, then is H the set of all hypotheses, with H included in the powerset of R, where each h is treated as a set of rules. Applying the refinement operators to a rule r in R results in $R' = R \setminus \{r\} \cup \{\rho_1(r), \ldots, \rho_n(r)\}$, thus changing H to $H' = H \setminus \{h | r \in h\} \cup \{h_i | h_i = h \setminus \{r\} \cup \rho_i(r)\}$ for $i = 1 \ldots n$.

This makes the implementation of our search approach lack elegance when compared to breadth-first search combinators proposed by Spivey [Spivey 2000, Spivey and Seres 2000], where the space for breadth-first search can be defined as an infinite list. Katayama for example efficiently uses this approach [Katayama 2008, 2007], because he is able to define his search space intensionally *a priori*.

Following the current implementation, this is not applicable for us. Hypotheses represent partial or unfinished programs, so our search space changes over time, because refinement operators may but need not finish a hypothese but refine it to multiple, also unfinished, successor hypotheses. Thus, refining one rule may affect multiple hypotheses and change the ordering in the search space after each step.

Therefore we need to pull the whole search space explicitly through all our computations. Again, a stateful monad transformer on top of our `C` monad does the trick.

```
data Igor =
  Igor { iodata :: IOData
       , searchSpace :: HSpace}

type I a = StateT Igor C a

modifyHS :: (HSpace -> HSpace) -> IM()
modifyHS f =
  modify (\igor@(Igor _ sp _) ->
    igor{searchSpace = f sp})

modifyIO ::(IOData -> IOData) -> IM()
modifyIO f =
  modify (\igor@(Igor io _ _) ->
    igor{iodata = f io})
```

The data structure `Igor` bundles the data structures `IOData`, known from section 3.4 to manage the various IO examples and `HSpace`, a priority queue on hypotheses w.r.t. to their heuristical rating. `HSpace` also supports efficient access to hypotheses by their rules to facilitate updating hypotheses after one refinement step. `Igor` serves as state for the monad `I`. The functions `modifyHS` and `modifyIO` allow us to modify `HSpace` and `IOData` inside `I`.

The main loop returns a list of equivalent programs inside `I`, w.r.t. the given heuristic, explaining the IO examples of the target function. Each program consists of a list of declarations `Decl` where each `Decl` defines one function by at least one `Rule`. First it fetches the currently best hypotheses, extracts the call dependencies and the unfinished rules from this hypothesis. If there are no open rules in all candidate hypotheses, the loop is exited and the candidate hypotheses are returned as result. Otherwise one rule is chosen for refinement, refined using the call dependencies and thus modifying the search space. After all, the loop is entered again.

```
type Prog = [Decl]

enterLoop :: I [Prog]
enterLoop =  do
  chs        <- currentBestHypos
  (deps,crs) <- chooseOneHypo chs
  if (null crs)
```

```
        then stopWith chs
        else chooseOneRule crs >>=
             refine deps >>
             enterLoop
```

Finally, `refine` computes all refinements, introduced in Section 2, of the given unfinished rule with `refineRule` and propagates the result, a set of all possible refinements, to the whole search space and updates all affected hypotheses with `propagate`.

```
refine :: CallDep
       -> CovrRule
       -> I ()
refine cd cr =
    refineRule cd cr >>=
    (modifyHS .) . propagate $ cr

refineRule :: CallDep -> CovrRule
           -> IM [(CovrRules,[Call])]
refineRule cd cr = do
    parts <- partition cr
    cllfs <- callFunction cd cr
    subfs <- inventSubfunction cr
    return $ parts ++ subfs ++ cllfs
```

4. Empirical Results

To test our new implementation (in the following named as IGOR 2_H) against the old we have chosen some usual example problems on lists. As usually, they incorporate different recursions patterns, simple linear as in *last* or mutual recursive as in *odd/even*. Most of the problems suggest for inventing auxiliary function as e.g. *lasts*, *repeatlst*, *sort*, *reverse*, *oddpos* but only *reverse* is explicitly only solvable with.

Most of the problems have the usual semantics on lists and can be found in a standard library of a functional Language. Table 1 shows a short explanation of each of them nevertheless.

The tests were run on a laptop with a 1.6Ghz Intel Pentium processor with 2GB RAM using Ubuntu 8.10. IGOR2.2 with MAUDE 2.4 and version 0.5.9.4 of the HASKELL implementation have been used. All programs as well as the used specification and a batch file for the HASKELL implementation can be downloaded from our webpage[3].

Keeping in mind that MAUDE is an interpreted language and IGOR 2_H is compiled, it is not surprising that the new implementation is faster. A cutback to a 1/10 or more in most of the cases is more than expected, though. Table 2 shows all runtimes and the approximte ratio of old to new.

[3] http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html

Table 1. Problem descriptions

add	is addition on Peano integers,
append	appends two lists,
drop	drops the first n elements of a list,
evenpos	are all elements in a list which index is even,
init	are all elements but the last of a list,
last	is the last element in a list,
last	maps *last* over a list of lists,
length	is the length of a list as Peano integer,
odd/even	defines *odd* and *even* mutually recursive on Peano integers,
oddpos	are all elements in a list which index is odd,
repeatfst	overwrites all elements in a list with the first,
repeatlst	overwrites all elements in a list with the last,
reverse	reverses a list,
shiftl	shifts all elements in a list one position to the left,
shiftr	shifts all elements in a list one position to the right,
sort	sorts a list of Peano integers using *ins* as background knowledge which inserts into a sorted list,
swap	changes the position of two consecutive elements in a list element in a list,
switch	changes the position of the first and the last element,
take	takes the first n elements from a list, and
weave	merges two lists into one by alternating their elements.

5. Conclusion

We introduced the new program design of our system IGOR 2, which has been ported from MAUDE to HASKELL. We described how, for our purpose necessary, features of the homoiconic language MAUDE can be simulated in HASKELL using a stateful monad transformer. Although we can not model MAUDE's full reflexive capabilities, we can simulate all functionality necessary in our use case. With our new implementation we paved the way to use higher-order context during our synthesis and extract information from types and their classes useable as background knowledge. Keeping our new implementation as close as possible to our old, it was possible to keep all features of our system as e.g. termination by construction of both synthesised programs and IGOR 2-algorithm, minimality of generalisation, using arbitrary user-defined data types and background knowledge, and others.

For the future we plan to utilise universal properties of higher-order functions such as *fold, map* and *filter* to introduce certain recursion schemes as programming patterns

Table 2. Runtimes on different problems in seconds

	IGOR 2	IGOR 2_H	$\frac{\text{IGOR } 2_H}{\text{IGOR } 2}$ *
add	0.236	0.076	1/3
append	46.338	0.080	1/579
drop	0.084	0.004	1/21
evenpos	0.056	0.004	1/14
init	0.024	0.004	1/6
last	0.024	0.001	1/24
lasts	6.744	0.020	1/337
length	0.028	0.001	1/28
odd/even	0.080	0.004	1/20
oddpos	18.617	0.048	1/388
repeatfst	0.052	0.004	1/13
repeatlst	0.100	0.004	1/25
reverse	0.617	0.032	1/19
shiftl	0.084	0.008	1/11
shiftr	0.308	0.020	1/15
sort	0.148	0.012	1/12
swap	0.108	0.008	1/14
switch	2.536	0.036	1/70
take	1.380	0.012	1/115
weave	0.348	0.036	1/13

* rounded to nearest proper fraction

when applicable. In this context we will make use of type information which is now accessable. Furthermore, it should be promising to reconsider the current algorithm to make use of lazy data structures to better take advantage of the benefits of lazy evaluation. Memoization could also be helpful to avoid propagating the change of a rule over the whole search space.

Acknowledgments

Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.

References

Alan W. Biermann, Yves Kodratoff, and Gerard Guiho. *Automatic Program Construction Techniques*. The Free Press, NY, USA, 1984. ISBN 0029490707.

Pierre Flener. Inductive logic program synthesis with Dialogs. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 28–51. Stockholm University, Royal Institute of Technology, 1996.

José Hernández-Orallo and M. José Ramírez-Quintana. Inverse narrowing for the induction of functional logic programs. In José Luis Freire-Nistal, Moreno Falaschi, and Manuel Vilares Ferro, editors, *Joint Conference on Declarative Programming*, pages 379–392, 1998.

Martin Hofmann. *Automatic Construction of XSL Templates – An Inductive Programming Approach*. ISBN: 978-3-639-00194-5. VDM Verlag, Saarbrücken, 2007. doi: ISBN978-3-639-00194-5.

Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007.

Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI*, pages 199–210, 2008.

David King and Philip Wadler. Combining monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

Emanuel Kitzelmann. Data-driven induction of recursive functions from I/O-examples. In Emanuel Kitzelmann and Ute Schmid, editors, *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

Emanuel Kitzelmann. Analytical inductive functional programming. In m. Hanus, editor, *Proceedings of th 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008, Valencia, Spain)*, volume 5438 of *LNCS*, pages 87–102. Springer, 2008.

Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006. ISSN 1533-7928.

Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997. ISBN 0070428077.

Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

Roland J. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1): 55–83, 1995.

J. Ross Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

J. Michael Spivey. Combinators for breadth-first search. *J. Funct. Program.*, 10(4):397–408, 2000.

Mike Spivey and Silvija Seres. The algebra of searching. In *Proceedings of a symposium in celebration of the work of*. MacMillan, 2000.

Phillip D. Summers. A methodology for LISP program construction from examples. *Journal ACM*, 24:162–175, 1977.

Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

Automated Method Induction

Functional Goes Object Oriented

Thomas Hieber, Martin Hofmann
University of Bamberg - Cognitive Systems Group
thomas-wolfgang.hieber@stud.uni-bamberg.de, martin.hofmann@uni-bamberg.de

Abstract

The development of software engineering has had a great deal of benefits for the development of software. Along with it came a whole new paradigm of the way software is designed and implemented - object orientation. Today it is a standard to have UML diagrams automatically translated into program code wherever possible. However, as few tools really go beyond this we demonstrate a simple functional representation for objects, methods and variables. In addition we show how our inductive programming system *Igor* can not only understand those basic notions like referencing methods within objects or using a simple protocol like something we called *message-passing*, but how it can even learn them by a given specification - which is the major feature of this paper.

Keywords Inductive Programming, Object Oriented Programming, Igor, Maude, Java, Recursion

1. Introduction

Since mainstream software for business use is commonly not created with functional programming languages it is about time to raise the question whether it is possible to adapt object oriented language features to a functional setting. *Igor* is a system for synthesizing recursive functional programs, which learns recursive functions solely from input/output (I/O) examples, and will henceforth be our system of choice concerning program induction. Since *Igor* is naturally based on functional programming, the main focus of this paper lies on finding a way to use *Igor* for program inference in an object oriented background. This requires us to express the behavior of objects and method calls by I/O examples. In order to do so, it is necessary to find a way to express object oriented programs in a functional way. The main part of this paper will be concerned with this task on a very general scale. It is not meant to be a complete approach but an analysis of what is possible and what is not.

At the same time, it is necessary to enable an object oriented programmer to provide input to the synthesis system as unobtrusive as possible. For this purpose, we have devised an interface for Eclipse which will allow a programmer to use annotations in order to provide input for our induction process, thus seamlessly integrating with software engineering tools like *Rational Software Architect (RSA)*. Since it is not the focus of this paper to explain the functionality of this prototype we only mention it for the sake of completeness - more on this subject is to be found in [Hieber 2008].

In section 2 we start off with a short survey of current and past research in the field of functional programming and object orientation. Section 3 is a short roundup about *Inductive Programming (IP)* and *Igor* while section 4 focuses on how we represent object orientation to *Igor* and how it can even learn all of the concepts created. The prototype plug in for *Eclipse*, which is a further result of our research, will then be described very shortly in section 5 before we finally conclude in section 6.

2. The Status Quo

In the past 30 years, many different inductive programming (IP) systems have been developed, many of them sharing a functional approach. The extraction of programs from input/output examples started in the the seventies and has been greatly influenced by Summers' [Summers 1977] paper on the induction of *LISP* programs. After the great success of *Inductive Logical Programming (ILP)* on classification learning in the nineties, the research focus shifted more to this area. Prominent ILP systems for IP are for example *FOIL* [Quinlan and Cameron-Jones 1993], *GOLEM* [Muggleton and Feng 1990] or *PROGOL* [Muggleton 1995] - systems which make use of *Prolog* and predicate logic.

Later, the functional approach was taken up again by the analytical approaches *Igor1* [Kitzelmann and Schmid 2006], and *Igor2*[Kitzelmann 2007] and by the evolutionary/generate-and-test based approaches *Adate*[Olsson 1995] and *MagicHaskeller*[Katayama 2007].

All in all you can subsume the concern of *Inductive Programming* as the search for algorithms which use as little additional information as possible to generate correct computer programs from a given minimal specification consisting of input/output examples.

At the same time functional languages have had to face the development in programming paradigms which led to many approaches to support object orientation. Established functional languages have their own object oriented extensions like OCaml [Remy and Vouillon 1998] or OOHaskell ([Kiselyov and Laemmel 2005]). Additionally there are various approaches to include an object system into a functional language without changing the type system or the compiler (see e.g. [Kiselyov and Laemmel 2005] for a Haskell related overview).

For our purpose we do not need such sophisticated techniques (yet), therefore we content ourselves with taking on a quite naive and very simplified perspective, though sufficient for our case, and treat objects merely as tuples.

On the other hand there are some very powerful tools for object oriented programmers which support automated code-generation to a certain extent and the community for *Automated Software Engineering* is very productive to take this even further. In this context it is inevitable to have a look at program synthesis since we ideally do not want to stop at automatically generating class files from UML diagrams like IBM's *RSA*, or generate a GUI by 'WYSIWYG' editors such as *NetBeans* or *Visual Studio*.

3. Inductive Programming & IGOR

Summers' theories have been taken up again in [Schmid 2003], where the *Igor* system was put into existence. The basic idea is to generate a set of (recursive) equations from a specification consisting of input/output examples. Its first system was written in *LISP* and it was closely connected to Summers' suggestions. A few years later a newer version of *Igor* was created and it extended the prior version by a number of improvements. In [Kitzelmann 2008] you find a more detailed description of *Igor2* as a system which now employed mechanics such as anti-unification of the initial input/output examples and a *best-first* search over succeeding sets of equations which are to be formed by *term-rewriting*. Since this shift in the way programs were now processed did not play to the strengths of *LISP* like the former version, *Igor2* was written in the reflective term-rewriting language *Maude*.[1]

In order to understand how the system works before we go ahead and use it, let us consider the following example of the list-operator *length*.

Listing 1: Input/Output Examples for Igor

```
length([]) = 0
length([y]) = succ(0)
length([y,x]) = succ(succ(0))
length([y,x,z]) = succ(succ(succ(0)))
```

Given those examples, *Igor* correctly identifies the following recursive program:

Listing 2: Recursive Program for length

```
sub1(cons(x0,x1)) = length(sub2(cons(x0,x1)))
sub2(cons(x0,x1)) = x1
length[] = 0
length(cons(x0,x1)) = succ(sub1(cons(x0,x1)))
```

[1] see `http://maude.cs.uiuc.edu/`

This is what *Igor* produces from our specification (we have only adjusted the syntax to increase readability), especially the two functions **sub1** and **sub2** have been automatically introduced by the system. The *succ* operator is the well known *successor* in order to define the natural numbers as *peano numbers*. Evidently this system's output is purely functional so we are going to be concerned with finding out how to bring this closer to an object oriented context.

4. Igor and Object Orientation

As already mentioned, *Igor* is firmly based within the functional paradigm along with all its strengths and weaknesses. Nevertheless it is going to be subject of our concern in which way it could be possible to represent an object-oriented specification with *Maude* and feed it to *Igor*. For this we are going to put together some example specifications, have them synthesized and evaluate the output. In order to do so it is important to understand how we could possibly map the way object-oriented programs are presented to a functional notation. We are going to deal with this problem's theory first and then try and find out how *Igor* will react to our input.

When we are dealing with *Maude* specifications in the following chapters, let the following notation be established:

$$[object].[datatype]$$

This is the way data types are represented by *Maude* and we will stick to it for the sake of transparency. For the *Maude* results we will also establish a notation since the code generated is not quite readable. So the way results will be displayed like this:

$$ResultRule/Pattern(EquationLHS) = (EquationRHS)$$

4.1 Representing the Object

In this attempt we will try and keep it simple, as we are only exploring so the motto is to start small. When thinking about objects we can agree that they basically consist of an **identifier**, a set of **(member) variables** [2] and a set of **methods**. So it seems quite advisable to represent any object like this:

$$identifier.String \times properties.List \times methods.List \rightarrow Object$$

Let us for now just take properties and methods as black boxes, we will deal with them after this. Apart from the elaboration on those, there are only two things left in order to get a basic quasi object-oriented system: **calls** and **exceptions**. The former is the basic notion of a messages sent between objects in our system. The latter are a vital part of any high level programming language and, more importantly, we are going to need them in order to correctly specify some of our components. Since error handling is not the major part of our

[2] a.k.a. properties - see section 4.4

concern, let it just be introduced as black box - we are not going to analyse it any further.

Messages shall be defined like this:

$ParamList \times Object \rightarrow Message$

So a message consists of a number of arguments (*ParamList*) and an object which in case of a function call can carry the return value back to the sender, which leaves us with the following definition of the object in the *Maude* specification:

Listing 3: Object Constructor

```
op ___ : Identifier PropList MethodList -> Object [ctor]
    .
```

Note the ___ as the constructor's name - it is *Maude* syntax for n-ary operators with blanks as constructors (three underscores → three parameters). The constructor uses an *identifier*, a set of *properties* and a list of *methods* in order to create a new object. Now that we have an idea of how to represent an object, let us try and find out how we can do the same thing on methods.

4.2 Representing the Method

Before going on we have to bear in mind that - for now - we are dealing with methods only on a syntactic level. We only want to find out how to represent them in the context of an object. We are not concerned with the procedures within the method's body nor with how they are used. All we need to know for now is what information we need about a method on the object level in order to keep it as abstract as possible. Remember that we want to have this representation to be kept within the *MethodList* in our newly defined object. Right now we can say that a representation of a method must contain the following information:

- Method Name
- Return Value
- Argument Specification

When we formally put this together it ends up looking like this:

$identifier.String \times return\_value.DType \times arguments.List \rightarrow Method$

The *DType* is again to be taken as black box here since we are not interested in type inference or casting, so to understand that it is necessary information for any object calling the method is enough in this context.

This leaves us with the definition in *Maude* as follows:

Listing 4: Method Constructor

```
op met : Identifier DType ParamList -> Method [ctor].
```

As we have seen before, this is basic Maude notation for defining an operator called *met*. It takes three parameters (Identifier, DType and ParamList) and produces a *Method*. Since this is a classical constructor the *[ctor]* command is used at the end of the definition.

4.3 Representing the Method Call

Before we can actually call a method we have to resolve the identifier within the object which supposedly encapsulates it. In order not to become too confusing we are going to step away from objects for one moment and just focus on the way you might find a method within an object. For this let us assume that there exists a method list as depicted in 4.1 and an object trying to call a method by an identifier. The idea is to get a matching process like:

$Identifier \times MethodList \rightarrow Method$

By now we have introduced a few basic notations in object orientation. They all share the tuple-structure which is important in order to build the bridge to functional programming. As a consequence, these concepts can now be modelled in *Maude* (see section A.1) making it easy for us to construct simple examples in order to demonstrate how *Igor* responds to them. As a start we have picked the 'identifier-match' which is the mapping procedure we have just introduced. The full *Maude* specification is to be found in listing16 in this section we will only display short snippets in order to illustrate.

Listing 5: Identifier Match

```
sorts List Method Identifier DType ParamList NPException
    .
  subsort Method < NPException .
```

In the first part there are some *sort* definitions which are quite obvious and should be familiar by now. The only slightly strange thing is the second line. Here we basically bring in the exception since we want a *NPException (= Null Pointer Exception)* to be thrown in case an identifier is not found within the method list. The exception is here derived from *Method* which is obviously not very elegant or - strictly spoken - even wrong. But since we have not yet constructed a well defined object framework we can forsake the strict rules which would come along with it and just have the exception be the subclass of *Method*. This gives us the chance to explain another concept in Maude - sorts and subsorts. You can see that there is a number of sorts defined, was well as a relation between *Method* and *NPException*. The operator used here is < which can be seen like an arrow pointing from the specific to the more general sort.

The next part of the specification (listing 6) gives us some constructors and properties before we can go ahead and define our input examples.

Listing 6: Identifier Match - Constructors and Properties

```
op [] : -> List [ctor] .
op cons : Method List -> List [ctor] .
op mm : Identifier DType ParamList -> Method [ctor] .
ops id1 id2 id3 : -> Identifier .
op parlist : -> ParamList [ctor] .
op exc : -> NPException .
op dt : -> DType .
op match : Identifier List -> Method [metadata "induce"]
  .

vars ml m2 m3 : Method .
```

Here we find a basic procedure of constructing a list (ll 1,2), a method (*mm* operator), some random identifiers, arguments, an exception as well as a datatype. Note that identifiers, arguments, exception and datatype are just instantiated without any concrete data attached but for the current level of abstraction it is not necessary to do so. The operator *match* now is the method to be induced by *Igor* and after declaring a few properties as methods all there is left to do is to assert our input/output examples.

Listing 7: Identifier Match - Input/Output Examples

```
eq match(id1 , [] ) = exc .

eq match(id2 , [] ) = exc .

eq match(id1 , cons(mm(id1 , dt , parlist) ,[]) ) =
 mm(id1 , dt , parlist) .

eq match(id1 , cons(mm(id2 , dt , parlist), []) ) = exc .

eq match(id2 , cons(mm(id1 , dt , parlist) ,[]) ) = exc .

eq match(id2 , cons(mm(id2 , dt , parlist), []) ) =
 mm(id2 , dt , parlist) .

[...]
```

The equations in listing 7 are used to give *Igor* some basic examples in the problem domain. Here we bring together what we have defined earlier (Listings 5 and 6). The first two are quite obvious and finally explain why we insisted on exceptions earlier. Of course there could just be an empty method as a return value, but since we are trying to conquer the object oriented world with *Igor*, it feels more natural to express it this way. All the other examples (see complete listing 16) are summarised quite quickly - every time the method called is contained in the method list it is returned.

If this is now fed to *Igor*, one of the resulting hypotheses (translated into a little more readable syntax) returned is a set of equations. $X1$ and $X2$ are identifiers, $X3$ is a list, dt a datatype and $parlist$ a list of parameters:

1. $match(X1, []) = exc$
2. $Sub1(X1, cons(mm(X2, dt, parlist), X3)) =$
   $case\,(X1 == X2)\,of\,False\ \rightarrow X1$
3. $Sub2(X1, cons(mm(X2, dt, parlist), X3)) =$
   $case\,(X1 == X2)\,of\,True\ \rightarrow X3$
4. $match(X1, cons(mm(X2, dt, parlist), X3)) =$
   $case\,(X1 == X2)\,of\,False\ \rightarrow$
   $match(Sub1(X1, cons(mm(X2, dt, parlist),$
   $case\,(X1 == X2)\,of\,True\ \rightarrow$
   $mm(X1, dt, parlist)$

From this simple example we can already see how *Igor* tackles this problem. The *base case* is the first equation. Equations 2 to 4 ensure that the number of methods in the list is gradually decreased every time the first method in the list does not correspond to the one called. So at the end the list of methods becomes either void (→ equation 1) or the method is found at the head of the current method list (→ equation 5).

Already we can observe how *Igor* tries to find a recursive solution to this problem, which may seem a little complicated for this purpose, but it is exactly what we wanted to achieve and so we can go on at this point knowing that *Maude* and *Igor* can handle what we outlined earlier.

4.4 Concerning Properties

For our purpose, properties are very similar to methods. They just happen to be much more simple since there is no need for a list of arguments to be carried around. This comes all down to this simple line in our object specification in *Maude*:

Listing 8: Property Constructor

```
op prop : Identifier DType -> Property< [ctor].
```

The way a property is referenced is exactly the same as we have just done it with methods just that our *MethodList* would now be a *PropertyList* - so there is no point in repeating the procedure all over.

4.5 Messages

As already mentioned, we are going to relate every action within our system to messages. In 4.1 we have defined the specification of them and this is how they look in *Maude*:

Listing 9: Message Constructor

```
op msg : ParamList Object -> Message [ctor] .
```

We have seen how the matching of identifiers works, so let us now find out about messages sent between two imaginary objects. Since we are now only concerned with the way data is wrapped within them we drop overhead like identifiers and the like for now and focus on the core procedure which takes a message and its arguments and returns an object as result value.

We are going to test this with an example problem - the *even* operation which determines if a number is even or not. As before, we first have to define a couple of sorts.

Listing 10: Identifier Match Sorts

```
sorts InVec Object .
sorts Message ParamList .
sorts Nat Bool Param .
 subsorts Param < Nat Bool .
 subsorts Object < Nat Bool .
```

As we want to compute some real data this time, we have to refer *Param* and *Object* to real values as we do here.

Listing 11: Identifier Match Constructors

```
op <> : -> ParamList [ctor] .
op msg : ParamList Object -> Message [ctor] .
op null : -> Object [ctor] .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op t : -> Bool [ctor] .
op f : -> Bool [ctor] .
op cpar : Param ParamList -> ParamList [ctor] .

op method : Message -> Message [metadata "induce"] .
```

Next to the already known definitions of *message* and the usual list operations there are some more definitions. In addition to the *successor* operator (we call it just *s* this time) we need the boolean values *true (t)* and *false (f)*. What we want for *Igor* to do now is to unwrap a message, take the argument list as input and put the result back into a message. Formulated with input/output examples this is what we get:

Listing 12: Identifier Match Input/Output Examples

```
eq method( msg(cpar( 0,  <>), null) ) =
 msg(<> ,t ) .

eq method( msg(cpar( s(0),  <>), null) ) =
 msg( <> , f ) .

eq method( msg(cpar( s(s(0)), <>), null) ) =
  msg( <> , t ) .

eq method( msg(cpar( s(s(s(0))), <>), null) ) =
 msg( <> , f ) .

eq method( msg(cpar( s(s(s(s(0)))), <>), null) ) =
 msg( <> , t ) .
```

So we assume that *Igor* simulates an object getting a message with a natural number as parameter, returning a message containing a boolean. Now we will once again run this through the system and get the following set of equations (with $X1$ being a natural number):

1. $Sub19(msg(cpar(s(s(X1)), <>), null)) = msg(cpar(X1, <>), null)$
2. $method(msg(cpar(0, <>), null)) = msg(<>, t)$
3. $method(msg(cpar(s(0), <>), null)) = msg(<>, f)$
4. $method(msg(cpar(s(s(X1)), <>), null)) = method(Sub19(msg(cpar(s(s(X1)), <>), null)))$

On the level of semantics this looks just like what we wanted. On every left hand side there is a message with arguments and the right hand side contains messages with return value. So *Igor* has learnt the concept of message-passing, but since we provided a real problem specification encapsulated within the message this time, we will have to evaluate the resulting program for functional validity also. For this it seems appropriate to take off the wrapping from the synthesised equations and just show the important bits.

1. $Sub19(s(s(X1))) = X1$
2. $method(0) = t$
3. $method(s(0)) = f$
4. $method(s(s(X1))) = method(Sub19(s(s(X1))))$

Now this looks just like what we intended. Equations 2 and 3 are the *base cases*, 4 and 1 make sure that any number bigger than 1 will gradually be reduced by two until one of the *base-cases* is reached. Then the result value is ultimately returned.

4.6 Back Into Perspective

Before we try and draw a conclusion out of the results learned let us quickly summarize what we have gained so far. We have modelled a simple object-oriented protocol consisting of **Objects**, **Properties**, **Methods** and **Message-Calls**. Modelling those concepts in a functional way has brought our program synthesis system - *Igor* - to understand and even ‘learn‘ simple routines like ‘identifier-matching‘ and ‘message-passing‘. Now for a final test let us have a look how it can handle some of the syntactic sugar which is widespread in object oriented programming languages - a simple iteration over a collection. The task is to take a set of abstract objects and apply a method to every object within the set (which is actually a list). In listing 13 (proper Maude example in listing 20) we take one collection of objects and as we iterate over them we apply a method to them and put the results into a new collection. The results are represented in listing 21.

Listing 13: Iterate Collection Input/Output Examples

```
eq iterate([]) = {} .
eq iterate( put(Y,[]) ) = put2( met(Y), {}) .
eq iterate( put(X,put(Y,[])) ) = put2( met(X), put2( met(
    Y) ,{})) .
[...]
```

As you can see from the equations in listing 13 we employ two different collections and along with it two different constructors *put* and *put2*, which are like a *cons* operator. This is not necessary but in order to illustrate that we are actually removing the objects from one to another collection it seems to be more appropriate.

In our second example in listing 22 we go the same way we already did with methods and objects. We expand our simple iteration example by enhancing the method call itself. Another layer of abstraction is added or, if you will, some more object oriented ‘overhead‘ by adding more detail into the method call like identifier and the parent object the method is to be invoked on. Now it is not just *met(Y)* but a method call specified like this:

$$object.Object \times identifier.String \times return\_value.DType \times arguments.ParamList \rightarrow Method$$

The result (listing 21) shows that, like before, all the additional information is just wrapped around the detected procedure which still does nothing else than moving objects from one collection to another. The following equations display the result in a more readable notation. Note that $X1$ is an *Object*, $X2$ a *Collection*, $id1$ an *Identifier*, $dt1$ a *DType* and pp a *ParamList*.

1. $Sub1(push(X1, X2)) = call(X1, id1, dt1, pp)$
2. $Sub2(push(X1, X2)) = it\_apply(Sub5(push(X1, X2)))$
3. $Sub5(push(X1, X2)) = X2$
4. $it\_apply([]) = []$
5. $it\_apply(push(X1, X2)) = push(Sub1(push(X1, X2)), Sub2(push(X1, X2)))$

Not only have we modeled primitive object-oriented concepts - we have had *Igor* synthesize them on its own just by providing some generic input/output examples. After that we went one step further trying to model some object-oriented procedures like iteration or the ‘foreach‘ loop just by using those primitives. It turned out that *Igor* does not appear to be struggling with the example specification - even though we have wrapped them in quite a complex model - especially in the ‘foreach‘ example. We have constructed some more examples to show that we can now use our primitive objects to build a more complex model consisting solely of the concepts illustrated in this section. This means that it is possible to take this further, modeling a complete object-oriented model just with a functional programming language. At the same time it has to be said that we did indeed skip quite a lot of things as type-inference, inheritance, references (pointers) or exceptions, to name but a few. Since it has been pointed out that the model constructed in this paper does not claim to be a full scale approach we cannot conclude that a serious foundation has been created to build on.

But we have demonstrated that it is basically possible to model parts of an object oriented system functionally, which is quite an interesting observation and is definitely worth a more thorough approach.

In the next chapter we have a glance at *autoJAVA*, a plug-in for eclipse which was designed to integrate *Igor* into the eclipse workbench together with a simple way to provide input/output specifications to our system. The output of the system uses our simple protocol to generate ‘quasi-object-oriented‘ notation.

5. AutoJava

Since this paper’s focus is clearly on the theoretical part of how to design an object oriented program with a functional programming language it seems quite obvious not to get too much involved into the practical part of devising an application for this. However, *AutoJava* is a plug in for *eclipse* which basically provides a functionality to use *Igor* in an object oriented environment and as the focus of this paper is not on this prototype we are just going to have a short look at how a java file in this tool would look like:

Listing 14: Automated Solution of Last

```
/**
*@IgorMETA(
* methodName  = "last".
* retValue   = "Object".
* params     = "List".
*);
*@IgorEQ(
* equations = {
*     "([x])=x".
*     "([x,y])=y".
*     "([x,y,z])=z".
*     "([x,z,c,n])=n".
* }
*);
*@Method(last);
*/
public void last(){
  /***
  /* The following code has automatically been generated
       by AutoJava
  /* according to the user specification in the
       annotations above
  /* the result is printed below
  ***/

  //hypo(true, 2, eq 'Sub1['cons['X1:Object,'cons['X2:
       Object,
  //     'X3:List]]] = 'cons['X2:Object,'X3:List] [none] .
  //eq 'last['cons['X1:Object,'`['`].List]] = 'X1:Object [
       none] .
  //eq 'last['cons['X1:Object,'cons['X2:Object,'X3:List
       ]]] = 'last['Sub1['cons[
  //     'X1:Object,'cons['X2:Object,'X3:List]]]] [none]
       .)

}
```

In this example we can see that java annotations [3] contain the specification which is considerably simplified from what we have seen so far. The most important parts are **IgorMETA** which contains method name, return value and input arguments of the method to infer. In the second bit *IgorEQ*, the input/output can be specified.

6. Conclusion

By now we have shown that it is possible to successfully model objects, methods, properties and messages in our simple protocol, moreover, we had igor synthesize all of them. So machine learning approaches have been used in order to have a system learn how to describe generic processes within programming languages. We provided a showcase of how functional programming can be combined with object orientation. The running plug in should prove this to be true and opens up many paths for future expansion.

A rather interesting point is the integration of the specification within the annotations which creates an entry point for large-scale applications such as IBMs *RSA*. As the developer can annotate his UML diagrams and have those annotations transferred into the auto-generated code you could think of a use case like Igor using the specification during the code generation filling in the method implementation.

All in all there has to be said that even though the results presented in this paper do not seem very novel or breathtaking. But they nevertheless show that by enabling functional programs to deal with object orientation we can play to the strengths of both paradigms. Even though it has been mentioned that our model does not claim to be complete or even fully correct - it feels like that we have created an inspiration for some next steps which might gradually improve the methodology and finally result in a larger scale prototype which actually produces Java code instead of functional programs.

[3] see `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`

A. Complete Listings

A.1 Maude Specifications and some Results

Note: *Igor* frequently produces more than one output hypothesis - for the sake of transparency only one of them is listed in our results sections.

Listing 15: Object

```
fmod OBJECT is

  *** Knowledge about how objects wrap properties and
      methods
  *** Uses 'IDENTIFYER-MATCH' in the way methods are
      called on an object
  *** The same goes for property extraction

  sorts InVec Object Prop Method PropList MethodList List
        ListEl NPException .
    subsorts Method < NPException .
    subsorts Prop < NPException .
    subsorts List < PropList MethodList .
    subsorts ListEl < Prop Method .
  sorts Identifier DType .
  sort MyBool .

  *** DT definitions
  *** list to store any value
  op [] : -> List [ctor] .
  *** object constructor, taking a list of properties & a
      list of methods together with an
  *** identifier for the object
  op ___ : Identifier PropList MethodList -> Object [ctor
      ] .
  op met : Identifier DType -> Method [ctor].
  op prop : Identifier DType -> Prop .
  ops id1 id2 id3 : -> Identifier .
  op dt : -> DType .
  op exc : -> NPException .

  *** standard operations
  op cons : ListEl List -> List [ctor] .

  *** defined function names (to be induced, preds, bk)
      ***
  op mcall : Object Identifier -> Method [metadata "
      induce"] .

  var oid : Identifier .


  eq mcall( (oid [] []), id1 ) =  exc .
  eq mcall( (oid [] []), id2 ) =  exc .

  eq mcall( (oid [] cons(met(id1, dt), []) ), id1 ) = met
      (id1, dt) .
  eq mcall( (oid [] cons(met(id2, dt), []) ), id1 ) = exc
      .
  eq mcall( (oid [] cons(met(id1, dt), []) ), id2 ) = exc
      .
  eq mcall( (oid [] cons(met(id2, dt), []) ), id2 ) = met
      (id2, dt) .

  eq mcall( (oid [] cons( met(id1, dt), cons( met(id2, dt
      ), []) ) ), id1 )  = met(id1, dt) .
  eq mcall( (oid [] cons( met(id2, dt), cons( met(id1, dt
      ), []) ) ), id1 )  = met(id1, dt) .
  eq mcall( (oid [] cons( met(id2, dt), cons( met(id1, dt
      ), []) ) ), id2 )  = met(id2, dt) .

  eq mcall( (oid [] cons( met(id1, dt), cons( met(id2, dt
      ), cons( met(id3, dt), []))) ), id1 )  =
    met(id1, dt) .

  eq mcall( (oid [] cons( met(id3, dt), cons( met(id1, dt
      ), cons( met(id2, dt), []))) ), id1 )  =
    met(id1, dt) .

  eq mcall( (oid [] cons( met(id3, dt), cons( met(id2, dt
      ), cons( met(id1, dt), []))) ), id1 )  =
    met(id1, dt) .

endfm
```

Listing 16: Identifier-Match

```
fmod IDENTIFIER-MATCH is

  *** Knowledge about how methods are called by providing
      an identifyer ***
  *** If a list of methods contains the called identifyer
      , the method is returned ***

  sorts InVec List Method Identifier DType ParamList
        NPException .
    subsort Method < NPException .

  op [] : -> List [ctor] .
  op cons : Method List -> List [ctor] .
  op mm : Identifier DType ParamList -> Method [ctor] .
  ops id1 id2 id3 : -> Identifier .
  op parlist : -> ParamList [ctor] .
  op exc : -> NPException .
  op dt : -> DType .


  op match : Identifier List -> Method [metadata "induce"
      ] .

  vars m1 m2 m3 : Method .

  eq match(id1, [] ) = exc .
  eq match(id2, [] ) = exc .

  eq match(id1, cons(mm(id1, dt, parlist) ,[]) )= mm(id1,
      dt, parlist) .
  eq match(id1, cons(mm(id2, dt, parlist), []) ) = exc .
  eq match(id2, cons(mm(id1, dt, parlist) ,[]) ) = exc .
  eq match(id2, cons(mm(id2, dt, parlist), []) ) = mm(id2
      , dt, parlist) .

  eq match(id1, cons(mm(id1, dt, parlist), cons(mm(id2,
      dt, parlist), [])) ) = mm(id1, dt, parlist) .
  eq match(id1, cons(mm(id2, dt, parlist), cons(mm(id1,
      dt, parlist), [])) ) = mm(id1, dt, parlist) .
  eq match(id2, cons(mm(id2, dt, parlist), cons(mm(id1,
      dt, parlist), [])) ) = mm(id2, dt, parlist) .

  eq match(id1, cons(mm(id1, dt, parlist), cons(mm(id2,
      dt, parlist), cons(mm(id3, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .

  eq match(id1, cons(mm(id3, dt, parlist), cons(mm(id1,
      dt, parlist), cons(mm(id2, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .

  eq match(id1, cons(mm(id3, dt, parlist), cons(mm(id2,
      dt, parlist), cons(mm(id1, dt, parlist), []))) ) =
    mm(id1, dt, parlist) .


endfm
```

Listing 17: Identifier-Match Result

```
eq: match(X1,()) = exc;
ceq: Sub1(X1,cons(mm(X2,dt,parlist),X3)) = X1 if == (X1,
     X2) = false;
ceq: Sub2(X1,cons(mm(X2,dt,parlist),X3)) = X3 if == (X1,
     X2) = false;
ceq: match(X1,cons(mm(X2,dt,parlist),X3)) = match(Sub1(X1
     ,cons(mm(X2,dt,parlist),X3)),Sub2(X1,cons(mm(X2,dt,
     parlist),X3))) if == (X1,X2) = false;
ceq: match(X1,cons(mm(X2,dt,parlist),X3)) = mm(X1,dt,
     parlist) if == (X1,X2) = true;
```

Listing 18: OO-Call

```
fmod OO−CALL is

  sorts InVec Object .
  sorts Message ParamList .
  sorts Nat Bool Param Res .
    subsorts Param < Nat Bool .
    subsorts Res < Nat Bool .
    subsorts Object < Nat Bool .


  *** DT definitions
  op * : −> Object [ctor] .
  op <> : −> ParamList [ctor] .
  op msg : ParamList Object −> Message [ctor] .
  op null : −> Object [ctor] .


  op 0 : −> Nat [ctor] .
  op s : Nat −> Nat [ctor] .
  op t : −> Bool [ctor] .
  op f : −> Bool [ctor] .

  *** Standard Operators ***
  *** op call : Message −> Message [metadata "pred␣
       nomatch"] . ***
  op cpar : Param ParamList −> ParamList [ctor] .


  *** defined function names (to be induced, preds, bk)
       ***
  op method : Message −> Message [metadata "induce"] .

  *** input encapsulation ***
  op in : Message −> InVec [ctor] .

  vars pl : ParamList .
  vars n : Nat .

  *** input output examples for "even" ***
  eq method( msg(cpar( 0, <>), null) ) = msg(<> ,t ) .
  eq method( msg(cpar( s(0), <>), null) ) = msg( <> , f
        ) .
  eq method( msg(cpar( s(s(0)), <>), null) ) = msg( <> ,
       t ) .
  eq method( msg(cpar( s(s(s(0))), <>), null) ) = msg( <>
        , f ) .
  eq method( msg(cpar( s(s(s(s(0)))), <>), null) ) = msg(
        <> , t ) .

endfm
```

Listing 19: OO-Call Result

```
eq: method(msg(cpar(X1,<>),null)) = msg(<>,f) if == (X1,s
      (0)) = true AND == (X1,s(0)) = true AND == (X1,s(0))
       = true;
ceq: method(msg(cpar(X1,<>),null)) = msg(<>,f) if == (X1,
      s(s(s(0)))) = true AND == (X1,s(s(0)))) = true AND
      == (X1,s(s(0)))) = true AND == (X1,s(s(s(0)))) =
      true AND == (X1,s(s(0)))) = true;
ceq: method(msg(cpar(X1,<>),null)) = msg(<>,t) if == (X1,
      s(s(s(0)))) = false;
```

Listing 20: Iterate-Collection

```
fmod ITERATE−COLLECTION is

  sorts Object Collection ResultCollection Method Result
       InVec .

  *** DT definitions (constructors)
  op [] : −> Collection [ctor] .
  op {} : −> ResultCollection [ctor] .
  op put : Object Collection −> Collection [ctor] .
  op put2 : Result ResultCollection −> ResultCollection [
       ctor] .
  op met : Object −> Result .
  *** defined function names (to be induced, preds, bk)
  op iterate : Collection −> ResultCollection [metadata "
       induce"] .
  *** input encapsulation
  op in : Collection −> InVec [ctor] .

  vars U V W X Y Z F : Object .


  eq iterate([]) = {} .
  eq iterate( put(Y,[]) ) = put2( met(Y), {}) .
  eq iterate( put(X,put(Y,[])) ) = put2( met(X), put2(
       met(Y) ,{})) .
  eq iterate( put(Y,put(X,put(Z,[]))) ) = put2( met(Y),
       put2( met(X), put2( met(Z),{}))) .

endfm
```

Listing 21: Iterate-Collection Result

```
eq: Sub1(put(X1,X2)) = met(X1);
eq: Sub2(put(X1,X2)) = iterate(Sub5(put(X1,X2)));
eq: Sub5(put(X1,X2)) = X2;
eq: iterate(()) = {};
eq: iterate(put(X1,X2)) = put2(Sub1(put(X1,X2)),Sub2(put(
      X1,X2)));
```

Listing 22: Foreach-Do

```
fmod FOREACH−DO is

  sorts InVec Object Prop Method PropList MethodList List
       ListEl ParamList Collection .
    subsorts List < PropList MethodList .
    subsorts ListEl < Prop Method .
    subsorts Object < Method .
  sorts Identifier DType .

  *** DT definitions (constructors) ***
  op [] : −> Collection [ctor] .
  op pp : −> ParamList .


  *** STANDARD OPERATORS ***
  op push : Object Collection −> Collection [ctor] .

  *** METHOD DECLARATION ***
  op call : Object Identifier DType ParamList −> Method [
       ctor].
  op id1 : −> Identifier [ctor] .
  op dt1 : −> DType [ctor] .

  *** defined function names (to be induced, preds, bk)
       ***
  op it_apply : Collection −> Collection [metadata "
       induce"] .

  *** input encapsulation ***
  op in : Collection −> InVec [ctor] .

  *** VARIABLES ***
  vars a b c : Object .


  *** ITERATION SPECIFICATION ***
  eq it_apply([]) = [] .

  eq it_apply( push(a, []) ) = push( call(a, id1, dt1, pp
       ), [] ) .

  eq it_apply( push(a, push(b, [])) ) =
    push( call(a, id1, dt1, pp), push( call(b, id1, dt1,
         pp), []) ) .

  eq it_apply( push(a, push(b, push(c, []))) ) =
    push( call(a, id1, dt1, pp), push( call(b, id1, dt1,
         pp), push( call(c, id1, dt1, pp), [])) ) .
```

endfm

References

Thomas Hieber. Transportation of the JEdit plug-in ProXSLbE to eclipse. Technical report, Otto Friedrich University of Bamberg, 2008. URL `http://www.cogsys.wiai.uni-bamberg.de/effalip/data/programs/autoXSL/ausarbeitung/projektbericht.pdf`.

Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005*, volume 6, pages 111–126. Intellect, 2007. URL `http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/14num.pdf`.

Oleg Kiselyov and Ralf Laemmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005. URL `http://dblp.uni-trier.de/db/journals/corr/corr0509.html#abs-cs-0509027`. informal publication.

Emanuel Kitzelmann. Data-driven induction of recursive functions from I/O-examples. In Emanuel Kitzelmann and Ute Schmid, editors, *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP'07)*, pages 15–26, 2007.

Emanuel Kitzelmann. Analytical inductive functional programming. In *Pre-Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*. Michael Hanus, 2008.

Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006. ISSN 1533-7928.

Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995. URL `citeseer.ist.psu.edu/muggleton95inverse.html`.

Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990. URL `citeseer.ist.psu.edu/muggleton90efficient.html`.

Roland J. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1): 55–83, 1995.

J. Ross Quinlan and R. Mike Cameron-Jones. FOIL: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning,Proceedings*, volume 667, pages 3–20. Springer-Verlag, 1993. URL `citeseer.ist.psu.edu/quinlan93foil.html`.

Didier Remy and Jrme Vouillon. Objective ML: An effective object-oriented extension to ML, 1998.

Ute Schmid. *Inductive Synthesis of Functional Programs – Learning Domain-Specific Control Rules and Abstract Schemes*. Number 2654. Springer, 2003.

P. D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.

Exhaustive program generation by interpretation of Herbelin's LJT variant

Susumu Katayama
University of Miyazaki
skata@cs.miyazaki-u.ac.jp

Abstract

This paper reports the on-going research on formalization of the algorithm behind MagicHaskeller, an inductive functional programming system based on systematic search, as a monadic interpretation of inference rules of a variant of Herbelin's LJT.

1. Introduction

In TFP2005 symposium we presented an algorithm to generate a stream of all the possible expressions with a given type from a given set of expressions by using breadth-first search [Katayama 2005], and since then the algorithm has been released and updated as a generate-and-test style inductive functional programming system, called MagicHaskeller. The research, started as an antithesis to genetic programming that tends to ignore systematic search, has attracted some positive-minded scientists, while its lack in enough formalization has been a source of difficulty in understanding and theoretically manipulating the algorithm.

In the same year of 2005 Augustsson released Djinn, that generates one or some functional programs with the given type, based on theorem proving by Roy Dyckhoff's LJT [Dyckhoff 1992] (a.k.a. G4ip by Hudelmaier [Hudelmaier 1992]). That has driven us to work on formalization of our exhaustive search algorithm, though Dyckhoff's LJT cannot straightforwardly be applied to exhaustive program/proof generation because it replaces equivalent expressions for making its $\supset\Rightarrow_4$ rule efficient, but actually replacing equivalent expressions corresponds to several proofs, and thus should multiply the number of proofs.

As the matter of fact, there is also a correspondence between automatic proof and the algorithm behind MagicHaskeller. This on-going research starts with monadic interpretation of each rule of a variant of Herbelin's LJT[Herbelin 1995] in combination of Spivey's algebraic framework for combinatorial search[Spivey 2006], and aims at associating our algorithm presented at [Katayama 2005] with generation of the stream of all the proofs for the given proposition.

2. Basic ideas

We use Spivey's monad for combinatorial search[Spivey 2006]. By using his algebraic interface, we can easily implement combinatorial search using the bind operator $\triangleright$ for combinations and plus operator $\oplus$ for alternative selections. Hence, when generating proofs of some proposition, if the proposition matches conclusions of plural inference rules, we only need to generate proofs of premises of such rules and compute the direct sum of such processes (which means backtracking when generating a single proof), and if the proposition matches the conclusion of an inference rule which has plural premises, we only need to generate proofs of the premises and compute the direct product of such processes. In general, such alternation and combination of proof tree generation can be interpreted in the way shown in Table 1, where $\mathcal{E}[X]$ denotes the monadic value holding the infinite stream of proofs of X, and $\langle\otimes\rangle$ denotes a multiplication, whose definition is dependent on how to construct a pair of proofs. $\langle\otimes\rangle$ can be defined in Haskell as follows using monadic operators defined in [Spivey 2006]:

$$x\langle\otimes\rangle y = x \triangleright \lambda a \rightarrow y \triangleright \lambda b \rightarrow \mathit{return}\ (a \otimes b)$$

i.e.

$$x\langle\otimes\rangle y = \mathit{liftM2}\ (\otimes)\ x\ y$$

Table 1. interpretation of alternatives and combinations

rules	interpretations
$\dfrac{B}{A}$ $\qquad$ $\dfrac{C}{A}$	$\mathcal{E}[A] = \mathcal{E}[B] \oplus \mathcal{E}[C]$
$\dfrac{x :: B \qquad y :: C}{x \otimes y :: A}$	$\mathcal{E}[A] = \mathcal{E}[B]\langle\otimes\rangle\mathcal{E}[C]$

3. A simple, but more concrete example

Let us consider a more concrete and more interesting inference rule set, with which we can indeed generate an infinite set of λ-expressions. Consider the inference rule set shown in Table 2.

We assume that each premise is not a list but a set.

The rule set can be used to infer the types of λ-expressions. Also, it can be used to mechanically generate a proof of a

Table 2.

$$\frac{\Gamma, x :: A; \vdash e :: B}{\Gamma \vdash \lambda x.e :: A \to B} \to\text{R}$$

where x does not appear as a label in Γ

$$\frac{\Gamma \vdash e_1 :: A_1 \quad ... \quad \Gamma \vdash e_n :: A_n}{\Gamma, f :: A_1 \to ... \to A_n \to B \vdash f e_1 ... e_n :: B} \text{Ax+}\!\to\!\text{L}$$

where B is atomic

proposition by matching from the bottom to the top, and we are interested in generating the infinite set of all the proofs.

The proviso for Ax+→L rule is added in order to permit only η-long normal form and identify η-equivalent proofs. This limitation does not only limit the unnecessary search space expansion by η-equivalent program generations when generating programs (proofs) from the bottom to the top, but also make implementation simpler and more efficient by preventing the proposition to be proved from matching plural conclusions of inference rules.

In order to show the implementations, we first define a datatype of λ-expressions:

```
data Expr = Expr :$Expr       -- function application
     | Lambda Var Expr        -- lambda abstraction
     | V Var                  -- variable
```

Then, the rule set can be interpreted as follows.

$$\mathcal{E}[\Gamma \vdash A \to B] = \langle \lambda x. \rangle (\mathcal{E}[x :: A, \Gamma \vdash B]) \qquad (1)$$

$$\mathcal{E}[\Gamma \vdash B] = \bigoplus_{f :: T \in \Gamma} \mathcal{E}[\Gamma; f :: T \vdash B], \qquad \text{if } B \text{ is atomic.} \qquad (2)$$

$$\mathcal{E}[\Gamma; f :: A_1 \to ... \to A_n \to B \vdash B] = wrap^n \; (\\ return(V f) \; \langle :\$ \rangle \; \mathcal{E}[\Gamma \vdash A_1] \; \langle :\$ \rangle \; ... \; \langle :\$ \rangle \; \mathcal{E}[\Gamma \vdash A_n] \\ ) \qquad (3)$$

$$\mathcal{E}[\Gamma; f :: T \vdash B] = zero, \qquad \text{if } T \text{ does not return } B. \qquad (4)$$

We assume $\langle :\$ \rangle$ is left associative. $zero$ means the empty set. $\langle \lambda u. \rangle(m)$ means mapping λ-abstraction by u of each element of m, i.e.

$\langle \lambda u. \rangle \; (m) = fmap \; (Lambda \; u) \; m$

or

$\langle \lambda u. \rangle \; (m) = m \triangleright (return \circ Lambda \; u)$

Rule →R is straightforwardly interpreted to Equation 1. As for the atomic case corresponding to Ax+ →L, the interpretation should be the $\oplus$ sum of all the possible choices of $f \in \Gamma$ as in Equation 2, that return the same type as the requested one.

We silently insert the $wrap$ operation from [Spivey 2006], that pushes the search process deeper in the search tree, into the product operation, which corresponds to function application. This is done because we regard the search depth as the program (proof) size, measured by the number of function applications. [1]

[1] The $wrap$ operation can be included in $\langle :\$ \rangle$, though for efficiency it should be applied as soon as the arity is known.

4. Interpretation of Cut-free LJT

The rule set shown in Table 2 can equivalently be translated to Herbelin's cut-free LJT[Herbelin 1995], except for the proviso that prevent η-equivalent expressions from being generated. Without the proviso, the interpretation becomes as follows:

$$\mathcal{E}[\Gamma \vdash A \to B] = \langle \lambda x. \rangle (\mathcal{E}[x :: A, \Gamma \vdash B]) \\ \oplus \bigoplus_{f :: T \in \Gamma} \mathcal{E}[\Gamma; f :: T \vdash A \to B]$$

$$\mathcal{E}[\Gamma \vdash B] = \bigoplus_{f :: T \in \Gamma} \mathcal{E}[\Gamma; f :: T \vdash B], \quad \text{if } B \text{ is atomic.}$$

5. Remaining work

We have not presented the structural rules for bundling the variables with the same type, which are useful for efficient implementation, especially in combination with memoization. Neither have we presented rules related to universal quantification, for which we implement unification algorithm on the top of state monad transformer that keeps track of the current substitution, which transforms Spivey's monad of combinatorial search.

Our recent work on removing semantically equivalent expressions [Katayama 2008] requires more complicated implementation, and we have not proved the exhaustiveness under such circumstances. We hope that further formalization will reinforce this line of research.

References

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, pages 795–807, 1992.

Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75, 1995.

J. Hudelmaier. Bounds on cut-elimination in intuitionistic propositional logic. *Archive for Mathematical Logic*, 31:331–354, 1992.

Susumu Katayama. Systematic search for lambda expressions. In *Sixth Symposium on Trends in Functional Programming*, pages 195–205, 2005.

Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In Tu Bao Ho and Zhi-Hua Zhou, editors, *PRICAI*, volume 5351 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2008. ISBN 978-3-540-89196-3.

Michael Spivey. Algebras for combinatorial search. In *Workshop on Mathematically Structured Functional Programming*, 2006.

Quick filtration of semantically equivalent expressions in program search results

Susumu Katayama
University of Miyazaki
skata@cs.miyazaki-u.ac.jp

Abstract

This research is on improving the efficiency of our previous work for removing semantically equivalent expressions in program search results.

1. Introduction

In [Katayama 2008] we have proposed a Las Vegas algorithm for removing all the semantically equivalent programs except one by Monte-Carlo search within the program space, for the purposes of

- speeding up exhaustive search by bootstrapping,
- improving the readability of search results (like search engines such as Google which bundle "similar pages")
- providing guesses on how quickly the search space bloats

We tried generating an infinite stream of all the typed λ-expressions with the given type that consist of given primitive set, in the increasing order of program size. The results were impressing – the algorithm's estimation of the number of functions with type $\forall\, a.\ [a] \ \rightarrow\ [a]$ which consist of nil, cons, and foldr, and is constructed with λ-abstractions and 10 or less function applications was below a hundred. Moreover, even when we used a set of tens of library functions as the primitive set, there were only hundreds of functions with type $\forall\, a.\ [a] \ \rightarrow\ [a]$, which is constructed with λ-abstractions and 7 or less function applications. Those interesting results suggest a new possibility of search-based non-heuristic inductive functional programming.

On the other hand, the filtration algorithm requires more computational cost than that is expected from the fewness of the final result, because it is dependent on execution of huge amount of expressions generated. Especially to our regret, if the set of programs to be filtered is not very redundant, i.e., if it does not include a lot of semantically equivalent expressions (e.g. when using the primitive set with only constructors and induction functions), program generation with such filtration costs more time than that without it. Hence we improve the efficiency of the filtration algorithm.

Evaluating semantical equivalence of syntactically different expressions by supplying random arguments is not a new idea. [Martin 1971] discussed a method that guesses the equivalence of two algebraic expressions by evaluating them using finite field arithmetic. Monte-Carlo search for program errors is called random testing in the field of software engineering. Due to the space limitation, interested readers are referred to references in [Katayama 2008].

2. The old algorithm

Monte-Carlo algorithms are randomized algorithms whose final results may be inaccurate, while Las Vegas algorithms always yield the correct answers if they halt, though their computational costs are random and unknown before execution. It is often the case that a Monte-Carlo algorithm can be converted into a Las Vegas algorithm by repetition until the correct answer is obtained, especially when we can tell if the obtained result is correct or not.

Our algorithm presented in [Katayama 2008] is based on a similar idea, though we cannot exactly decide if the obtained infinite stream exhaustively include semantically different expressions. Here is the rough sketch of the filtration algorithm presented in [Katayama 2008]:

- let S_d the search result until depth d; define equivalence by a random point set r: $f \sim_r g \overset{def}{\Leftrightarrow} \forall p \in r. f(p) = g(p)$;
- generate a stream of random point sets $\{r_d\}_{d=0,1,\ldots}$;
- compute the quotient set $S_d \ / \sim_{r_d}$ and the complete set of representatives; expressions without uniqueness proof will simply be dropped;
- use iterative deepening, and refine $\sim_{r_d}$ at each iteration by letting $r_d \subset r_{d+1}$; thanks to the deepening, dropped but distinct expression will resurrect.

The above algorithm requires at least nb times of executions when filtering n expressions for computing the depth b, because each expression has to be executed for b or more random points. Obviously this is inefficient and rather deteriorates the efficiency, because the number of expressions exponentially increases as the iteration goes deeper. For this reason, when applying this filter to subexpressions during program generation for efficiency, [Katayama 2008] uses a more efficient filter which permits minor redundancy that uses

Table 1. Experimental results. (o.m. means out of memory, and ∞ means no result in an hour.)

	time (sec)	number of programs at each depth									
generating $Int \to Int$ from the $mnat$ primitive set until depth 10.											
not filtered (redundant)	3.5	2	2	6	22	78	326	1506	7910	44806	283014
with old Filter 2	∞	2	2	3	4	12	27	62	146	448	N/A
with old Filter 1 + Filter 2	24	2	2	3	4	12	27	28	107	282	842
with the new filter	11	2	2	3	4	12	27	52	109	321	1009
generating $[Char] \to [Char]$ from the $reallyall$ primitive set until depth 9.											
not filtered (redundant)	o.m.	2	5	42	225	1755	12228	98034	771730	N/A	
with old Filter 2	o.m.	2	1	3	13	21	113	299	1082	N/A	
with old Filter 1 + Filter 2	72	2	1	3	12	21	98	264	981	3692	
with the new filter	51	2	1	5	18	35	160	422	1611	6256	

only a small fixed number of random points per expression, and then apply the above inefficient but not redundancy-permitting filter to the final result. Its idea is that the set of expressions is already thinned up by the efficient filter when the inefficient filter is applied, and thus the usage of the inefficient filter is not the bottleneck any longer.

By using this two-staged filtration, the total computational cost using a rich primitive set have reduced from the original algorithm not using such filtrations and the one using only the above inefficient filter. However, when using minimal primitive sets that consist of constructors and induction functions, the algorithm using the two-staged filtration is still slower than the original algorithm without filtrations. In this research we seek more efficient filtration process.

3. The improved algorithm

We focus on improving the execution time of filtration (which includes that of the generated (sub)expressions) rather than that of program synthesis, because the time profiling reports show that most of the total computation time is comprised of that part. Two-staged filtration is dependent on iterative deepening, and when new expressions are generated, expressions at the shallower nodes in the search tree are re-executed with a different random point set as the argument, and re-categorized into new equivalence classes based on it, along with the newly generated expressions. The new improved algorithm omits the re-execution.

Given the set of expressions before filtration at depth d as x_d, the resulting set y_d of expressions at depth d after the first filtration of the two-staged filtration was

$$\begin{aligned} y_0 &= pick(S_0 \mathbin{/} \sim_{r_0}) \\ y_d &= pick((S_d \mathbin{/} \sim_{r_d}) \setminus_{\sim_{r_{d-1}}} (S_{d-1} \mathbin{/} \sim_{r_{d-1}})) \end{aligned}$$

where $pick$ is a function that collects representatives from equivalence classes, $\setminus_r$ is an operator for set subtraction by using the equivalence defined by the point set r, and $S_d = \bigcup_{i=0}^{d} x_i$.

The new algorithm computes y_d as

$$y_d = pick \left(\bigcup_{i=0}^{d} ((x_{d-i} \mathbin{/} \sim_{r_i}) \setminus_{\sim_{r_i}} (S_{d-i-1} \mathbin{/} \sim_{r_i})) \cup x_0 \mathbin{/} \sim_{r_d} \right)$$

The ideas are:

- the computation of $S_d \mathbin{/} \sim_{r_d}$ costs $|S_d||r_d|$ because r_d includes all the points in $r_i|_{i<d}$; thus we want to use r_0 instead of r_d for computing x_d;
- Because $r_i|_{i>0}$ is a refinement of r_0, representatives of $x_{d-i} \mathbin{/} \sim_{r_0}|_{i>0}$ should fall into different equivalence classes of $x_{d-i} \mathbin{/} \sim_{r_i}|_{i>0}$, and in order to avoid duplicates we subtract $S_{d-1} \mathbin{/} \sim_{r_0}$.

Also, by starting with small number of random points, we can keep the random values small at first, by which we can expect edge and corner cases to be checked often.

4. Experimental results

Table 1 shows some results on the $mnat$ primitive set and the $reallyall$ primitive set defined in [Katayama 2008]. $mnat$ consists only of 0, successor function, and addition and paramorphism for natural numbers. $reallyall$ is rather a large primitive set consisting of 5 boolean operations, 12 instances of (in)equality predicates, and 24 list operations taken from the Standard Prelude. All the experiments are conducted using the same parameters as [Katayama 2008] on Intel Pentium D 2.8.GHz machine running Linux 2.6.24.

The proposed filter requires less time for generating more programs than the old filter.

References

Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In Tu Bao Ho and Zhi-Hua Zhou, editors, *PRICAI*, volume 5351 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2008. ISBN 978-3-540-89196-3.

William A. Martin. Determining the equivalence of algebraic expressions by hash coding. *Journal of the Association for Computing Machinery*, 18(4):549–558, 1971.

Author Index

Bamberger Beiträge zur Wirtschaftsinformatik

Stand August 26, 2009

Nr. 1 (1989)	Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)
Nr. 2 (1990)	Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG
Nr. 3 (1990)	Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen
Nr. 4 (1990)	Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)
Nr. 5 (1990)	Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)
Nr. 6 (1991)	Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten
Nr. 7 (1991)	Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender
Nr. 8 (1991)	Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung
Nr. 9 (1992)	Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell
Nr. 10 (1992)	Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)
Nr. 11 (1992)	Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells
Nr. 12 (1992)	Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt
Nr. 13 (1992)	Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell
Nr. 14 (1992)	Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen
Nr. 15 (1992)	Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch
Nr. 16 (1992)	Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip
Nr. 17 (1993)	Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation

Nr. 18 (1993)	Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells
Nr. 19 (1994)	Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach
Nr. 20 (1994)	Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994
	Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994
Nr. 21 (1994)	Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen
Nr. 22 (1994)	Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems
Nr. 23 (1994)	Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle
Nr. 24 (1994)	Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen
Nr. 25 (1994)	Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme
Nr. 26 (1995)	Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes
Nr. 27 (1995)	Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995
Nr. 28 (1995)	Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach
Nr. 30 (1995)	Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit
Nr. 31 (1995)	Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse
Nr. 32 (1995)	Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger
Nr. 33 (1995)	Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?
Nr. 34 (1995)	Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -
Nr. 35 (1995)	Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme
Nr. 36 (1996)	Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996)	Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996
Nr. 38 (1996)	Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996
Nr. 39 (1996)	Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze
Nr. 40 (1997)	Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997
Nr. 41 (1997)	Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997
Nr. 42 (1997)	Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997
Nr. 43 (1997):	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997
	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998
Nr. 44 (1997)	Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung
Nr. 45 (1998)	Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998
Nr. 46 (1998)	Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997
Nr. 47 (1998)	Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998
Nr. 48 (1998)	Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering
Nr. 49 (1998)	Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems

Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460

Nr. 50 (1999) Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)

Nr. 51 (1999) Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)

Nr. 52 (1999) Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule" im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999

Nr. 53 (1999) Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999

Nr. 54 (1999) Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.

Nr. 55 (2000) Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur

Nr. 56 (2000) Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000

Nr. 57 (2000) Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.

Nr. 58 (2000) Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.

Nr. 59 (2001) Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001

Nr. 60 (2001) Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001" im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Note: The title of our technical report series has been changed from *Bamberger Beiträge zur Wirtschaftsinformatik* to *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* starting with TR No. 61