Ute Schmid
Emanuel Kitzelmann
Rinus Plasmeijer (Eds.)

LNCS 5812

Approaches and Applications of Inductive Programming

Third International Workshop, AAIP 2009
Edinburgh, UK, September 2009
Revised Papers

Springer

Ute Schmid Emanuel Kitzelmann
Rinus Plasmeijer (Eds.)

Approaches and Applications of Inductive Programming

Third International Workshop, AAIP 2009
Edinburgh, UK, September 4, 2009
Revised Papers

Springer

Volume Editors

Ute Schmid
Emanuel Kitzelmann
Otto-Friedrich-Universität Bamberg
Fakultät Wirtschaftsinformatik und Angewandte Informatik
96045 Bamberg, Germany
E-mail: {ute.schmid,emanuel.kitzelmann}@uni-bamberg.de

Rinus Plasmeijer
Radboud University Nijmegen
Institute for Computing and Information Sciences
6525AJ Nijmegen, The Netherlands
E-mail: rinus@cs.ru.nl

Library of Congress Control Number: 2010923416

CR Subject Classification (1998): I.2, D.2, F.3, H.3, D.3, F.4.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-11930-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-11930-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

To Phil Summers
who laid the foundation

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 generalizing 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 generalized 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 generalized 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 few years, some work on exploiting induction techniques has been started also in the functional programming community. Therefore, the third workshop on "Approaches and Applications of Inductive Programming" took 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 2009 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. Accordingly, the organizers as well as the Program Committee and the reviewers consisted of members from both communities.

The workshop was enriched by three invited talks from members of 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 organization. Furthermore, we want to thank all presenters for submitting their work to our workshop and all attendants for stimulating discussions.

We are proud that all authors of accepted workshop papers as well as two of the invited speakers provided revised papers for this proceedings publication. Thereby we can present a rather representative selection of current research in the field of inductive programming. For everybody interested in inductive programming, we recommend visiting the website
www.inductive-programming.org.

December 2009

Ute Schmid
Emanuel Kitzelmann
Rinus Plasmeijer

Organization

Organizing Committee

Ute Schmid	University of Bamberg, Germany
Emanuel Kitzelmann	University of Bamberg, Germany
Rinus Plasmeijer	Radboud University Nijmegen, The Netherlands
Technical Support	Martin Hofmann, University of Bamberg, Germany

Program Committee

Pierre Flener	Uppsala University, Sweden
Lutz Hamel	University of Rhode Island, Kingston, USA
Jose Hernandez-Orallo	Technical University of Valencia, Spain
Johan Jeuring	University of Utrecht, The Netherlands
Susumu Katayama	University of Miyazaki, Japan
Pieter Koopman	Radboud University Nijmegen, The Netherlands
Oleg G. Monakhov	Russian Academy of Sciences, Siberian Branch, Russia
Ricardo Aler Mur	Universidad Carlos III de Madrid, Spain
Roland Olsson	Ostfold College, Norway
Maria José Ramírez Quintana	Technical University of Valencia, Spain

Board of Reviewers

All members of the Organizing Committee and of the Program Committee served as reviewers for the workshop and the proceeding submissions. In addition, we thank the following external reviewers:

Wolfgang Jeltsch	BTU Cottbus, Germany
Janis Voigtländer	University of Bonn, Germany

Table of Contents

Deriving a Relationship from a Single Example

Neil Mitchell

http://community.haskell.org/~ndm

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 [22], *type classes* [29] 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. [27] 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 Languge 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 lists, and each of 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 [30], 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:

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

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 the instance above – for example the relationship might always generate an 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 these 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 (§3).
- We outline a method for deriving a relationship in our DSL, without resorting to unguided search (§4).
- We give results (§5), including reasons why our inference fails (§5.1). 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 relationships, 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 value of a DSL (domain specific language). The person defining a derivation scheme is required to define suitable types named Input, Output and the DSL. To use a value of DSL, we need an apply function to serve as an interpreter, which takes a DSL value and an input and produces an output:

$\mathsf{apply} :: \mathsf{DSL} \rightarrow \mathsf{Input} \rightarrow \mathsf{Output}$

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 [28]:

$\mathsf{sample} :: \mathsf{Input}$
$\mathsf{derive} \; :: \mathsf{Output} \rightarrow [\mathsf{DSL}]$

We require instantiations of 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:

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

Note that given an incorrect derive function it is possible to create a correct derive function by simply filtering out the incorrect results – correctness can be tested inside the derive function.

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 be confident the relationship matches their intuition. 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. $\mathsf{apply\ d_1} \equiv \mathsf{apply\ d_2}$.

$\mathsf{d_1} \cong \mathsf{d_2} \Longleftrightarrow \forall\, \mathsf{i} \in \mathsf{Input} \bullet \mathsf{apply\ d_1\ i} \equiv \mathsf{apply\ d_2\ i}$

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 which give the same answer on the sample input are congruent:

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

The combination of this predictability property and the correctness property from §2.1 implies the consistency property. It is important to note that predictability does not impose conditions on the derive function, only on the DSL, the apply function and the sample input.

2.3 Scheme Roles

The creation and use of a derivation scheme can be split into separate roles, perhaps completed by different people, each focusing on only a few aspects of the scheme.

The scheme creator defines the Input, Output and DSL types, the apply function, and the sample value of type Input (§3). Their choice must satisfy the predictability property (§3.4).

The derivation function author defines the derive function (§4). They may choose to ensure the correctness property, or filter the results. They do not need to concern themselves with predictability.

The relationship creator gives an appropriate output based on the previously defined sample input (§1 and examples throughout). In order to ensure their relationship matches their intuition, they may wish to familiarise themselves with some details of the DSL, but hopefully these will not be too onerous.

The relationship user simply gives an input, and receives an output – the output will always be what the relationship creator intended.

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 [30] 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 representation of Haskell source code is 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 [24]. 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 make it easier to express some operations 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 §7.

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 [16,17].

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 [2]. 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]]. These constructed values represent the AST defined by haskell-src-exts, so can represent all of Haskell – e.g. a case expression would be OApp "Case" [on, alts].

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 [17], 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

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