Porting IgorII from Maude to Haskell

Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid

Cognitive Systems Group, University of Bamberg
{martin.hofmann,emanuel.kitzelmann,ute.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 especially the treatment of code as data and vice versa, can be simulated in Haskell using a stateful monad transformer which makes type and class information available. 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.

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 manifold. 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 (1).

Beginnings of IP research addressed inductive synthesis of functional programs from small sets of positive I/O examples only (2). One of the most influential classical systems was Thesys (3) 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 (4), and Igor 2 (5) and by the evolutionary/generate-and-test based approaches Adate (6) and MagicHaskeller (7).

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

U. Schmid, E. Kitzelmann, and R. Plasmeijer (Eds.): AAIP 2009, LNCS 5812, pp. 140–158, 2010.

such as minimality of the generalisation w.r.t. to the given examples and termination. However they are restricted to programs describable 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 synthesis able 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 (8), Golem (9), Progol (10), and the interactive system Dialogs (11). Synthesis of functional logic programs is covered by the system Flip (12).

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 (13). 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 term rewriting 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 (14) 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. Please note that in the following chapter we adopt the MAUDE syntax, where, contrary to HASKELL variables are in upper case and constructor symbal are in lower case.

2.1 The Igor 2-Algorithm

We only sketch the algorithm here. For a more detailed description see (14).

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 a set of directed equations or rules of the form $F(p_1, \ldots, p_n) \rightarrow t$ where F is a defined function symbol, the p_i are constructor terms and t is a term. This corresponds to *pattern matching* over user-defined data types 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, i.e. as non-recursive I/O examples.

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. We will explain all of them in the following paragraphs.

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

$$\begin{aligned} reverse([]) &= [] \\ reverse([X]) &= [X] \\ reverse([X,Y]) &= [Y,X] \end{aligned}$$

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. A defined function is either the target function, a function from the background knowledge, or an auxiliary function invented on the fly. 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:

$$reverse([Q|Qs]) = [Q2|Qs2] \tag{1}$$

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 $sub2$:

$$\begin{array}{lcl} sub1([X]) & = & X \\ sub1([X,Y]) & = & Y \\ sub2([X]) & = & [] \\ sub2([X,Y]) & = & [X] \end{array}$$

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:

$$sub2([Q|Qs]) = Qs2 \tag{2}$$

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 *sub*3.

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 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 placeholders (`_`) 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,

[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.

```
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 accessible 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) (15) 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 in the following..

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 Haskellspecific 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
```

[2] aka `1:2:[]`. or `[1,2]`.

```
-- 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")))
```

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 maps to its type
  , sctx_ctors    :: (M.Map Name Type)
  -- constructor name maps to its type
  , sctx_classes  :: (M.Map Name [Name])
  -- class name maps to its superclasses
  , sctx_members  :: (M.Map Name [Name])
  -- class name maps to member functions names
  , sctx_instnces :: (M.Map Type [Name])
  -- type maps to 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 (16), and so do we. While we are at it we can start stacking monads with monad transformers (17) 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, the expressions **"Peano"**`lookIn sctx_classes` yields 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 `TExp`s 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 `TExp`s 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. Let `CallDep` be the type of a calling dependency, which encapsualtes the information which function calls which.

```
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 their syntactic size, thus the dependency could be of either type `LT`, `EQ`, or `GT` ($\overset{<}{\longrightarrow}$, $\overset{=}{\longrightarrow}$, $\overset{>}{\longrightarrow}$).

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:

$$\begin{aligned} f \overset{>}{\longrightarrow} g &\Rightarrow g \text{ is not allowed to call } f \\ f \overset{=}{\longrightarrow} g &\Rightarrow g \overset{<}{\longrightarrow} f \\ f \overset{<}{\longrightarrow} g &\Rightarrow g \overset{<}{\longrightarrow} f \text{ or } g \overset{=}{\longrightarrow} f \\ f = g &\Rightarrow f \overset{<}{\longrightarrow} f \end{aligned}$$

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 partitions, 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 $\rho_1 \dots \rho_n$ 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 $R' = R \setminus \{r\} \cup \{\rho_1(r), \dots, \rho_n(r)\}$, thus changing H to $H' = H \setminus \{h | r \in h\} \cup \{h_i | h_i = h \setminus \{r\} \cup \rho_i(r)\}$ for $i = 1 \dots n$.

This makes the implementation of our search approach lack elegance when compared to breadth-first search combinators proposed by Spivey (18; 19), where the space for breadth-first search can be defined as an infinite list. Katayama for example efficiently uses this approach (20; 7), 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 hypotheses. Rather it is refined 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, we use a stateful transformer on top of our `C` monad.

```
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* explicitly needs to it to be solvable.

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.

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 insertion into a sorted list,
swap	changes the position of two consecutive elements 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.

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 speedup by the factor of 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 2. Runtimes on different problems in seconds

	Igor 2	Igor 2_H	$\frac{\text{Igor 2}}{\text{Igor 2}_H}$ *		Igor 2	Igor 2_H	$\frac{\text{Igor 2}}{\text{Igor 2}_H}$ *
add	0.236	0.076	3	*repeatfst*	0.052	0.004	13
append	46.338	0.080	579	*repeatlst*	0.100	0.004	25
drop	0.084	0.004	21	*reverse*	0.617	0.032	19
evenpos	0.056	0.004	14	*shiftl*	0.084	0.008	11
init	0.024	0.004	6	*shiftr*	0.308	0.020	15
last	0.024	0.001	24	*sort*	0.148	0.012	12
lasts	6.744	0.020	337	*swap*	0.108	0.008	14
length	0.028	0.001	28	*switch*	2.536	0.036	70
odd/even	0.080	0.004	20	*take*	1.380	0.012	115
oddpos	18.617	0.048	388	*weave*	0.348	0.036	13

* rounded to nearest proper fraction

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 usable 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 when applicable. In this context we will make use of type information which is now accessible. 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.

References

[1] Hofmann, M.: Automatic Construction of XSL Templates – An Inductive Programming Approach. VDM Verlag, Saarbrücken (2007)
[2] Biermann, A.W., Kodratoff, Y., Guiho, G.: Automatic Program Construction Techniques. The Free Press, NY (1984)
[3] Summers, P.D.: A methodology for LISP program construction from examples. Journal ACM 24, 162–175 (1977)

[4] Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
[5] Kitzelmann, E.: Data-driven induction of recursive functions from I/O-examples. In: Kitzelmann, E., Schmid, U. (eds.) Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP 2007), pp. 15–26 (2007)
[6] Olsson, R.J.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
[7] Katayama, S.: Systematic search for lambda expressions. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, vol. 6, pp. 111–126. Intellect (2007)
[8] Quinlan, J.R.: Learning first-order definitions of functions. Journal of Artificial Intelligence Research 5, 139–161 (1996)
[9] Muggleton, S., Feng, C.: Efficient induction of logic programs. In: Proceedings of the 1st Conference on Algorithmic Learning Theory, Ohmsma, Tokyo, Japan, pp. 368–381 (1990)
[10] Muggleton, S.: Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming 13(3-4), 245–286 (1995)
[11] Flener, P.: Inductive Logic Program Synthesis with Dialogs. In: Muggleton, S. (ed.) Proceedings of the 6th International Workshop on Inductive Logic Programming, Stockholm University, Royal Institute of Technology, pp. 28–51 (1996)
[12] Hernández-Orallo, J., Ramírez-Quintana, M.J.: Inverse narrowing for the induction of functional logic programs. In: Freire-Nistal, J.L., Falaschi, M., Ferro, M.V. (eds.) Joint Conference on Declarative Programming, pp. 379–392 (1998)
[13] Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education, New York (1997)
[14] Kitzelmann, E.: Analytical inductive functional programming. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 87–102. Springer, Heidelberg (2009)
[15] Sheard, T., Jones, S.P.: Template metaprogramming for Haskell. In: Chakravarty, M.M.T. (ed.) ACM SIGPLAN Haskell Workshop 2002, pp. 1–16. ACM Press, New York (2002)
[16] Wadler, P.: The essence of functional programming. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albequerque, New Mexico, pp. 1–14 (1992)
[17] King, D., Wadler, P.: Combining monads. Mathematical Structures in Computer Science, pp. 61–78 (1992)
[18] Spivey, J.M.: Combinators for breadth-first search. J. Funct. Program. 10(4), 397–408 (2000)
[19] Spivey, M., Seres, S.: The algebra of searching. In: Proceedings of a symposium in celebration of the work of. MacMillan, Basingstoke (2000)
[20] Katayama, S.: Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 199–210. Springer, Heidelberg (2008)