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 II 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 II, 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 I [Kitzelmann and Schmid 2006], and IGOR II [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 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 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 DILOG [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 $F(p_1, \ldots, p_n) \rightarrow t$ 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 (rhs) 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 patterns, 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 $\sigma$ 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

\[
\begin{align*}
\text{reverse}([]) &= [] \\
\text{reverse}([X]) &= [X] \\
\text{reverse}([X,Y]) &= [Y,X]
\end{align*}
\]

be some examples for the \textit{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 \( \text{cons} \) 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 \textit{reverse} examples as they have been put into one subset in the previous section. The initial rule for these two examples is:

\[
\text{reverse}([Q,Qs]) = [Q2,Qs2]
\]

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 example sets for two new help functions \( sub1 \) and \( sub2 \):

\[
\begin{align*}
sub1([X]) &= X \\
sub1([X,Y]) &= Y
\end{align*}
\]

\[
\begin{align*}
sub2([X]) &= [] \\
sub2([X,Y]) &= [X]
\end{align*}
\]

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:

\[
\text{reverse}([Q,Qs]) = sub1([Q,Qs] | sub2[Q,Qs])
\]

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:

\[
sub2([Q,Qs]) = Qs2
\]

The example outputs, \( [], [X] \) of \( sub2 \) match the first two example outputs of the \textit{reverse}-function. That is, the unfinished rhs \( Qs2 \) can be replaced by a (recursive) call to the \textit{reverse}-function. The argument of the call must map the inputs \( [X], [X,Y] \) of \( sub2 \) to the corresponding inputs \( [], [X] \) of \textit{reverse}, i.e., a new help function, \( sub3 \) is needed. This leads to the new example set:

\[
\begin{align*}
sub3([X]) &= [] \\
sub3([X,Y]) &= [X]
\end{align*}
\]

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:

\[
sub2([Q,Qs] = \text{reverse}(sub3([Q,Qs]))
\]

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\(^1\) \( \Sigma \), a set of variables \( X \), and a term rewriting system over \( \Sigma \) 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, 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 sub-sort 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 sub-sort 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

\[
\begin{align*}
\text{eq } & \text{rev} = \text{nil} . \\
\text{eq } & \text{rev}(\text{cons}(X, \text{nil})) = \text{cons}(X, \text{nil}) .
\end{align*}
\]

where X is a variable of sort Item. Applying upEqs(’M, false) then yields:

\[
\begin{align*}
\text{eq } & \text{rev} = \text{nil.List} [\text{none}] . \\
\text{eq } & \text{rev}(\text{cons}(X, \text{nil.List})) = \text{cons}(X, \text{nil.List}) [\text{none}] .
\end{align*}
\]

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,

\[
\text{metaMatch(upModule(’M, false), ’X:List, ’cons[’Y:Item, ’nil.List], nil, 0)}
\]

yields the term

\[
\text{’X:List } \leftarrow \text{’cons[’Y:Item, ’nil.List]}
\]

of sort Assignment which is a sub-sort 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) [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:

```haskell
data Exp
  = VarE Name
  | ConE Name
  | LitE Lit
  | AppE Exp Exp
```

Template Haskell’s dual quasi-quoting ([| |]) and splicing ($) operators would provide us with the means to transform code into such an algebraic data type and these expressions back into code, similar to MAUDE’s upXYZ functions. So [ | ] would be LitE (IntegerL 1) inside the TH’s Q monad and ($ (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.

```haskell
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 [])` would be represented as follows:

```haskell
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.

```haskell
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:

```haskell
ForallT ["a"] [AppT (ConT "Show")
    (VarT "a")]
```

For our convenience, we also create the class `Typed` to easily have access to a type of an expression or the like.

```haskell
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.

```haskell
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.

```haskell
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 Name to Type, storing the relevant key value pairs.

```haskell
data SynCtx = SCtx { sctx_types :: (Map Name Type),  -- function name -> its type, sctx_ctors :: (Map Name Type),  -- constructor name -> its type, sctx_classes :: (Map Name [Name]),  -- class name -> its superclasses, sctx_members :: (Map Name [Name]),  -- class name -> member functions names, sctx_instances :: (Map Name [Name]),  -- type -> classes, sctx_typesyns :: (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 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.

```haskell
type C a = StateT SynCtx (ErrorT String a)
(lookIn) :: (Ord a) => a -> (SynCtx -> Map a b) -> C b
(lookIn) n f = gets f >>= \m ->
  maybe (fail "Not in context!")
    (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 UT again as a monad transformer as follows.

```haskell
type Substitution t = [(Name, t)]
nullSubst = []
```

```haskell
type UT = StateT (Substitution t) C ()
```

Note that the last argument of StateT is the unit type. Consequently, a computation inside UT 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 UT monad with the empty substitution as initial state. As result the final state is returned.

```haskell
class (Term t) => Unifiable t where
  mgu :: t -> t -> C (Substitution t)
  mgu x y = execStateT (unify x y) nullSubst
  match :: t -> t -> UT
  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 UT 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 UT 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.

```haskell
type VarImg t = [(t, Name)]
nullImg = []
```

```haskell
type AU t = StateT (VarImg t) C t
```

However, unlike in the UT 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 \((\text{Term}\ t) \Rightarrow \text{Antiunifiable}\ t\) where

\[
\begin{aligned}
\text{aunify} & : [t] \rightarrow \text{AU} t \\
\text{antiunify} & : [t] \rightarrow \text{C} t \\
\text{antiunify} t & = \\
& \text{runStateT} (\text{aunify} t) \text{ nullImg}
\end{aligned}
\]

The types \(TExp\) and \(Type\) are now added as instances to these type classes. We omit the concrete implementations, since they are straightforward 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 \(\text{Rule}\), with a list of \(TExp\)s on the left-hand side (\(\text{lhs}\)) and one \(TExp\) on the right-hand side (\(\text{rhs}\)).

\[
\text{data \ Rule} = \text{R}\ { \begin{array}{l}
\text{lhs} :: [TExp] \\
\text{rhs} :: TExp
\end{array} }
\]

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 \(\text{Rule}\) in a covering rule \(\text{CovrRule}\).

\[
\text{data \ CovrRule} = \text{CR}\ { \begin{array}{l}
\text{name} :: \text{Name} \\
\text{covr} :: [\text{Int}] \\
\text{rule} :: \text{Rule}
\end{array} }
\]

The accessor functions \(\text{name}\), \(\text{rule}\), and \(\text{covr}\) return the name of the function, the rule itself, and the indices of the covered I/O examples. A \(\text{CovrRule}\) makes therefore only sense, when there is something the indices refer to. The data structure \(\text{IOData}\) 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 \(\text{IOData}\) just a synonym.

\[
\text{type \ IOData} = \text{M.Map} \text{ Name} [\text{Rule}]
\]

The indices in a \(\text{CovrRule}\) are just the position of rules in the list stored under a name. The indices should not be visible outside \(\text{IOData}\). For this purpose there are a couple of functions to create and modify \(\text{CovrRule}\) referring to a certain \(\text{IOData}\). We refrain from the concrete implementations here.

\[
\text{getNth} :: \text{Name} \rightarrow \text{IOData} \\
& \rightarrow \text{Maybe} [\text{CovrRule}]
\]

As the names suggest, \(\text{getNth}\) is simply a lookup and returns just a list of covering rules, each covering one I/O pair, and \(\text{getAll}\) just picks the \(n^{th}\) of all. The following functions are used to breakup and fuse covering rules. So \(\text{breakup}\) returns a list of covering rule, each covering one I/O pair of those covered by the original one, and \(\text{fuse}\) is the inverse of it, fusing many covering rules into one which covers all their I/O pairs.

\[
\begin{aligned}
\text{breakup} & :: \text{CovrRule} \rightarrow \text{IOData} \rightarrow [\text{CovrRule}] \\
\text{fuse} & :: [\text{CovrRule}] \rightarrow \text{C} \text{CovrRule}
\end{aligned}
\]

We have to be inside the \(\text{C monad}\) 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 \(\text{Decl}\), for each function one, and all calling dependencies between all functions to prevent the system to generate non-terminating programs.

\[
\begin{aligned}
\text{type \ Decl} & = (\text{Name},[\text{Rule}]) \\
\text{data \ Hypo} & = \text{HH}\ { \begin{array}{l}
\text{open} :: [\text{CovrRule}] \\
\text{clsd} :: [\text{Decl}] \\
\text{callings} :: \text{CallDep}
\end{array} }
\end{aligned}
\]

The basic idea behind calling dependencies is that if function \(f\) calls function \(g\), then \(f\) depends on \(g\) \((f \rightarrow g)\). The argument(s) of a call could either increase, decrease or remain in size, thus the dependency could be of either type \(\text{LT}\), \(\text{EQ}\), or \(\text{GT}\) \((\lessdot, \lessgtr, \lessless, \geq, \lessgtrdot, \leq, \lesslessdot)\).

Calling dependencies are transitive, so if \(f \rightarrow g\) and \(g \rightarrow h\) then also \(f \rightarrow h\). The kind of the transitive dependency has the maximal type of all compound dependencies with the obvious ordering \(\text{LT} < \text{EQ} < \text{GT}\).

If already a calling dependency \(f \rightarrow g\) exists, the following possibilities for \(g\) calling \(f\) are allowed:

\[
\begin{aligned}
\text{f} \lessdot \text{g} & \Rightarrow \text{g is not allowed to call f} \\
\text{f} \lessgtr \text{g} & \Rightarrow \text{g \lessdot f} \\
\text{f} \lesseqq \text{g} & \Rightarrow \text{g \lesseqq f} \text{ or g \lessgtrdot f} \\
\text{f} = \text{g} & \Rightarrow \text{f \lesseqq f}
\end{aligned}
\]

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.

\[
\begin{aligned}
\text{admissible} & :: (\text{Name},\text{Ordering},\text{Name}) \\
& \rightarrow \text{CallDep} \\
& \rightarrow \text{Bool} \\
\text{allowedCalls} & :: \text{Name} \\
& \rightarrow \text{CallDep} \\
& \rightarrow \text{M.Map} \text{ Name} [\text{Ordering}]
\end{aligned}
\]

The first one checks if the given (new) calling dependency is admissible, and the second returns for each function in a \(\text{CallDep}\) which additional calls to it are allowed. If a function is not mentioned in the \(\text{Map}\) returned by \(\text{allowedCalls}\), anything goes.

### 3.5 Comparing Rules and Hypotheses

To compare rules and hypotheses to decide which to process we establish the class \(\text{Rateable}\) with the member func-
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.

\[
\text{numberOfPartitions} :: \text{Hypo} \to \text{RatingData}
\]

\[
\text{numberOfPartitions} h = \text{liftM} \ \text{length} \ \$ \ \text{allRules} \ h
\]

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.

\[
\text{instance Rateable \ Hypo where}
\]

\[
\text{rate} \ h = \text{numberOfPartitions} \ h
\]

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 data structure representing this search space. In each step, for searching a space of hypotheses we need to maintain a data structure representing this search space. In each step, for searching a space of hypotheses we need to maintain a data structure representing this search space.

\[
\text{The Synthesis Monad}
\]

\[
\text{The main loop returns a list of equivalent programs inside I, w.r.t. the given heuristic, explaining the IO examples of the target function. Each program consists of a list of declarations Decl where each Decl defines one function by at least one Rule. First it fetches the currently best hypotheses, extracts the call dependencies and the unfinished rules from this hypothesis. If there are no open rules in all candidate hypotheses, the loop is exited and the candidate hypotheses are returned as result. Otherwise one rule is chosen for refinement, refined using the call dependencies and thus modifying the search space. After all, the loop is entered again.}
\]

\[
\text{class Rateable \ r \ where}
\]

\[
\text{rate} :: r \to \text{C \ Int}
\]

\[
\text{Hypo = \{ \text{ioData} :: IOData , \text{searchSpace} :: HSpace\}}
\]

\[
\text{type \ I \ a = StateT \ Igor \ C \ a}
\]

\[
\text{modifyIO} :: ( \text{IOData} \to \text{IOData} ) \to \text{IM} ()
\]

\[
\text{modifyHS} :: ( \text{HSpace} \to \text{HSpace} ) \to \text{IM} ()
\]

\[
\text{modifyIO} \ f = \text{modify} ( \text{\{igor@Igor \_ sp \_\} \to \igor\{searchSpace = f sp\}})
\]

\[
\text{The data structure Igor bundles the data structures IOData, known from section 3.4 to manage the various IO examples and HSpace, a priority queue on hypotheses w.r.t. to their heuristical rating. HSpace also supports efficient access to hypotheses by their rules to facilitate updating hypotheses after one refinement step. Igor serves as state for the monad I. The functions modifyHS and modifyIO allow us to modify HSpace and IOData inside I.}
\]

\[
\text{The main loop returns a list of equivalent programs inside I, w.r.t. the given heuristic, explaining the IO examples of the target function. Each program consists of a list of declarations Decl where each Decl defines one function by at least one Rule. First it fetches the currently best hypotheses, extracts the call dependencies and the unfinished rules from this hypothesis. If there are no open rules in all candidate hypotheses, the loop is exited and the candidate hypotheses are returned as result. Otherwise one rule is chosen for refinement, refined using the call dependencies and thus modifying the search space. After all, the loop is entered again.}
\]

\[
\text{type \ Prog = [Decl]}
\]

\[
\text{enterLoop = do}
\]

\[
\text{chss <- currentBestHypos}
\]

\[
\text{(deps,crs) <- chooseOneHypo \ chss}
\]

\[
\text{if (null crs)}
\]
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.

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 $\frac{n}{10}$ or more in most of the cases is more than expected, though. Table 2 shows all runtimes and the approximate ratio of old to new.

---

Table 1. Problem descriptions

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>is addition on Peano integers,</td>
</tr>
<tr>
<td>append</td>
<td>appends two lists,</td>
</tr>
<tr>
<td>drop</td>
<td>drops the first $n$ elements of a list,</td>
</tr>
<tr>
<td>evenpos</td>
<td>are all elements in a list which index is even,</td>
</tr>
<tr>
<td>init</td>
<td>are all elements but the last of a list,</td>
</tr>
<tr>
<td>last</td>
<td>is the last element in a list,</td>
</tr>
<tr>
<td>last</td>
<td>maps last over a list of lists,</td>
</tr>
<tr>
<td>length</td>
<td>is the length of a list as Peano integer,</td>
</tr>
<tr>
<td>odd/even</td>
<td>defines odd and even mutually recursive on Peano integers,</td>
</tr>
<tr>
<td>oddpos</td>
<td>are all elements in a list which index is odd,</td>
</tr>
<tr>
<td>repeatfst</td>
<td>overwrites all elements in a list with the first,</td>
</tr>
<tr>
<td>repeatlst</td>
<td>overwrites all elements in a list with the last,</td>
</tr>
<tr>
<td>reverse</td>
<td>reverses a list,</td>
</tr>
<tr>
<td>shiftl</td>
<td>shifts all elements in a list one position to the left,</td>
</tr>
<tr>
<td>shiftr</td>
<td>shifts all elements in a list one position to the right,</td>
</tr>
<tr>
<td>sort</td>
<td>sorts a list of Peano integers using ins as background knowledge which inserts into a sorted list,</td>
</tr>
<tr>
<td>swap</td>
<td>changes the position of two consecutive elements in a list element in a list,</td>
</tr>
<tr>
<td>switch</td>
<td>changes the position of the first and the last element,</td>
</tr>
<tr>
<td>take</td>
<td>takes the first $n$ elements from a list, and</td>
</tr>
<tr>
<td>weave</td>
<td>merges two lists into one by alternating their elements.</td>
</tr>
</tbody>
</table>

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.
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.

Acknowledgments

Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.

References


| Table 2. Runtimes on different problems in seconds |
|-----------------|-----------------|-----------------|
|                | IGOR 2          | IGOR 2H         | IGOR 2H/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          |
| repeatlst      | 0.052           | 0.004           | 1/13           |
| reverse        | 0.100           | 0.004           | 1/25           |
| shiftl         | 0.617           | 0.032           | 1/19           |
| shiftr         | 0.084           | 0.008           | 1/11           |
| sort           | 0.308           | 0.020           | 1/15           |
| swap           | 0.148           | 0.012           | 1/12           |
| switch         | 0.108           | 0.008           | 1/14           |
| take           | 2.536           | 0.036           | 1/70           |
| weave          | 1.380           | 0.012           | 1/115          |

* rounded to nearest proper fraction