Translating Curry To Haskell
System Demo

Bernd Braßel* and Frank Huch

University of Kiel, Institute of Computer Science
Olshausenstr. 40, 24098 Kiel, Germany
{bbr,fhu}@informatik.uni-kiel.de

Abstract

There exist several implementations of the functional logic language Curry: a transformation to Prolog and implementations of abstract machines for C and Java. We show that there are many advantages of a further implementation as a transformation to Haskell: increases in performance, availability of libraries and tools, and open access to the implementation. We present the basic ideas and a prototypical implementation of our transformation, which generates Haskell programs without use of impure features.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors (*Compilers*); D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; D.1.6 [*PROGRAMMING LANGUAGES*]: Logic Programming

General Terms Languages

Keywords Curry, Haskell, Compiler, Translation

1. Why compile to Haskell?

The language Curry is among the most widely used functional logic languages, integrating the key concepts of the two main paradigms of declarative programming: functional and logic programming. It supports lazy evaluation, logic search, constraint programming, and a modern polymorphic type system.

1.1 Previous Implementations

According to the nature of Curry, there are three basic ways to implement the language:

- implement an abstract machine in a suitable base language like C or Java
- transform Curry programs into logic programs
- transform Curry programs into (lazy) functional programs

Each way has some advantages and disadvantages: Designing an abstract machine has the advantage of giving the developer full access to all features, allowing him to gather information about sharing, or control the search mechanisms in order to implement for instance encapsulated search. However, both functional and logic programming come with a long history of optimization techniques, knowledge of how to avoid space leaks, how to design garbage collection and so on. When implementing a new machine from scratch, all of this work has to be reimplemented and chances are high that the machine will be behind state-of-the-art forever. In addition, libraries of the base language are comparatively hard to include in the implemented language. From the point of view of the abstract machine, these libraries are strictly external.

This is what makes approaches to transformation into related languages promising. All of the optimization techniques for the base language will be the more effective the lesser the level of interpretation is. There is no need to reimplement garbage collection or reconsider discussions about space leaks. The greater the similarities between base and implemented language, the easier is the integration of the base language's libraries. Furthermore, transformation to an existing language can involve less work than implementing an abstract machine from scratch. This is because many of the base language's features can be used without reconsidering implementation details of these features.

On the other hand, the developer of a transformation has less control on program execution. When transforming to a logic language, for instance, the developer has to rely on the base language features to control logic search. Implementing an own approach to encapsulated search, like the one proposed for Curry in [4], are hard or even impossible to realize. When transforming to a functional language (in the following we will consider the lazy functional language Haskell [8]), features like sharing are beyond access. This makes implementing Curry's features like call-time choice tricky to transform. Moreover, the developer has to comply with standards of the target language. For instance, in transforming to Prolog he has to consider implementing lazy evaluation in a strict language. When transforming to Haskell he has to obey the type system, if he wants to take advantage of all of Haskell's optimizations.

There have been several implementations of abstract machines for Curry in imperative languages. An early Java implementation [7] has by now been set aside, an implementation in C [11] has reached the state of usability. A second attempt to implement Curry in Java with new concepts is still under development [3]. All of these implementations more or less follow the idea of compiling into code for an abstract machine.

To the best of our knowledge, there has been only one transformation of Curry into a declarative language: the system PAKCS [6] transforms Curry programs to SICStus Prolog [13], thereby using Prolog's features of constraint solving, free variables and logic search. There have been no attempts so far to translate Curry to a functional language. One of the main reasons for this is the strong type systems of modern functional languages. Strong typed lan-

* The research described in this paper has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.

WCFLP'05 September 29, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-069-8/05/0009...$5.00.

guages are very good for programmers, but they can be a real obstacle in using them as a target language for program transformations. Recent generalizations of Haskell's type system make dynamic typing possible (classes `Dynamics` and `Typeable`, [9]). This allows the transformation of Curry into Haskell which can be compiled by the Glasgow Haskell Compiler (*ghc*) [5]. In the next subsection we will show that this opportunity is indeed very promising for the future of Curry.

1.2 What Haskell Could Bring

Although Prolog is also a declarative language, there are many differences between Prolog and Curry, more, as we will see, than there are between Curry and Haskell. Curry is not only syntactically very close to Haskell, many Curry modules in fact *are* Haskell programs. A great part of every-day Curry programming is functional programming, and what makes the basic concepts of Curry powerful is that each function can be used to perform logic search without changing its defi nition. Whether a deterministic function is used logically or functionally only depends on the way it is called: Calling a given function with free variables as arguments automatically induces a search if these variables are needed, whereas a call without free variables implies a deterministic evaluation like in Haskell.

This fact implies a great deal of potential optimization when translating Curry to Haskell: whenever we can make ensure by analyzing the source program that a given expression does not induce non-determinism or binding of free variables, we can simply use the original Curry code without any transformation at all. Clearly, this way we will automatically profi t from all of Haskell's optimization techniques. Accordingly, the amount of interpretation is, even in case of potentially non-deterministic programs, much lesser than in Prolog. This leads to other advantages like easy integration of Haskell-libraries, at least for deterministic parts of Curry programs.

One last point in favor of a Haskell transformation stems from recent research in encapsulated search: In [4] it was shown that new basic concepts are needed to provide a declarative access to search operators. Unfortunately, these concepts are not realizable if the features normally provided by Prolog are used. As logic search has to be added to the Haskell transformation, the developer has full control on this part of the implementation. Thus, a transformation to Haskell can provide a better, i.e. more declarative, way of implementing encapsulated logic search.

2. An Example Derivation

In this section, the kernel concepts of treating requests and managing the bindings of logical variables are exemplifi ed. Note however, that for the sake of readability, the derivation is only sketched. The real derivation is about twice as long (and twice as broad).

We consider the follwoing Curry program:

```
data Nat = O | S Nat

plus x O = x
plus x (S y) = S (plus x y)

isO O = True

main = isO (plus x x) where x free
```

Its evaluation with respect to Curry's semantics can be sketched as follows, where free variables are denoted by `X` and `Y`:

```
   main
→  isO (plus X X)
→  isO O | isO (S (plus Y (S Y)))
→  True  | ↛
```

The evaluation induces non-determinism, as `(plus X X)` demands instantiating the free variable `X` with `O` and `(S Y)`, because these are the patterns of `plus`. Only the fi rst of the two branches, separated by |, succeeds, the other fails. The important point is that `X` is immediately substituted by the binding, wherever it occurs. As free variables are represented by constructors, no purely functional implementation of such an immediate variable substitution is possible. Rather, we wrap the evaluation of the translated expression with a function `top`, which manages the bindings for free variables in a *store*. Whenever the evaluation of a function demands the value of a free variable, a request is generated. This request contains a) the free variable b) the patterns this variable should be bound to, if it is still free, and c) the function to be applied to the bound variable in order to continue the evaluation. The request eventually reaches `top`, which looks up the variable in the store. If there is no binding for the variable in the store, `top` induces a branch for every requested binding. This is yielding the constructor `Or` applied to a list representing the non-deterministic branches, cf. [4]. In these branches the evaluation continues with `top` applying the continuation to the binding of the variable.

The evaluation of our example can be sketched as:

```
main → top [] (isO (plus X X))
→ top [] (isO (Request [O,S Y] X (plus X)))
→ top [] (Request [O,S Y] X (isO . (plus X)))
→ Or [top [X↦O]   (isO . (plus X)) O,
      top [X↦S Y] (isO . (plus X)) (S Y)]
```

A nice feature of the translation to Haskell is that laziness enables us to choose which of the branches should be evaluated next. We decide to evaluate the fi rst branch only and continue with

```
  top [X↦O] (isO . (plus X)) O
→ top [X↦O] (isO (plus O X))
→ top [X↦O] (isO X)
→ top [X↦O] (Request [O] X (isO_1))
→ top [X↦O] (isO_1 O)
→ top [X↦O] True
→ True
```

3. Basic Concepts of the Transformation

As a well designed language, Curry has a certain number of kernel structures, with which every Curry program can be expressed. This kernel is called "FlatCurry" and forms the base of an operational semantics for Curry [1]. According to this conceptual design there are front ends, transforming Curry programs to Flat Curry programs. Therefore, all we have to provide is a transformation of FlatCurry to Haskell. The general syntax of FlatCurry programs is listed in Figure 1.

3.1 Transforming Data Declarations

First of all, we have to extend all data declarations by three new constructors enabling logic programming. In addition to the constructor terms which form the values in functional languages, a functional logic language knows a further kind of value: a free variable. Conceptually there are free variables of any type, representing all possible values of this type. In addition, failing computations in Curry to not directly correspond to run-time errors in Haskell. A fail is not the end of the whole program execution but rather only the end of the evaluation of a single branch. In order to represent free variables and failing evaluations correctly, we extend each data type with additional constructors `Fail` and `FreeVar`. For the purpose of controlling logical search, one further constructor is needed, which we call `Request`. The nature of requests will be explained below. In addition to the new constructors, we derive class instances for `Eq`, `Show` and `Typeable`, the fi rst of which correspond to general

P	$::=$	$D_1 \ldots D_m$	(program)
D	$::=$	$data\ t(\chi_1, \ldots, \chi_n) = C_1 \vert \ldots \vert C_m$	(type declaration)
	$\vert$	$f(x_1, \ldots, x_n) = e$	(function definition)
C	$::=$	$c(\tau_1, \ldots, \tau_n)$	(constructor definition)
τ	$::=$	χ	(type variable)
	$\vert$	$t(\chi_1, \ldots, \chi_n)$	(type)
e	$::=$	x	(variable)
	$\vert$	$c(e_1, \ldots, e_n)$	(constructor call)
	$\vert$	$f(e_1, \ldots, e_n)$	(function call)
	$\vert$	$let\ \{\overline{x_k = e_k}\}\ in\ e$	(let binding)
	$\vert$	$let\ x\ free\ in\ e$	(free variable)
	$\vert$	$e_1\ or\ e_2$	(disjunction)
	$\vert$	$case\ e\ of\ \{\overline{p_k \to e_k}\}$	(rigid case)
	$\vert$	$fcase\ e\ of\ \{\overline{p_k \to e_k}\}$	(flexible case)
p	$::=$	$c(x_1, \ldots, x_n)$	(pattern)

Domains			
$P_1, P_2, \ldots$	$\in$	$Prog$	(Programs)
$\chi, \chi_1, \chi_2, \ldots$	$\in$	Var	(Type Variables)
$t, t_1, t_2, \ldots$	$\in$	$\mathcal{T}$	(Types)
$\tau, \tau_1, \tau_2, \ldots$	$\in$	$\mathcal{T}Exp$	(Type Expressions)
$x, y, z, \ldots$	$\in$	Var	(Variables)
$a, b, c, \ldots$	$\in$	$\mathcal{C}$	(Constructors)
$f, g, h, \ldots$	$\in$	$\mathcal{F}$	(Functions)
$s, s_1, s_2, \ldots$	$\in$	$\mathcal{C} \cup \mathcal{F}$	
$p_1, p_2, \ldots$	$\in$	Pat	(Patterns)
$e, e_1, e_2, \ldots$	$\in$	Exp	(Expressions)

Figure 1. Syntax of FlatCurry

definitions in Curry,[1] the latter is needed to store bindings of free variables as described below.

For the example from Section 2 we obtain the following type:

```
data Nat = O
         | S Nat
         | NatFreeVar Int
         | NatFail
         | NatRequest (Request Nat)
   deriving (Eq, Show, Typeable)
```

The general scheme of transforming data declarations is easily derivable from the example.

The main advantage of extending *every* data type like this, is that purely functional programs can operate without any change. If their evaluation does not include non-determinism or binding of variables, it is guaranteed that there will be only constructors of the original data declaration.

In order to access the extended values in a uniform way, each datatype is an instance of the type class Curry, which features functions of the following types[2]:

```
class Typeable a => Curry a where
  failed :: a
  freeVar :: Int -> a
  request :: Request a -> a
  isVal :: a -> Bool
  isFail :: a -> Bool
  isFreeVar :: a -> Bool
  freeVarRef :: a -> Int
  selectReq :: a -> Request a
  subst :: Store -> a -> a
  ccase :: Curry b =>
             CaseMode a -> (a -> b) -> a -> b
  nf :: Curry b => (a -> b) -> a -> b
```

With the exception of ccase and nf, the instantiation for these functions is trivial and can easily be conceived from the example transformation of type Nat:

```
instance Curry Nat where
  failed = NatFail
  freeVar = NatFreeVar
  request = NatRequest
```

[1] It is possible to derive Eq and Show, because we provided a trivial instance for the data type Request. By the semantics of the transformed program, requests are never shown nor compared.

[2] The necessity for the class constraint Typeable and the purpose and definition of requests and the Store will be explained below.

```
  isVal O      = True
  isVal (S _) = True
  isVal _      = False
  isFail NatFail = True
  isFail _       = False
  isFreeVar (NatFreeVar _) = True
  isFreeVar _               = False
  freeVarRef (NatFreeVar r) = r
  selectReq (NatRequest r) = r
  subst store O = O
  subst store (S x) = S (subst store x)
  subst store (NatLogVar r) = fetch store r
```

The function ccase is one of our kernel concepts. It controls the evaluation of an expression (its third argument) according to the semantics of Curry. Its first argument is the CaseMode. There are three modes, corresponding to Curry's rigid and flexible case expressions and to the evaluation to head normal-form.

```
data CaseMode a = Rigid | Flexible [a] | HNF
```

Case expressions in FlatCurry are of the form given in Figure 1. The operational meaning of both case expressions is: Evaluate e to head normal-form, determine which pattern p_i matches this head normal-form and continue with the evaluation of the corresponding e_i. The difference between a rigid and a flexible case expression in Curry comes apparent, when e evaluates to a free variable. The evaluation of a rigid case suspends (residuation) whereas the evaluation of a flexible case induces a non-deterministic branching, binding the free variable to each of the patterns p_i. Therefore, the CaseMode Flexible has one argument which is the list of the bindings for a free variable.

The CaseMode HNF is used where an expression should be evaluated to head normal-form. In contrast to the modes Rigid and Flexible the evaluation of a ccase with mode HNF is finished when the result is a free variable.

The second argument of ccase is a continuation. Whenever the evaluation of the given expression (ccase's third argument) is finished, the continuation is applied to the result. Hence, the first rules of the transformation of our example type Nat are defined as follows:

```
ccase _ f O = f O
ccase _ f v@(S _) = f v
```

A failure has to be propagated to the top level:

```
ccase _ _ NatFail = failed
```

The interesting cases are those treating free variables and incoming requests. When ccase encounters a free variable, it sends a request to the top level of the evaluation. This is done, because the free variable might have been bound by the previous evaluation and all the bindings for free variables are managed at the top level. A request is either introduced by a case evaluation (`CaseReq`), containing a `CaseMode` and the reference of a free variable (an `Int`) or it is a request to create a new free variable (`NewVarReq`). Both kinds of requests contain a continuation `(b -> a)`:

```
data Request a =
    forall b. (Curry b, Typeable b) =>
      CaseReq (CaseMode b) FreeVarRef (b -> a)
  | forall b. (Curry b, Typeable b) =>
      NewVarReq (b -> a)
```

Here is our first use of Glasgow Haskell Compiler's extensions to the Haskell98 type system: ghc provides existential types in data declarations [10]. The data type `Request a` may contain `CaseModes` over arbitrary types `b` which by means of the continuation can be converted into a value of type `a`.

As explained above, when applied to a free variable, the function `ccase` has to request the current value of this variable from the top level, which manages the variable bindings:

```
ccase m f (NatFreeVar r) = request (CaseReq m r f)
```

As the request also contains the `CaseMode`, the top level function managing the variable bindings can react according to this mode, i.e. introduce a non-deterministic branching with different bindings for the variable or suspend the evaluation.

We have to guarantee that every request reaches the top level. Hence, each `ccase` forwards incoming requests. This is done by extending the continuation which is part of every request by the function forwarding the request:

```
ccase m f (NatRequest req) =
  request (extendCont req (ccase m f))
```

All definitions like `data Request` and `class Curry` are contained in a module `Curry.hs`. This module also contains the definition of `extendCont`:

```
extendCont (CaseReq m r f) f' =
            CaseReq m r (f' . f)
extendCont (NewVarReq f) f' = NewVarReq (f' . f)
```

This completes the definition of `ccase`. The last function to be explained is `nf`, which is responsible for evaluating a given expression to normal form. There are two notable differences between `nf` and `ccase`: 1) `nf` does not have a `CaseMode` as argument as its request always contains the mode `HNF` and 2) the treatment of complex data structures. Whenever a data declaration defines a complex constructor c with arity n, `nf` is responsible for evaluating the normal forms of the n arguments before reconstructing the term. The transformation scheme is therefore:

```
nf f (c x1 ... xn) =
  nf (\v1 ->...nf (\vn -> f (c v1...vn)) xn)...x1)
```

The whole definition of `nf` for our example type `Nat` is:

```
nf f O = f O
nf f (S x1) = nf (\v1 -> f (S v1)) x1
nf _ NatFail = failed
nf f (NatFreeVar r) = request (CaseReq HNF r f)
nf f (NatRequest req) =
  request (extendCont req (nf f))
```

This completes the transformation of data declarations. We can now examine how function declarations are translated to Haskell.

3.2 Transforming Function Declarations

We present our translation by means of the example from Section 2. Its FlatCurry representation is

```
plus x y = fcase x of {O-> y, S z-> S (plus x y)}
isO x = fcase x of {O -> True}
main = let x free in isO (plus x x)
```

Left-hand sides of functions, variables and function/constructor calls can be left unchanged.[3] An expression (`or` $e\ e'$) is translated as the construct with the semantically equivalent expression

`let x free in fcase x of {True ->`e`, False -> `e'`}`

The translation of expressions (`let x free in` e) is straight forward. Free variables are introduced by a call to function `free` applied to the continuation `(\x->e)`. The general scheme is therefore

$$trExp(\texttt{let x free in } e) = \texttt{free (\textbackslash x->}trExp(e)\texttt{)}$$

and `free` is defined as[4]:

```
free :: (Curry b, Curry a) => (b -> a) -> a
free f = request (NewVarReq f)
```

The main work for the transformation lies in translating case expressions. Similar to the scheme proposed in [2], we divide the tasks of evaluating the given expression to normal form and then choose the corresponding case branch into two different functions. Therefore, we introduce an auxiliary function for each case expression, performing the actual pattern matching. The evaluation to head normal form is done by a call to the function `ccase` as detailed above. The auxiliary function is then the continuation of `ccase`. For instance, the example function `isO` is translated to:

```
isO   x = fcase [O] isO_1 x
isO_1 O = True
isO_1 _ = failed
```

And generally `fcase` is defined as:

```
fcase ps = ccase (Flexible ps)
```

As explained in Section 3.1, the `CaseMode Flexible` contains a list of constructor terms which correspond to the patterns of the `fcase` expression. If the third argument of `ccase` is evaluated to a free variable, a non-deterministic branching is introduced. In each branch the variable is bound to one of the constructor terms in the list. If the constructor term corresponds to a complex pattern, i.e. headed by a constructor of arity greater than 1, the arguments of the constructor are fresh free variables. The example function `plus` is therefore translated to:

```
plus x y = fcase [O,S (free (\z->z))] (plus_1 x) y
plus_1 x O = x
plus_1 x (S z) = S (plus x z)
plus_1 _ _ = failed
```

Finally, the function `main` is treated specially. This is because ghc is a stand-alone compiler which expects `main` to be a constant of type `IO ()`, marking the start of the computation. The translated right hand side of main is wrapped by a function `start` and, if necessary by a call to `print`. For our example we obtain:

```
main = print (start (free (\x -> isO (plus x x))))
```

And `start` is generally defined as:

```
start = top emptyStore (nf id x)
```

The discussion of the functions `top` and `emptyStore` belongs to the next Section.

[3] Some functions and constructors have to be renamed to avoid name clashes with Haskell definitions.

[4] An alternative definition of `free` is discussed in Section 5.

4. Basic Concepts of the Run-Time System

Every Haskell program generated by our compiler imports the module `Curry.hs`. This module contains the basic definitions of our run-time system. We have already presented some of the definitions of `Curry.hs`, e.g. the type class `Curry`, the data declarations `CaseMode` and `Request` and the functions `extendCont`, `free` and `start`. Mentioned, but not defined were the functions `fetch`, `top` and `emptyStore`. The first topic of this section is the management of variable bindings.

4.1 Managing Free Variables: the Store

For implementing the store, we need a data structure which holds bindings for all free variables. Unfortunately, when complying to the Haskell 98 standard, it is not possible to define a data structure mapping variable indices (`Ints`) to values of arbitrary type. The solution is dynamic typing [9] provided in ghc [5] which allows the definition of an abstract data type `Store` with the following interface:

```
emptyStore :: Store
newStoreEntry :: Store -> Int -> (Store,[Int])
addToStore :: Typeable a=> Store-> Int-> a-> Store
fromStore :: Typeable a => Store -> Int -> Maybe a
```

The function `newStoreEntry` is used to introduce a number of new free variables, `addToStore` binds a free variable to a value and `fromStore` looks up the binding of a variable. The values read from the store are directly casted into the correct type, which is guaranteed to be equal for every occurrence of the same free variable by Curry's type system. For fast access, the store is efficiently implemented as a Braun tree [12].

Using this interface, we can now define the function `fetch`. `fetch` is called by all instances of `subst`, which is contained in the class `Curry` as defined above. The purpose of `subst` is to substitute all occurrences of free variables by their bindings after a normal from has been computed, cf. next section.

```
fetch :: (Curry a) => Store -> Int -> a
fetch store r = case fromStore store r of
                  Nothing -> freeVar r
                  Just t  -> subst store t
```

4.2 Representing Search

To represent search in Haskell, our compiler employs the concept proposed in [4]. There each non-deterministic computation yields a data structure representing the actual search space. The definition of this representation is independent of the search strategy employed and is captured by the following algebraic data type:

```
data SearchTree a =
        Fail | Value a | Or [SearchTree a]
```

Thus, a non-deterministic computation yields either the successful computation of a completely evaluated term v (i.e., a term without defined functions) represented by `Value` v, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or` $[t_1, \dots, t_n]$ where $t_1, \dots, t_n$ are search trees representing the subcomputations.

Analogously to `findall` in MCC, this structure is provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence, it is possible to define arbitrary search strategies on the structure of search trees. For instance, depth-first search can be defined as follows:

```
  depthFirst :: SearchTree a -> [a]
  depthFirst (Val v) = [v]
  depthFirst Fail    = []
  depthFirst (Or ts) = concatMap depthFirst ts
```

Evaluating the search tree lazily, this function evaluates the list of all values in a lazy manner too. With similar ease, breadth-first search can be defined [4].

4.3 The Kernel: Function `top`

It has been mentioned a few times that requests and variable bindings are managed at the top-level of the evaluation. We are now ready to give the according details, i.e. the definition of the function `top`. At the start of the computation, `top` is applied to an empty store and an expression of the form `nf id` e. It has to ensure that the final result is the normal form of e. First, function `top` determines whether the evaluation has finished or a request has to be treated. If the evaluation was successful, i.e. `nf id` e was reduced to a value, any remaining free variables have to be substituted by their bindings held in the store.

```
top store x
  | isFail x = Fail
  | isVal x || isFreeVar x = Value (subst store x)
  | otherwise = req store (selectReq x)
```

The function `req` is responsible for treating requests. As defined above, there are two kinds of requests: 1) `NewVarReqs` originate from the function `free` and call for the introduction of a fresh variable. These requests contain a continuation which is applied to the newly generated free variable. 2) `CaseReqs` originate from case (resp. hnf) expressions and call for looking up the binding of a given variable. Beside the reference of the variable to look up, these requests also contain a `CaseMode` and a continuation. If the referenced variable is still unbound, the subsequent action depends on the kind of the requesting case expression, i.e. the `CaseMode`, see below. If the variable is bound, the continuation is applied to the value found in the store.

```
req s (NewVarReq f) = top s' (f (freeVar ref))
    where (s',[ref]) = newStoreEntry s 1

req s (CaseReq cm ref f) = case fromStore s ref of
        Nothing -> treatCaseMode s cm ref f
        Just v  -> toplevel s (f v)
```

Finally, the function `treatCaseMode` is responsible for the case that the referenced variable is still free. For a flexible case, the variable has to be bound to the various values in the pattern list, thereby inducing a non-deterministic branching if more than one pattern is given. For a rigid case, the computation suspends, and for `HNF` the variable stays free. `treatCaseMode` is defined locally.

```
  where
   treatCaseMode Rigid = suspend
   treatCaseMode (Flexible [v]) = newTop v
   treatCaseMode (Flexible vs) = Or (map newTop vs)
   treatCaseMode HNF = toplevel s (f (freeVar ref))

   newTop v = top (addToStore s ref v) (f v)
```

This concludes the discussion of the run-time system.

5. Call-Time vs. Run-Time Choice

All the concepts discussed so far result in referentially transparent Haskell programs. There is however one feature of Curry, which is not efficiently translatable in this way: call-time choice.

A simple standard example shows the difference between run-time (RTC) and call-time choice (CTC):

```
  coin = or 0 1
```

```
main1 = coin+coin
main2 = let x=coin in x+x
```

In RTC, both versions of `main` reduce to the same value, namely `Or [Or [0,1],Or [1,2]]` whereas in CTC, only `main1` produces this value, whereas `main2` reduces to `Or [0,2]`.

The example reveals a difference in the notion of referential transparency between Curry and Haskell. In functional programming, it is part of referential transparency that there may be no difference between evaluating a given expression once or twice. The example shows, however, that this is not the case in Curry. In order to achieve CTC results via a referential transparent Haskell program, `coin` has to be extended by an argument and needs to be applied to the same value in the case of `main2` and two different values for `main1`. Obviously, this approach destroys the sharing explicitly introduced in `main2` and because it might be necessary also to extend the functions calling `coin`, the approach might destroy sharing in many places. As we fear that such a transformation might be disastrous for the overall effi ciency of the translated programs, we have chosen a different solution.

Because `or` expressions are translated by a flexible case, cf. Section 3.2, there is only a single point where we have to change the translation in order to achieve CTC: the declaration of free variables by the function `free`. If `free` is defi ned as

```
free f = f (freeVar (incGlobalCounter ()))
```

where `incGlobalCounter` issues the side effect of adding 1 to a global integer value, evaluating `(free,free)` results in `(FreeVar 1,FreeVar 2)` whereas `let x=free in (x,x)` results in `(FreeVar 1,FreeVar 1)`, if the counter was 0 before evaluation. This solves the problem of introducing CTC for the price of introducing an impure feature. The main drawback of using impure features is that we cannot rely on ghc's compiler optimizations since the correctness of the generated programs is not guaranteed in general.

6. Runtimes

At the moment the compiler is still under development and we have multiple ideas for runtime-improvements. However, we can already present some fi rst measurements in comparison to PAKCS[6]. We compared the following fi ve programs: `logLast` computes the last element of a list by means of `(++)` and strict equality. `naive reverse` reverses a list by appending each element at the end of the list which needs quadratic time. `fib 30` is defi ned by the mathematical exponential defi nition. `nondetShare` applies `logLast` to a list containig two shared calls of `fib 30`. `permsort` uses a non-deterministic version of `perm` to sort the reverse of a sorted list of length nine. The results are quite encouraging as the following table shows[5]:

	PAKCS	CTC	opt.RTC	RTC
`logLast`	2.5	13.8 [0.18]	8.4 [0.30]	18.5
`naive reverse`	4.9	2.7 [1.8]	1.5 [3.2]	2.7
`fib 30`	15.0	4.0 [3.8]	3.5 [4.3]	4.0
`nondetShare`	30.0	4.2 [7.1]	3.5 [8.6]	4.2
`permsort`	10.9	11.7 [0.93]	4.1 [2.66]	9.2

We compare runtimes (in seconds) of PAKCS, our compiler with call time choice (CTC) and run time choice (RTC). For RTC we can use ghc's optimizations and obtain the values in column opt.RTC. The values in square brackets represent the gained speedups compared to PAKCS.

In most cases our compiler produces more effi cient code. Only in computations in which a lot of free variables are guessed and most of the computation consists of search (like `logLast`) our compiler produces slower code than PAKCS. However, in practice large parts of a program are purely functional and this slowdown will be accumulated by the speed up in the functional computations. This is already shown by the benchmark `premsort`. Although this test performs extensive search with binding many free variables, our compiler can keep up with PAKCS.

The benchmark `nondetShare` demonstrates another advantage of our compiler compared to PAKCS. The computation of `fib 30` is shared, although it occures in two different non-deterministic computations. Here PAKCS does not provide sharing, which results in exactly the double execution time compared to `fib 30`.

[5] In contrast to SICStus Prolog, ghc does dynamically extend its used memory. Hence, we started our executable with 100MB of memory.

7. Conclusion

We have presented the advantages that a translation of Curry to Haskell implies and sketched its implementation. Our prototypical implementation demonstrates encouraging performance, even in comparison to a widely used implementations of Curry. In contrast to PAKCS, our compiler is able to translate Curry modules separately which make the development of larger applications more convenient.

For future work we want to profi t from Curry's similarity to Haskell. For purely functional programs, almost no transformation should be necessary at all which should result in much faster code. Furthermore, the implementation of concurrency and encapsulated search have to be completed.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[2] S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[3] S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.

[4] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[5] The Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.

[6] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. `http://www.informatik.uni-kiel.de/~pakcs/`, 2004.

[7] M. Hanus and R. Sadre. A concurrent implementation of Curry in Java. In *Proc. ILPS'97*, Port Jefferson (New York), 1997.

[8] Simon Peyton Jones et al. Haskell 98 report. Technical report, `http://www.haskell.org`, 1998.

[9] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.

[10] K. Läufer. Type classes with existential types. *J. of Functional Programming*, 6(3):485–517, May 1996.

[11] Wolfgang Lux and Herbert Kuchen. An effi cient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 – 29. Jahrestagung der Gesellschaft für Informatik, Oktober 1999*, pages 390–399. Springer Verlag, 1999.

[12] Chris Okasaki. Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6):661–666, November 1997. Functional Pearl.

[13] SICStus Prolog. `www.sics.se/isl/sicstuswww/site/index.html`.