The Curry interpreter Harry

Axel Rauschmayer

July 5, 1999

Abstract

This is a documentation of the main ideas behind *Harry*, an interpreter for the functional logic language *Curry*. Harry is implemented in the functional language *Haskell*.

Contents

1 Introduction

The goal of project “Harry” was to write an interpreter that supports the main features of the functional-logic programming language *Curry*.
As Curry was designed to be a superset of the functional programming language *Haskell*, this language was chosen for the implementation.
To keep the project managable, only the most important aspects of Curry have been implemented. These include:

- Residuation
- Needed narrowing
- A complete type system that includes lists

Not implemented were:

- Other basic types than Int
- Encapsulated search

- Committed choice
- Input/Output

Unfortunately, due to time and space constraints, this documentation requires a lot of prior knowledge. The following is a list of topics one should be familiar with before reading on (in decreasing importance):

- The Curry report [HAK+99] (especially where it's introducing the language and explaining definitional trees).
- Implementation of functional languages
- Implementation of logic languages

2 Core syntax

Before interpretation, the subset of Curry that Harry supports, is transformed into a variant of the language, called *Core*. Core makes a lot of things, that are implicitly part of Curry, explicit (like suspension etc.). It also contains a few extensions to represent *definitional trees* (see [HAK+99, section D.1]) within the language. Here is the abstract syntax of Core in Haskell notation:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Function} & = & \mathit{Function}\ \mathit{Name}\ [\mathit{Name}]\ \mathit{Expr} \\
\\
\textbf{data}\ \mathit{Expr} & = & \mathit{Lambda}\ [\mathit{Name}]\ \mathit{Expr} \\
 & | & \mathit{FreeDecl}\ \mathit{Name}\ \mathit{Expr} \\
 & | & \mathit{Let}\ \mathit{LocalDef}\ \mathit{Expr} \\
 & | & \mathit{Letrec}\ [\mathit{LocalDef}]\ \mathit{Expr} \\
 & | & \mathit{Appl}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Var}\ \mathit{Name} \\
 & | & \mathit{Constructor}\ \mathit{ConstrTag}\ [\mathit{Expr}] \\
 & | & \mathit{Literal}\ \mathit{Lit} \\
 & | & \mathit{Builtin}\ \mathit{BuiltinTag}\ \mathit{Arity} \\
 & | & \mathit{Case}\ \mathit{Expr}\ [\mathit{Clause}] \\
\\
\textbf{data}\ \mathit{LocalDef} & = & \mathit{VarBinding}\ \mathit{Name}\ \mathit{Expr} \\
 & | & \mathit{FuncBinding}\ \mathit{Function} \\
\\
\textbf{data}\ \mathit{BuiltinTag} & = & \mathit{BuiltinTagGuard} \\
 & | & \mathit{BuiltinTagUnify}
\end{array}
$$

$$
\begin{array}{lcl}
 & | & \mathit{BuiltinTagOr} \\
 & | & \mathit{BuiltinTagFail} \\
 & | & \mathit{BuiltinTagIfFree} \\
 & | & \mathit{BuiltinTagSuspendOn} \\
 & | & \mathit{BuiltinTagAnd} \\
 & & \\
\textbf{data}\ \mathit{Clause} & = & \mathit{Clause}\ \mathit{ConstrTag}\ [\mathit{Name}]\ \mathit{Expr}
\end{array}
$$

The terminal symbols are:

- Name: An identifier, usually the name of a variable or a function.
- Lit: In Harry, only integer literals are possible.
- ConstrTag: The tag of a constructor.

The following is a presentation of the features of and the transformations into Core syntax. The transformations are only informally shown (by example). Generally, they have to be recursively applied.

2.1 Builtins

Builtin (i. e. internally implemented) functions perform a variety of tasks. The reason to use so many builtins was that Harry does a lot of search-and-replace within expressions in Core syntax. Using builtins instead of special constructs reduces the amount of replacement cases needed (as there is only one case for all builtins).

- `guard` *contstraint expr*: If *constraint* is fulfilled, the expression is returned, otherwise evaluation fails.
- `unify` $expr_1$ $expr_2$: Unifies the two expressions.
- `or` $expr_1$ $expr_2$: Returns the answers of both expressions (same effect as an *or* node in a definitional tree).
- `fail`: Fails explicitly (i. e. backtracks).
- `ifFree` *name* $expr_1$ $expr_2$: If the variable *name* is free, $expr_1$ is returned, otherwise $expr_2$.
- `suspendOn` *name expr*: If the variable *name* is free, evaluation suspends. As soon as it is bound to a value, *expr* is returned.
- `and` $constraint_1$ $constraint_2$: Concurrently solves the constraints (same effect as a conjuction between constraints, see [HAK$^+$99, section 2.5]).

2.2 Case

The case statement is introduced in four steps:

1. Introduce a theoretical annotated case statement.
2. Show how definitional trees and the annotated case relate.
3. Introduce the core case statement.
4. Show how the annotated case statement can be transformed into Core syntax.

2.2.1 Annotated case statement

The syntax of this statement roughly corresponds to the case construct in Haskell (defined in EBNF):

annotCase	::=	*caseRigid* | *caseFlex*
caseRigid	::=	`caseR` *var* `of` {*branch*}+
caseFlex	::=	`caseF` *var* `of` {*branch*}+
branch	::=	*constructor* {*var*} `->` *expression*

Note:

- No guards: Guards can be expressed with the guard builtin.
- Implicit evaluation annotation: The last character of the statement name indicates if it is to be interpreted rigidly or flexibly.
- Only basic patterns: More complicated patterns can be expressed with nested annotated cases.

2.2.2 Definitional trees and the annotated case statement

The annotated case is used for representing the branch nodes of definitional trees within the language.
As an aside, note that *or* nodes are represented as *or* builtins and rule nodes (or rather, their right sides) as normal Core expressions.
To give an example, take the following program from [HAK+99, section D.1].

$$
\begin{array}{l}
\textbf{data}\ Nat\ =\ Z \mid S\ Nat \\
leq\ ::\ Nat\ \rightarrow\ Nat\ \rightarrow\ Bool \\
leq\ Z\ \ \ \ \ \ \ n\ \ \ \ \ \ = True \\
leq\ (S\ m)\ Z\ \ \ \ \ = True \\
leq\ (S\ m)\ (S\ n) = leq\ m\ n
\end{array}
$$

Its definitional tree is

$$
\begin{array}{l}
\textbf{branch}(\text{leq } x_1\ x_2,\ 1,\ \text{flex}, \\
\qquad\qquad \textbf{rule}(\text{leq Z } x_2 = \text{True}), \\
\qquad\qquad \textbf{branch}(\text{leq (S } m)\ x_2,\ 2,\ \text{flex}, \\
\qquad\qquad\qquad\qquad \textbf{rule}(\text{leq (S } m) \text{ Z} \quad = \text{False}), \\
\qquad\qquad\qquad\qquad \textbf{rule}(\text{leq (S } m) \text{ (S } n) = \text{leq } m\ n)\ )\ )
\end{array}
$$

Because the branches are annotated as *flex*, two nested caseF statements are used to represent this definitional tree:

$$
\begin{array}{l}
\mathit{leq}\ x_1\ x_2 \ \rightarrow\ \textbf{caseF}\ x_1\ \textbf{of} \\
\qquad\qquad\quad Z \quad\ \rightarrow\ \mathit{True} \\
\qquad\qquad\quad S\ x_3 \ \rightarrow\ \textbf{caseF}\ x_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad Z \quad\ \rightarrow\ \mathit{False} \\
\qquad\qquad\qquad\qquad\quad S\ x_4 \ \rightarrow\ \mathit{leq}\ x_3\ x_4
\end{array}
$$

2.2.3 Core case statement

coreCase ::= `case` *var* `of` {*branch*}+

The Core case statement is a simplified version of the annotated one: The case variable has to be bound (which is even stricter than rigid).

2.2.4 Transformation of the annotated case statement into Core syntax

Transforming caseR:

$$
\begin{array}{l}
\textbf{caseR}\ v\ \textbf{of} \\
\quad A\ x\ y \ \rightarrow 11 \\
\quad B\ z \quad \rightarrow 22
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\textbf{suspendOn}\ v \\
\qquad\qquad \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\quad A\ x\ y \ \rightarrow 11 \\
\qquad\qquad\quad B\ z \quad \rightarrow 22
\end{array}
$$

Transforming caseF:

$$
\begin{array}{l}
\textbf{caseF}\ v\ \textbf{of} \\
\quad A\ x\ y \ \rightarrow 11 \\
\quad B\ z \quad \rightarrow 22
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\textbf{ifFree}\ v\ (\textbf{or}\ (\textbf{guard}\ (v\ =:=\ A\ x\ y)\ 11) \\
\qquad\qquad\qquad (\textbf{guard}\ (v\ =:=\ B\ z)\quad 22)) \\
\qquad\quad \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad A\ x\ y \ \rightarrow 11 \\
\qquad\qquad B\ z \quad \rightarrow 22
\end{array}
$$

2.3 Guards, where and constraints

Other Curry constructs are eliminated by transformation to a simpler Core statement, as well.

2.3.1 Guards and where

Non-boolean guards and where are transformed as follows

$$
\begin{array}{l}
f\ x \mid (x\ =:=\ A)\ =\ C\ y\ z \\
\quad\ \ \mid (x\ =:=\ B)\ =\ C\ z\ y \\
\quad \textbf{where} \\
\qquad y\ =\ 33 \\
\qquad z\ =\ 44
\end{array}
$$

becomes

$$
\begin{array}{l}
f\ x\ =\ \textbf{let} \\
\qquad\qquad y\ =\ 33 \\
\qquad\qquad z\ =\ 44 \\
\qquad\quad \textbf{in} \\
\qquad\quad \textbf{guard}\ (x\ =:=\ A)\ (C\ y\ z)\ \textbf{or}\ \textbf{guard}\ (x\ =:=\ B)\ (C\ z\ y)
\end{array}
$$

2.3.2 Constraints

$$(x\ =:=\ f\ y)\ \&\ (y\ =:=\ 3) \implies (\textbf{unify}\ x\ (f\ y))\ \textbf{and}\ (\textbf{unify}\ y\ 3)$$

3 Evaluation using narrowing

We explain Harry's evaluation strategy in two steps:

1. First, only the implementation of narrowing is explained.
2. Then we explain what is needed in addition to also implement residuation and parallelism.

We continue with explaining the implementation of narrowing and free variables.

3.1 Operational semantics for narrowing

Choosing a more informal approach than [HAK+99], a few example programs and queries are presented to show what the operational semantics of Curry mean for an implementation.

Independence of or branches

Once a free variable has been bound, its value remains constant in the following *or* branch. Additonally, the binding is local to this branch.

$$\begin{aligned} f\ 1 &= 22 \\ f\ x &= x \end{aligned}$$

Query	Answer
$f\ y$	$\{y = 1\}\ 22 \mid \{y = \mathit{free}\}\ \mathit{free}$

Sharing of variables

Using the same free variable twice in an expression means that the same value is being referred to.

$$\begin{aligned} f\ 1\ 1 &= 11 \\ f\ 2\ 2 &= 22 \\ f\ 1\ 2 &= 33 \end{aligned}$$

Query	Answer
$f\ x\ y$	$\{x = 1,\ y = 1\}\ 11 \mid \{x = 2,\ y = 2\}\ 22 \mid \{x = 1,\ y = 2\}\ 33$
$f\ x\ x$	$\{x = 1\}\ 11 \mid \{x = 2\}\ 22$

Binding to sets of answers

If two independent variables are each bound to a set of values, the cartesian product of the sets is used for computing the answers.

$$\begin{aligned} &\textbf{data}\ C = C\ \mathit{Int}\ \mathit{Int} \\ &f\ x\ y = C\ x\ y \\ &g\ x = C\ x\ x \\ &h = 1 \\ &h = 2 \end{aligned}$$

Query	Answer
$f\ h\ h$	$\{\}\ C\ 1\ 1 \mid \{\}\ C\ 2\ 2 \mid \{\}\ C\ 1\ 2 \mid \{\}\ C\ 2\ 1$
$g\ h$	$\{\}\ C\ 1\ 1 \mid \{\}\ C\ 2\ 2$

3.2 Abstract machine for narrowing

To correctly evaluate programs, Harry uses an abstract machine that comprises the following global structures (figure 1):

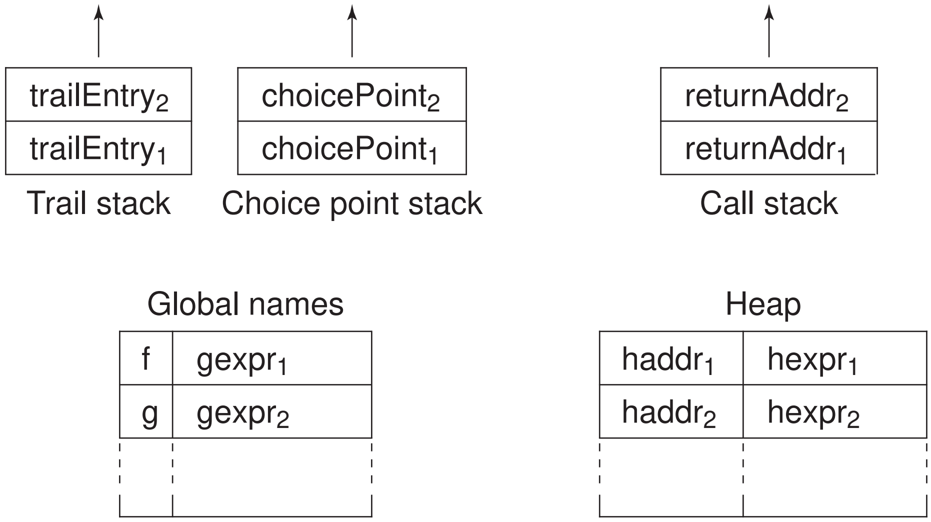

Figure 1: Structures of the abstract machine

- Heap: Maps *heap pointers* to expressions
- Call stack: Stores the return addresses for each evaluation step (continuations are used as pseudo-addresses)
- Choice point stack: To keep track of choice points (explained below)
- Trail stack: Records actions that have to be undone when backtracking

The following structures provide additional support for evaluation:

- Global names

3.3 Interpretation

Harry always interprets the abstract syntax of a program. Applications are represented as trees (like graph-based implementations of functional languages do, see figure 2).

3.3.1 Resolving expressions

The first concept one needs to understand is the *resolution of expressions*. Basically, it is the process of replacing the variable names with the structure they refer to at runtime (or formal parameters with actual parameters).

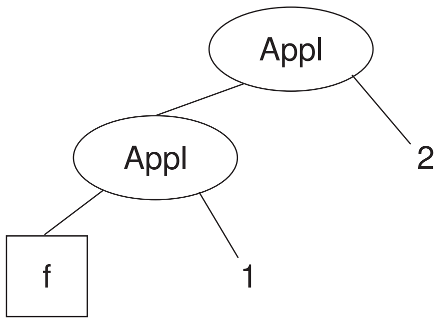

Figure 2: (f 1 2) as an application tree

- A function name is replaced by a lambda expression (which is its value!).
- Within an expression, equal names of variables are replaced with the same heap pointer to their *shared* value.
- If a variable is still free, the pointer refers to a specially tagged *free heap cell*. Free heap cells are replaced by values as soon as the variable is bound.

3.3.2 Evaluating expressions

Evaluation happens within the recursive *evalExpr* function. To interpret an expression, evalExpr looks for the redex (the leftmost unreduced expression in the application tree) and pushes parameters on a local *argument stack* as it goes downward. The initial call is *evalExpr expression* []

```
evalExpr expr args =
 case expr of
   CurExprAppl func arg          → evalExpr func (arg : args)
   CurExprLiteral _              → retSub expr
   CurExprLambda params body  → do
            heapArgs ← ⟨allocate the args on the heap⟩
            dict        ← ⟨map params to heapArgs⟩
            resBody   ← resolveExpr body dict
            goSub      (evalExpr resBody [])
```

Each usage of goSub pushes a return address onto the call stack (which is managed by the monad used for evaluation, EvalMonad) before jumping to its argument. The inverse, retSub, pops an address from the call stack and returns to it with a result value.

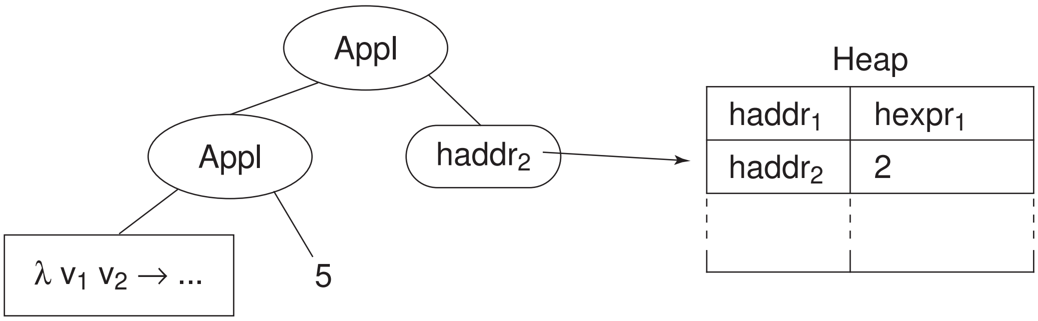

Figure 3: Resolved (g 5 y)

Example function call

To interpret the query f 1 2 (figure 2), do the following:

1. Resolve the expression, which in this case is easy: Only f needs to be replaced by its definition, the lambda expression $\lambda\, x\ y\ \rightarrow\ g\ 5\ y$ (*).

2. Look for the redex. On the way down, first 2, then 1, get pushed onto the argument stack. The redex in this case is (*).

3. Resolve the body of the lambda expression (*): Allocate the current parameter values (from the argument stack) on the heap. Then replace each occurence of x and y in the body with a corresponding heap pointer (figure 3).

4. Evaluation continues with the resolved body: Recursively sub-call evalExpr to evaluate g 5 2. This is recorded on the call stack.

3.3.3 Backtracking

Lending ideas from Prolog, backtracking and choice points are used to implement nondeterminism (i. e. evaluation of *or* branches).
A choice point contains:

- A list of the choices available (again as pseudo-addresses, i. e. continuations)
- The top of the trail stack
- The whole call stack

While evaluating, all changes to the heap (and thus to free variables) are recorded on the trail stack. To evaluate several *or* branches, a new choice point is created that contains the starting address of each branch (among other things).
Then evaluation goes on with the first branch. After that branch fails, all changes (i. e. variable bindings) the branch has made are untrailed and the saved call stack is restored. Restoring the call stack means that one will redo all the evaluation steps of the first branch–but this time with new bindings. This is because to finish an evaluation step, one returns a result to an address on the call stack.
Everything else is similar to Prolog: If one tries to backtrack and no more choices are available in the current choice point, it is popped from the stack and backtracking goes on analogously with the next choice point. If the choice point stack is empty, a failure occurs.
Contrary to Prolog, though, Curry always returns all possible answers for a query. Therefore, Harry uses backtracking to get a set of answers.

4 Evaluation using residuation

Up to now all evaluation has been done in a single task. To get residuation and concurrent evaluation, we need to implement parallelism.

4.1 Concurrent evaluation

4.1.1 Tasks

data *Task* = *Task TaskId CallStack*

Parallelism is handled by executing each concurrent evaluation in a separate task. It is implemented using coroutining, i. e. a task is only interrupted when it explicitly yields execution. A task consists of a unique ID and a call stack.
Coroutining makes it easy to sequentially record the changes to a task state which is essential for untrailing (and thus for backtracking, too).

Task states

Tasks can be in one of the following states (see figure 4):

- New: A task is *new* before it is activated for the first time.

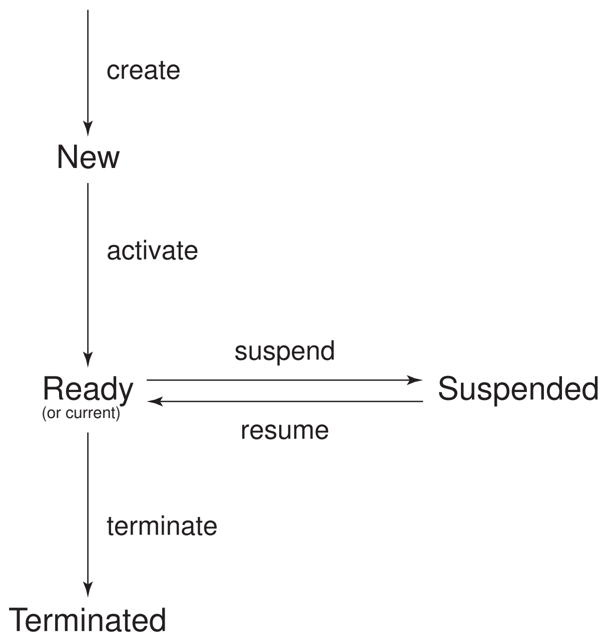

Figure 4: Task states

- Current: The task that is presently being executed is called the *current* task.
- Ready: Any one of the *ready* tasks is eligible for execution once the current task yields.
- Suspended.
- Terminated: A task that successfully finishes is called *terminated*.

A queue for each state holds the corresponding tasks. The exceptions are the current task which is held separately from the queues and terminated tasks (which are just being disposed of).

Scheduling operations

The following scheduling operations handle coroutining.

- createTask: Creates a task and adds it to the new queue.
- terminateCurTask: Terminates the current task.

- suspendCurTaskOnVar: Suspends the current task (moves it to the suspended queue) due to a free variable. Its ID is added to a list that is stored with that variable (the *suspension list*).

- bindFreeVar: Resumes the tasks in the suspension list of a free variable that is to be bound to a value. Each task is moved from the suspended to the ready queue.

- yield: The current task yields execution to another task (if no ready or new tasks are available, execution supends globally). This internal operation is executed by terminateCurTask and suspendCurTaskOnVar. After all other scheduling operations, the current task is unchanged.

4.1.2 Evaluating a conjunction of constraints

Concurrency can only occur while evaluating constraints. If a conjuction of one or more constraints is to be evaluated (concurrently!), the parent task calling for the evaluation is put to sleep and a new child task for each constraint is created. After all children have terminated, the parent task can go on evaluating.
The tasks don't need to explicitly return values, as a constraint either fails or is solved. A solved constraint results in successful termination, failure in backtracking (which may even go back before the creation of the child tasks and lead to a failure of the parent).
Note that things are different when concurrently evaluating two expressions for unification. There, a dictionary mapping TaskIds to expressions (the return values of the child tasks) is stored in the *and* node. But apart from that, the basic ideas remain the same.

$$\textbf{data}\ \mathit{AndNode}\ =\ \mathit{AndNode}\ \mathit{AndNodeId}\ \mathit{TaskId}\ \mathit{Int}$$

An *and* node does the necessary bookkeeping. It consists of its own unique ID, the ID of the parent task and a *suspension count*. *And* nodes are kept in a global *and node dictionary* mapping IDs to nodes.
The suspension count is initially set to the number of children and decremented each time a child terminates. As the parent is suspended upon the activation of the *and* node, its execution can continue as soon as the suspension count is 0.
The parent executes the following steps to create an *and* node (figure 5):

- Create the *and* node.

- Create the children.

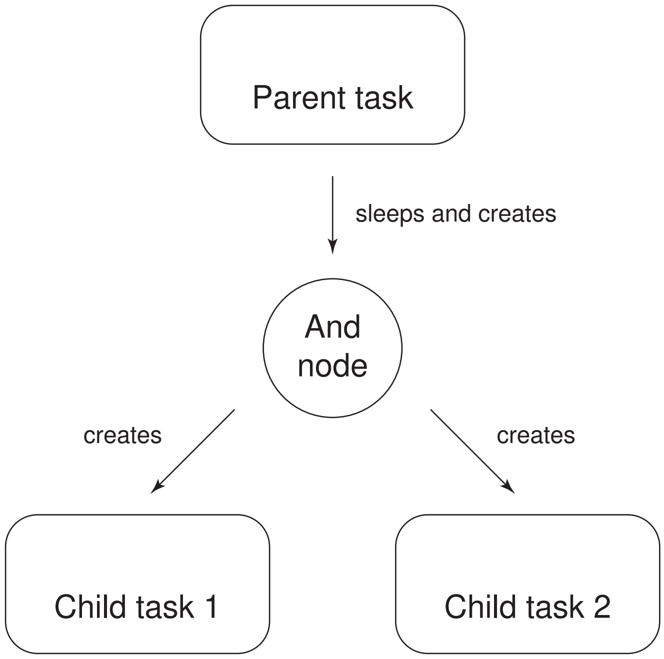

Figure 5: Creation of the child tasks

- Suspend yourself.

Each child contains the following execution steps:

- Sub-call evalExpr to evaluate constraint.
- Decrement suspension count (if it is 0 now, resume parent).
- Terminate yourself.

4.2 Concurrent backtracking

Backtracking has the same purpose for concurrent evaluation as for single-task evaluation: To undo the changes that have been effected so far and to redo a calculation with just one choice made differently. As execution this time spans several tasks, all changes in the life cycle of a task have to be closely recorded so that the mentioned undoing and redoing is possible.

4.2.1 Choice points

data *ChoicePoint* = *ChoicePoint* {

$$
\begin{array}{lll}
\quad \mathit{ownerCP} & :: & \mathit{TaskId}, \\
\quad \mathit{callStackCP} & :: & \mathit{CallStack}, \\
\quad \mathit{choicesCP} & :: & [\mathit{EvalMonad\ Expression}], \\
\quad \mathit{trailPtrCP} & :: & \mathit{PointerS} \\
\}
\end{array}
$$

Choice points are stored in a stack that is global to all tasks (figure 6). Each choice point now consists of

- ID of the task it applies to
- saved call stack of this task
- continuations for the choices
- trail pointer used like in single-task execution

4.2.2 Trailing

Corresponding to the choice point stack, the trail is global to all tasks, too. A lot more information is trailed now:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{TrailEntry} & = & \mathit{TrailHeap\ PointerH\ Expression} \\
 & \mid & \mathit{TrailTaskCreation\ TaskId} \\
 & \mid & \mathit{TrailTaskActivation\ TaskId\ TaskQueueEntry} \\
 & \mid & \mathit{TrailAndAllocation\ AndNodeId} \\
 & \mid & \mathit{TrailAndDisposal\ AndNode} \\
 & \mid & \mathit{TrailSuspensionCount\ AndNodeId\ Int} \\
 & \mid & \mathit{TrailSuspension\ TaskId} \\
 & \mid & \mathit{TrailResumption\ TaskId\ TaskQueueEntry} \\
 & \mid & \mathit{TrailSuspListAddition\ PointerH\ TaskId} \\
 & \mid & \mathit{TrailBindFreeVar\ PointerH}\ [\mathit{TaskId}] \\
 & \mid & \mathit{TrailTaskTermination\ TaskId}
\end{array}
$$

To untrail a(n)

- heap change: Restore the old value.
- task creation: Remove the task from the new queue.
- (first) task activation: Put the task from the ready to the new queue.
- *and* node allocation: Remove the *and* node from the *and* node dictionary.

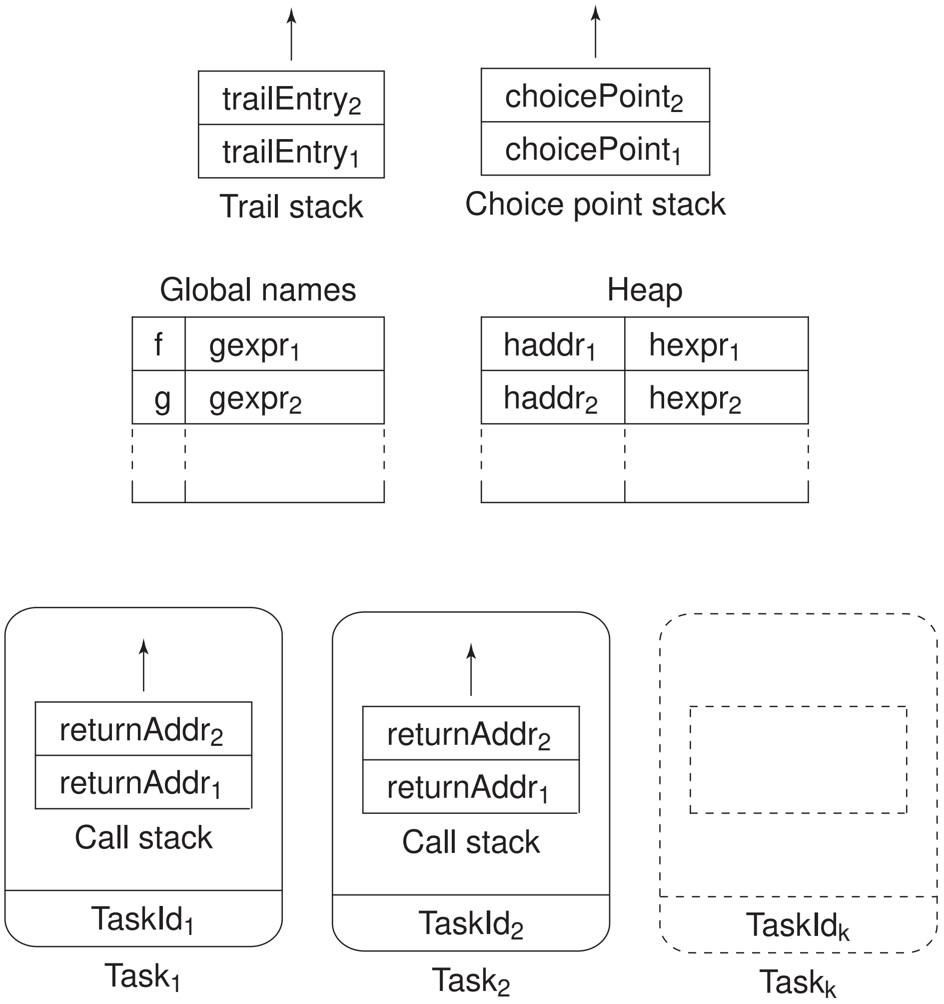

Figure 6: Structures of the abstract machine for residuation

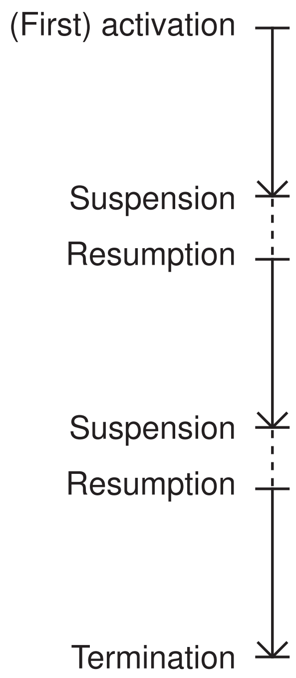

Figure 7: Active phases of a task

- *and* node disposal: Restore the *and* node.
- suspension count change: Undo the change.
- task suspension: Resume the task. Example: One constraint of a conjunction backtracks before the evaluation of the conjunction (and thus before the suspension of the parrent, see section 4.1.2).
- task resumption: Put the task back to sleep.
- addition to a suspension list: Remove the task's ID from the suspension list of the variable.
- binding to a free variable: Marks the variable as free and restores the suspension list. The re-suspension of these tasks is trailed separately by TrailResumption.
- task termination: Put the task back into the ready queue.

A choice point only saves the call stack for its own task. Therefore, when trying the next alternative of a choice point, extra care has to be taken so

that the stacks of *all* tasks are exactly as they were when that choice point was created.
Backtracking means stepping backward through the life of a task (figure 7). Untrailing usually does not touch the call stack. The only exception is that–when exiting a task upward while untrailing–one has to leave behind the call stack as it was before entering (while one is inside the task, choice points take care of restoring the call stack).
The only time that can happen is when untrailing a (first) activation or a resume. And in these instances, the call stack is correctly untrailed.

4.2.3 Starting evaluation from "outside"

$$\begin{array}{llllll} \mathit{collectAnswers} & :: \mathit{EvalState} & \rightarrow \mathit{VarDict} & \rightarrow \mathit{CurExpr} & \rightarrow \mathit{IO}\ [\mathit{Answer}] \\ \mathit{collectAnswers} & \ \ \ \mathit{state} & \ \ \ \mathit{dict} & \ \ \ \mathit{expr} & \end{array}$$

When the driver loop initially starts evaluation, it calls *collectAnswers* with a new EvalState, a dictionary of the variables whose value it wants to be part of the returned EvalAnswers and the expression to be evaluated.
Internally, *collectAnswers* uses the EvalMonad and continued backtracking to get all the answers to a query. The evaluation monad either finishes with normally returning a value, a global failure (essentially meaning that no more answers are available), a global suspension (all tasks are suspended) or an error. *collectAnswers* reacts to these cases accordingly:
That means that two different kinds of backtracking have to be implemented: *Global* backtracking is used to get all possible answers (for a query, from "outside"). Backtracking while executing a task is called *local*.
The only difference is that global backtracking takes the owner of the topmost choice point (on the stack) and makes it the active task. Afterwards, it continues with local backtracking for this task.
Local backtracking looks for the next choice point of the current task. If one untrails the activation of this task, backtracking continues with the next available choice point and its task.
If no more choice points are available, evaluation finishes with a global failure.

4.2.4 The prime task

The first task that is ever created, the so-called *prime task* plays a special role among its peers. As the evaluation monad has been initialized with this task, it has never been created by one of the usual scheduling functions. It never terminates, either (only the children of an and node do that).

Thus, the prime task is always present, because neither the untrailing of its creation nor its termination ever occurs. It also always is the active task when evaluation finishes normally.

5 Conclusion and further research

A lot remains to be done for Harry: As already mentioned in the introduction, only a subset of Curry has been implemented.
Apart from that, it would be interesting to experiment with other evaluation approaches: *Or* parallelism in addition to *and* parallelism, true concurrency etc.
Also, understanding the very compact form of the operational semantics of Curry with [HAK+99] wasn't always easy. Giving a definition that is better suited for implementations has already been done with [Sac99].

A Using Harry

Start Harry from the command line with `harry`. Optionally you can start it with `harry` *`filename`* where *filename* is the name of a file to be loaded after the standard prelude.
Once in Harry, the following commands are available (anything without a preceding ":" is evaluated as an expression):

```
HARRY> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filename>  load module from specified file
:reload           repeat last load command
:trace            toggle tracing
:?                display this list of commands
:quit             exit Harry interpreter
```

B Example code

Here are a two example Curry programs. For additional example code, have a look at [HAK+99].

B.1 ornodes.curry: Independence of or branches

```
pragma flex

f 1 1 = 11
f 2 2 = 22
f 1 2 = 33

g 1 = 44
g x = x
```

Queries:

```
HARRY> f x x
f x x
IN:  (f x x)
OUT: {x=2} 22 | {x=1} 11
HARRY> g x
g x
IN:  (g x)
OUT: {x=1} 44 | {x=_free0()} _free0()
```

B.2 kart.curry: Binding to sets of answers

```
data C = C Int Int

f x y = C x y

g = 1
g = 2

h x = C x x
```

Queries:

```
HARRY> f g g
f g g
IN:  (f g g)
OUT: {} (C[#0] 2 2) | {} (C[#0] 2 1) | {} (C[#0] 1 2) | {} (C[#0] 1 1)
HARRY> h g
h g
IN:  (h g)
OUT: {} (C[#0] 2 2) | {} (C[#0] 1 1)
```

C Example transformations

The following shows the transformation Harry applies to the example programs before interpreting them.

C.1 ornodes.curry

```
f _tc1 _tc2 =
   OR (GUARD (UNIFY _tc1 1)
             (OR (GUARD (UNIFY _tc2 1) 11)
                 (GUARD (UNIFY _tc2 2) 33)))
      (GUARD (UNIFY _tc1 2)
             (GUARD (UNIFY _tc2 2) 22))

g _tc1 = OR _tc1 (GUARD (UNIFY _tc1 1) 44)
```

C.2 kart.curry

```
f _tc1 _tc2 = C _tc1 _tc2
g           = OR 1 2
h _tc1      = C _tc1 _tc1
```

References

[HAK+99] Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco J. Lopez-Fraguas, and Frank Steiner. Curry–An Integrated Functional Logic Language. http://www-i2.informatik.rwth-aachen.de/~hanus/curry/papers/report.dvi.Z, January 1999.

[Sac99] David Sacher. Systematische Herleitung einer abstrakten Maschine für die funktional-logische Programmiersprache Curry. Master's thesis, Ludwig Maximilians Universität, München, 1999. In German.