ICurry

Sergio Antoy[1], Michael Hanus[2], Andy Jost[1], and Steven Libby[1]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
[2] Institut für Informatik, Kiel University, D-24098 Kiel, Germany

Abstract. FlatCurry is a well-established intermediate representation of Curry programs used in compilers that translate Curry code into Prolog or Haskell code. Some FlatCurry constructs have no direct translation into imperative code. These constructs must be each handled differently when translating Curry code into, e.g., C, C++ or Python code. We introduce a new representation of Curry programs, called ICurry, and derive a translation from all FlatCurry constructs into ICurry. We present the syntax and semantics of ICurry and the translation from FlatCurry to ICurry. We present a model of functional logic computations as graph rewriting and show how this model can be implemented with ICurry in a low-level imperative language.

1 Introduction

Functional logic languages [8] provide fast software prototyping and development, simple elegant solutions to otherwise complicated problems, a tight integration between specifications and code [9], and an ease of provability [10, 20] unmatched by other programming paradigms. Not surprisingly, these advantages place heavy demands on their implementation. Theoretical results must be proven and efficient models of execution must be developed. For these reasons, the efficient implementation of functional logic languages is an active area of research with contributions from many sources. This paper is one such contribution.

Compilers of high-level languages transform a *source* program into a *target* program which is in a lower-level language. This transformation maps constructs available in the source program language into simpler, more primitive, constructs available in the target program language. For example, pattern matching can be translated into a sequence of *switch* and assignment statements available in C, C++ and Python. We use this idea to map Curry into a C-like language. Our target language is not standard C, but a more abstract language that we call *ICurry.* The "I" in ICurry stands for "imperative", since a design goal of the language is to be easily mappable into an imperative language.

There are advantages in choosing ICurry over C. ICurry is simpler than C. It has no arrays, *typedef* declarations, types, explicit pointers, or dereferencing operations. ICurry is more abstract than concrete low level languages. Because of its simplicity and abstraction, it has been mapped with a modest effort to C, C++, and Python.

Section 2 is a brief overview of Curry, with focus on the features relevant to ICurry or to the examples. Section 3 discusses an operational model of execution for functional logic computations. This model can be implemented relatively easily in Curry or in common imperative languages. Section 4 presents *FlatCurry*, a format of Curry programs similar to ICurry. FlatCurry has been used in the translation of Curry into other, non-imperative, languages, but it is not suitable for the translation of Curry into an imperative language. Section 5 defines ICurry and its semantics, and discusses its generation and use. Section 6 addresses related work and offers our conclusion.

2 Curry

Curry is a declarative language that joins the most appealing features of functional and logic programming. A *Curry program* declares *data types*, which describe how information is structured, and defines *functions* or *operations*, which describe how information is manipulated. For example:

```
data List a = Nil | Cons a (List a)
```

declares a polymorphic type `List` in which `a` is a type parameter standing for the type of the list elements. The symbols `Nil` and `Cons` are the *constructors* of `List`. The values of a list are either `Nil`, the empty list, or `Cons` $e\ l$, a pair in which e is an element and l is a list.

Since lists are ubiquitous, a special notation eases writing and understanding them. Curry uses `[]` to denote the empty list and $e : l$ to denote the pair, where the infix constructor ":" associates to the right. A finite list is written `[`e_1`,`…`,`e_n`]`, where e_i is a list element. For example, `[1,2,3]` = `1:2:3:[]`.

Functions are defined by rewrite rules of the form:

$$\begin{aligned} f\ \bar{p} \mid c_1 &= e_1 \\ \cdots& \\ \mid c_n &= e_n \end{aligned} \tag{1}$$

where f is a function symbol, $\bar{p}$ stands for a sequence of zero or more expressions made up only of constructor symbols and variables, "$\mid c_i$" is a condition, and e_i is an expression. Conditions in rules are optional. The expressions in $\bar{p}$ are called *patterns*. For example, consider:

```
abs x | x <  0 = -x
      | x >= 0 =  x
length []     = 0
length (_:xs) = 1 + length xs
```
(2)

where `abs` computes the absolute value of its argument and shows some conditions, and `length` computes the length of its argument and shows some patterns.

In contrast to most other languages, the textual order of the rewrite rules in a program is irrelevant—all the rules that can be applied to an expression are applied. An emblematic example is a function, called *choice*, and denoted by the infix operator "`?`", which chooses between two *alternatives*:

```
x ? y = x
x ? y = y
```

Therefore, `0 ? 1` is an expression that produces `0` and `1` non-deterministically. In Curry, there are many other useful syntactic and semantic features, for example, rewrite rules can have nested scopes with local definitions. We omit their description here, since they are largely irrelevant to our discussion, with the exception of *let blocks* and *free variables*.

Let blocks support the definition of circular expressions which allows the construction of cyclic graphs. Fig. 1 shows an example of a let block and the corresponding graph. Expression `oneTwo` evaluates to the infinite list `1:2:1:2:`...

```
oneTwo = let x = 1 : y
             y = 2 : x
         in x
```

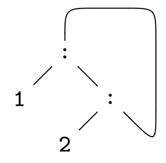

Fig. 1. Example of a let block with mutually recursive variables and the graph it defines.

Free variables abstract unknown information and are "computationally inert" until the information they stand for is required during a computation. When this happens, plausible values for a variable are non-deterministically produced by narrowing [6, 29]. Free variables might occur in initial expressions, conditions, and the right-hand side of rules, and need to be declared by the keyword `free`, unless they are anonymous (denoted by "`_`"). For instance, the following program defines list concatenation which is exploited to define an operation that returns some element of a list having at least two occurrences:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

someDup :: [a] → a
someDup xs | xs == _ ++ [x] ++ _ ++ [x] ++ _
           = x    where x free
```

3 The Execution Model

A *program* is a graph rewriting system [16, 28] over a *signature*, partitioned into *constructor* and *operation* symbols. We briefly and informally review the underlying theory. A *graph* is a set of *nodes*, where a node is an object with some attributes, and an identity by virtue of being an element in a set. Key attributes of a node are a *label* and a sequence of *successors*. A label is either a symbol of the signature or a variable. A successor is another node, and the sequence of successors may be empty. Exactly one node of a graph is designated as the

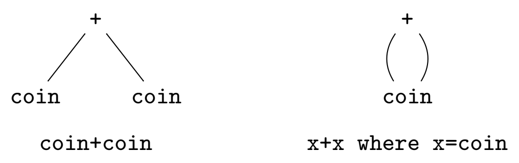

Fig. 2. Graphical and textual representation of expressions. In Curry, all the occurrences of the same variable are shared. Hence, the two occurrences of `x` stand for the same node. The expression `coin` is conventionally an integer constant with two values, `0` and `1`, non-deterministically chosen. The sets of values produced by the two expressions differ.

graph's *root*. Each node of a graph corresponds to an expression in the Curry program.

A *graph rewriting system* is a set of rewrite rules following the *constructor discipline* [27]. A *rule* is a pair of graphs, $l \to r$, called the left- and right-hand sides, respectively. Rules are unconditional without loss of generality [3]. A *rewrite step* of a graph e first identifies both a subgraph t of e, and a rule $l \to r$ in which t is an instance of l, then replaces t with the corresponding instance of r. The identification of the subgraph t and the rule $l \to r$ is accomplished by a *strategy* [4]. For example, given the rules (2), a step of `length [3,4]` produces `1+length[4]` where the subgraph reduced in the step is the whole graph, and the rule applied in the step is the second one.

A *computation* of an expression e is a sequence of rewrite steps starting with e, $e = e_0 \to e_1 \to \ldots$ Expression e is referred to as *top-level*, and each e_i as a *state* of the computation of e. A *value* of a computation is a state in which every node is labeled by a constructor symbol. Such expression is also called a *constructor normal form.* Not every computation has values.

We have modeled a functional logic program as a graph rewriting system [16, 28]. Functional logic computations are executed in this model by rewriting which consists of two relatively simple operations: the construction of graphs and the replacement of subgraphs with other graphs. The most challenging part is selecting the subgraph to be replaced in a way that does not consume computational resources unnecessarily. This is a well-understood problem [4] which is largely separated from the model.

In an implementation of the model, the expressions are objects of a computation and are represented by dynamically linked structures. These structures are similar to those used for computing with lists and trees. The nodes of such a structure are in a bijection with the nodes of the graph they represent. Unless a distinction is relevant, we do not distinguish between a graph and its representation.

The occurrence of a symbol, or variable, in the textual representation of an expression stands for the node labeled by the occurrence. Distinct occurrences may stand for the same node, in which case we say that the occurrences are *shared.* The textual representation accommodates this distinction, therefore it is a convenient, linear notation for a graph. Fig. 2 shows two graphs and their corresponding textual expressions.

4 FlatCurry

FlatCurry [17] is an intermediate language used in a variety of applications. These applications include implementing Curry by compiling into other languages, like Prolog [21] or Haskell [12]. FlatCurry is also the basis for specifying the operational semantics of Curry programs [1], building generic analysis tools [22], or verifying properties of Curry programs [19, 20]. The FlatCurry format of a Curry program removes some syntactic constructs, such as nested scopes and infix notation, that make source programs more human readable. This removal still preserves the program's meaning. We ignore some elements of FlatCurry, such as imported modules or exported symbols, which are not directly related to the execution model presented in Section 3. Instead, we focus on the declaration of data constructors, the definition of functions, and the construction of expressions. These are the elements that play a central role in our execution model.

FlatCurry is a machine representation of Curry programs. As such, it is not intended to be read by human. For example, each variable is identified by an integer, function application is only prefix, and pattern matching is broken down into a cascade of case distinctions. In the examples that follow, we present a sugared version of FlatCurry in which variables have symbolic names, typically the same as in Curry; the application of familiar infix operators is infix; and indentation, rather than parentheses and commas, show structure and grouping. The intent is to make the examples easier to read without altering the essence of FlatCurry.

In FlatCurry, data constructors are introduced by a type declaration. A type t has attributes such as a name and a visibility, and chief among these attributes is a set of constructors $c_1, c_2, \ldots c_n$. Each constructor c_i has similar attributes, along with an arity and type of each argument, which are not explicitly used in our discussion. The same information is available for operation symbols. Additionally, any operation f has an attribute that abstracts the set of the rules defining f.

The abstract syntax of FlatCurry operations is summarized in Fig. 3.[3] Each operation is defined by a single rule with a linear left-hand side, i.e., the argument variables $x_1, \ldots, x_n$ are pairwise different. The right-hand side of the definition consists of (1) variables introduced by the left-hand side or by a *let block* or by a *case* pattern, (2) constructor or function calls, (3) *case* expressions, (4) disjunctions, (5) *let* bindings, or (6) introduction of free variables. The patterns p_i in a *case* expression must be pairwise different constructors applied to variables. Therefore, deep patterns in source programs are represented by nested *case* expressions.

Case expressions closely resemble definitional trees [2]. We recall that a *definitional tree* of some operation f, of arity n, is a hierarchical structure of expressions of the form $f\ p_1\ \ldots\ p_n$, where each p_i is a pattern. Since f is constant and

[3] In contrast to some other presentations of FlatCurry (e.g., [1, 18]), we omit the difference between rigid and flexible case expressions.

$$
\begin{array}{lll}
D & ::= f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= x & \text{(variable)} \\
 & \mid\ c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & \mid\ f(e_1, \ldots, e_n) & \text{(function call)} \\
 & \mid\ \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(case expression)} \\
 & \mid\ e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
 & \mid\ \mathit{let}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \mathit{in}\ e & \text{(let binding)} \\
 & \mid\ \mathit{let}\ x_1, \ldots, x_n\ \mathit{free}\ \mathit{in}\ e & \text{(free variables)} \\
p & ::= c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Fig. 3. Abstract syntax of function definitions in FlatCurry

provides no information, except to ease readability, we also call these expressions patterns. The pattern at the root of the tree is $f\ x_1\ \ldots\ x_n$, where the x_i's are distinct variables. The patterns at the leaves are the left-hand sides of the rules of f, except from the names of the variables. For ease of understanding, in pictorial representations of definitional trees we add the right-hand side of the rules too. If $f\ p_1\ \ldots\ p_n$ is a branch node, β, of the tree, a variable x in some p_j is singled out. We call the variable x *inductive*. The pattern in a child of β is $f\ p_1\ \ldots\ q_j\ \ldots\ p_n$ where q_j is obtained from p_j by replacing x with $c\ y_1\ \ldots\ y_k$, where c is a constructor of the type of x and each y_i is a fresh variable. For example, consider the usual operation `zip` for zipping two lists:

```
zip []      y       = []
zip (x1:x2) []      = []                                  (3)
zip (x1:x2) (y1:y2) = (x1,y1) : zip x2 y2
```

The corresponding definitional tree is shown below where the inductive variable is boxed.

$$\begin{tikzcd}
 & \texttt{zip}\ \boxed{\texttt{x}}\ \texttt{y} \arrow[dl, dash] \arrow[dr, dash] & & \\
\texttt{zip [] y} \arrow[dd] & & \texttt{zip (x1:x2)}\ \boxed{\texttt{y}} \arrow[dl, dash] \arrow[dr, dash] & \\
 & \texttt{zip (x1:x2) []} \arrow[d] & & \texttt{zip (x1:x2) (y1:y2)} \arrow[d] \\
\texttt{[]} & \texttt{[]} & & \texttt{(x1,y1) : zip x2 y2}
\end{tikzcd}$$

The FlatCurry code of the rules of operation `zip`, closely corresponds to the code in (4). This would be harder for the programmer to write than (3) and less readable, but is semantically equivalent. Every program can be transformed into an equivalent program in which every operation has a definitional tree [3].

There is a relatively simple algorithm [4] to construct a definitional tree from the operation's rules.

```
zip x y = case x of
          { []       →  [] ;
            (x1:x2)  →  case y of                                    (4)
                        { []       →  [] ;
                          (y1:y2)  →  (x1,y1) : zip x2 y2 }}
```

Expressions are the final relevant element of FlatCurry. As the code of `zip` shows, an expression can be a literal, like `[]`; an application of constructors and operations to expressions possibly containing variables, like `(x1,y1) : zip x2 y2`; or a case expression, like `case y of` ... FlatCurry also has *let blocks* to support the construction of cyclic graphs, as shown in Fig. 1.

FlatCurry programs cannot be directly mapped to code in a C-like target language. There are two problems: case expressions as arguments of a symbol application, and let blocks with shared or mutually recursive variables. A contrived example of the first is:

```
 3 + case x of { []  →  0; (y:ys)  →  y }
```

Since the evaluation of the scrutinee of a case expression might yield a non-deterministic result, it cannot be directly mapped into imperative language constructs. An example of the second is shown in Fig. 1. ICurry proposes a solution to these problems in a language-independent form which is suitable for the imperative paradigm.

5 ICurry

In this section we define ICurry, discuss how to map it to imperative code that implements our earlier model of computation, and show how to obtain it from FlatCurry.

5.1 ICurry Definition

ICurry is a format of Curry programs similar in intent to FlatCurry. The purpose of both is to represent a Curry program into a format with a small number of simple constructs. Properties and manipulations of programs can be more easily investigated and executed in these formats. ICurry is specifically intended for compilation into a low-level language. Each ICurry construct can be translated into a similar construct of languages such as C, Java or Python. This should become apparent once we describe the constructs.

ICurry's data consists of nested applications of symbols represented as graphs. ICurry's key constructs provide the declaration or definition of symbols and variables, construction of graph nodes, assignment, and conditional executions of these constructs. Rewriting steps are implemented in two phases, once the redex and rule are determined. First, the replacement of the redex is constructed. This is defined by the right-hand side of the rule. Then, the successors pointing to

D	$::=$	$f = blck$	(function definition)
$blck$	$::=$	$decl_1 \ldots decl_k\ asgn_1 \ldots asgn_n\ stm$	(block)
$decl$	$::=$	$declare\ x$	(local variable declaration)
	$\mid$	$free\ x$	(free variable declaration)
$asgn$	$::=$	$v = exp$	(variable assignment)
stm	$::=$	$return\ exp$	(return statement)
	$\mid$	$exempt$	(failure statement)
	$\mid$	$case\ x\ of\ \{c_1 \to blck_1; \ldots; c_n \to blck_n\}$	(case statement)
exp	$::=$	v	(variable)
	$\mid$	$NODE(l, exp_1, \ldots, exp_n)$	(node construction)
	$\mid$	$exp_1\ or\ exp_2$	(disjunction)
v	$::=$	x	(local variable)
	$\mid$	$v[i]$	(node access)
	$\mid$	$ROOT$	(root of function call)
l	$::=$	c	(constructor symbol)
	$\mid$	f	(function symbol)

Fig. 4. Abstract syntax of function definitions in ICurry

the root of the redex are redirected [16, Def. 8], through assignments, to point to the root of the replacement.

The declaration of data constructors in ICurry is identical to that in FlatCurry as described earlier. However, the constructors of a type are in an arbitrary, but fixed, order. Therefore, we can talk of the first, second, etc., constructor of a type. This index is an attribute of constructor symbols which we call the *tag*. The tag is used to provide efficient pattern matching. We will return to this topic in Sect. 5.4.

The abstract syntax of operations in ICurry is summarized in Fig. 4. In FlatCurry, the body of a function is an expression. In ICurry, it is a block consisting of optional declarations and/or assignments and a final statement returning an expression. We describe expressions first.

Expressions are nested symbol applications represented as graphs. Therefore, an expression is either a *variable* or a *symbol application*. ICurry makes an application explicit with a *directive*, `NODE`, that constructs a graph node from its label (1st argument) and its successors (remaining arguments), and returns a reference to the node. Accordingly, there is a directive to access node components: assuming x is a variable referring to a node, $x[k]$ retrieves the k-th successor of the node. An ICurry variable v is a reference to a node n in a graph. When n is a Curry free variable, v is called *free* as well. Otherwise, v is called a *local* variable. The ICurry format distinguishes between constructor and function application, and between full and partial application. We do not discuss these details in this paper. It is expected that by providing this additional information, processors will be able to generate low-level code more easily, and the generated code should be easier to optimize.

In ICurry, there are only a handful of statement kinds: *declaration* of a variable, and *assignment* to a variable, *return*, and *case* expressions. Following FlatCurry, variables are represented by integers. A declaration introduces a variable which is a reference to a graph node. Successors of a node referenced by x are accessed through the $x[\ldots]$ construct. Arguments passed to functions are accessed through local variables. The return statement is intended to return an expression, the result of a function call. Case expressions in ICurry are structurally similar to those in FlatCurry, but with two differences for algebraically defined types, which have a finite number of data constructors. First, the branches of a case expression are in tag order. We will justify this decision in Sect. 5.4. Second, the set of branches of a case expression is complete, i.e., there is a branch for each constructor of the type.

ICurry code begins with a declaration, and possible assignment, of some variables. It is then followed by either a case statement, or a return statement. Each branch of the case expression may declare and assign variables, and may lead to either another case statement, or a return statement.

Below, we present two examples. The first example is the code of function `oneTwo`, a constant, of Fig. 1:

```
function oneTwo
  declare x
  declare y
  x = NODE(:, NODE(1), y)
  y = NODE(:, NODE(2), x)
  x[2] = y
  return x
```

Symbol application is explicit through `NODE`. In the above example, the definitions of the nodes referenced by `x` and `y` are mutually recursive, thus either node cannot be completely constructed before constructing the other. We resolve the impasse by partially constructing the node referenced by `x` (starting with `y` would be symmetric), constructing the node referenced by `y`, and finally coming back to `x` and finish the job. The missing information when the node referenced by `x` is constructed is the value of the node's second successor, which is addressed by `x[2]`. This value becomes known when the node referenced by `y` is constructed. At that point, the missing information is filled in with the assignment to `x[2]`.

The second example is the code of `head`, the usual function returning the head of a non-empty list:

```
head (x:_) = x                                                    (5)
```

The rule of `head` for the argument `[]` is missing in the Curry source code. Consequently, the case branch for the argument `[]` is missing in FlatCurry, too. ICurry has a distinguished statement, `exempt`, to capture the absence of a rule:

```
function head
  declare arg
  arg = ROOT[1]
  case arg of
```

```
        []  →  exempt
        :   →  return arg[1]
```

where ROOT is a reference to the root of the expression being evaluated. This expression is rooted by head, which is the reason why it is passed to function head.

5.2 Operational Semantics of ICurry

In this section, we define a small-step semantics for ICurry programs. We are motivated by the fact that ICurry is very similar to a simple imperative language, but has primitives to support non-deterministic computations. These primitives are the or expression, used to introduce non-determinism, and the exempt statement, used to express a failing branch of a computation.

Non-deterministic choices in a program execution require copying a computation into two branches. In order to reduce the effort for copying, pull-tabbing [5] can be used. Fundamentally, a *pull-tab step* moves a choice occurring in a demanded argument of an operation outside this operation. For instance, if f demands the value of its single argument, then the following is a pull-tab step.

$$f\ (e_1\ ?\ e_2) \quad \to \quad (f\ e_1)\ ?\ (f\ e_2)$$

Although ICurry's non-determinism can also be implemented with other strategies such as stack copying, we use pull-tabbing here due to its limited demand to copy structures. For this purpose, we make the following assumptions:

1. Each ICurry function contains at most one case statement. This can be obtained by replacing nested case statements by auxiliary operations.[4] Therefore, we denote by f^i an ICurry function which demands its i-th argument in a case statement, otherwise the superscript is omitted.
2. A graph might also contain *choice nodes* of the form $\texttt{?}^c(n_1, n_2)$. The expressions n_1, n_2 are the alternatives, and c is a choice identifier which is an integer uniquely determined when the choice node is created. Choice identifiers are necessary to distinguish choices in different computation branches [5, 12].

As discussed in Sect. 3, the execution model of Curry is based on graph rewriting. Therefore, the main component of ICurry's run-time system is a graph G. In the subsequent description, we use the following notation. We write $G[n] = s(n_1, \ldots, n_k)$ if n is a node of G with label s and successor nodes $n_1, \ldots, n_k$. The update of a node n of G is denoted by $G[n \leftarrow s(n_1, \ldots, n_k)]$. The label of n is replaced by s and the successors of n are set to $n_1, \ldots, n_k$. In order to implement sharing, it is sometimes necessary to redirect a graph node n to a node n' of a graph G. We denote this by $G[n \leftarrow n']$. This can be implemented either by a specific "redirection node" or by redirecting all edges pointing to n so that they point to n'. Finally, we denote the extension of a graph G with a

[4] Some implementations of Curry, e.g., [21] perform this transformation.

new node n by $G \uplus \{n : s(n_1, \ldots, n_k)\}$. The node n does not exist in G and has label s and successors $n_1, \ldots, n_k$.

In order to deal with non-deterministic computations, the run-time system manages a queue of computation tasks, where each task consists of a control block, a stack of pending computations, and a fingerprint [11] managing the consistency of non-deterministic choices for the task. To be more precise, the state of an ICurry computation is a triple (G, Q, R) where the components have the following structure:

- G is graph where each node is labeled with a function, constructor, or the choice symbol, "?". As discussed above, a choice node n has the form $G[n] = ?^c(n_1, n_2)$.
- Q is a queue (list) of tasks where each task is a triple (C, S, F) with:
  - C is the *control* which is either a graph node n to be evaluated or a pair (b, E) consisting of a block of ICurry (see Fig. 4) and an *environment* E (a mapping from local variables to graph nodes).
  - S is a stack where each stack element is a node n labeled by a function symbol. The stack contains the functions to be evaluated by a task.
  - F is a *fingerprint*, which is a (partial) mapping from choice identifiers to indexes of alternatives.
- R is the set of computed results, which are graph nodes. Note that ICurry evaluates expressions to head normal forms, that is graphs with a constructor at the root. This is sufficient since the evaluation to normal form can be implemented by auxiliary operations.

In the following, we use $\phi[x \mapsto v]$ to denote an update of a mapping ϕ for some argument v. If $\phi' = \phi[x \mapsto v]$, then $\phi'(x) = v$ and $\phi'(y) = \phi(y)$ for all $y \neq x$. Furthermore, we use Curry's list notation for states. Thus, an initial state of an ICurry computation has the form

$$(G, [(n, [], \{\})], \{\})$$

where the graph G contains the initial expression with root node n. Thus, there is only one task with an empty stack and fingerprint and an empty set of results. A final computation state has the form:

$$(G, [], R)$$

There are no tasks left and the set R contains the root nodes of all computed results.

We specify the small-step semantics of ICurry by a set of transformation rules on states. Some of the rules use an auxiliary operation *extend* to extend a graph by adding the graph representation of an expression occurring in an ICurry program. Informally, $\mathit{extend}(G, E, e)$ extends a graph G with an ICurry expression e w.r.t. an environment E and returns the pair (G', n) consisting of the extended graph and the root node n of the added expression. To define *extend*, we use an auxiliary function *lookup* to retrieve a graph node w.r.t. an environment:

$$lookup(G,E,v) = \begin{cases} E(v) & \text{if } v = x \text{ or } v = ROOT \\ n_i & \begin{array}{l}\text{if } v = v'[i],\ lookup(G,E,v') = n \\ \text{and } G[n] = l(n_1,\ldots,n_k)\end{array} \end{cases}$$

If e is a variable, its binding is looked up in the environment E and returned as n without extending the graph:

$$extend(G,E,v) = (G, lookup(G,E,v))$$

A disjunction e_1 *or* e_2 creates new subgraphs for the arguments e_1 and e_2 and connects them by a new choice node:

$$\begin{array}{l} extend(G,E,e_1 \textit{ or } e_2) = G'' \uplus \{n : ?^c(n_1,n_2)\} \\ \quad \text{if } extend(G,E,e_1) = (G',n_1) \text{ and } extend(G',E,e_2) = (G'',n_2) \end{array}$$

Here, c is a new choice identifier. We assume the existence of a global set of choice identifiers so that new unique identifiers can be obtained during the computation. Similarly, a node constructor creates new subgraphs for the argument expressions and a new node connecting these subgraphs. We assume that n_i is the root node for the subgraph created for e_i and G' is the graph containing G and the new subgraphs:

$$extend(G,E,NODE(l,e_1,\ldots,e_k)) = G' \uplus \{n : l(n_1,\ldots,n_k)\}$$

Now we can specify a small-step semantics of ICurry by the following transformation rules:

Function node: If the control contains a graph node labeled with a defined function whose i-th argument is demanded, the function node is put onto the stack and the control is replaced by the i-th argument:

$$\begin{array}{l} (G,(n,S,F):Q,R) \ \rightarrow\ (G,(n_i,n:S,F):Q,R) \\ \text{if } G[n] = f^i(n_1,\ldots,n_k) \end{array}$$

If the control contains a graph node labeled with a defined function which does not demand an argument, the function's body is put into the control together with an environment initialized with the graph node:

$$\begin{array}{l} (G,(n,S,F):Q,R) \ \rightarrow\ (G,((b,\{ROOT \mapsto n\}),S,F):Q,R) \\ \text{if } G[n] = f(\ldots) \text{ and } f = b \text{ is a declaration of the ICurry program} \end{array}$$

Variable declaration: If the control starts with a declaration of a local variable, it is initialized as a *null* pointer in the environment:

$$(G,((declare\ x; b,E),S,F):Q,R) \ \rightarrow\ (G,((b,E[x \mapsto null]),S,F):Q,R)$$

Free variables can be handled in various ways. For the sake of simplicity, we implement free variables as non-deterministic generator operations. This technique is also used in KiCS2 [12] and stems from the equivalence of logic variables and non-determinism [7]. For instance, a generator for a Boolean free variable can be defined as:

```
gen_Bool = False ? True
```

Since free variables of different types will have a different generator operation, we denote by gen_x the generator operation of the free variable x.[5] Then a free variable is introduced by initializing it with a node representing the generator operation:

$$\begin{array}{l}(G, ((\mathit{free}\ x; b, E), S, F) : Q, R) \ \rightarrow \\ (G \uplus \{n : \mathit{gen}_x()\}, ((b, E[x \mapsto n]), S, F) : Q, R)\end{array}$$

Assignment: If the control starts with an assignment to a local variable, the graph is extended with the expression and the environment is updated:

$$\begin{array}{l}(G, ((x = e; b, E), S, F) : Q, R) \ \rightarrow \ (G', ((b, E[x \mapsto n]), S, F) : Q, R) \\ \text{if } \mathit{extend}(G, E, e) = (G', n)\end{array}$$

If the control starts with an assignment to successor of a node, the graph is extended with the expression and the successor is set to the created subgraph:

$$\begin{array}{l}(G, ((v[i] = e; b, E), S, F) : Q, R) \ \rightarrow \\ (G'[n \leftarrow l(n_1, \ldots, n_{i-1}, n', n_{i+1}, \ldots, n_k)], ((b, E), S, F) : Q, R) \\ \text{if } \mathit{lookup}(G, E, v) = n,\ G[n] = l(n_1, \ldots, n_k),\ \mathit{extend}(G, E, e) = (G', n')\end{array}$$

Return statement: If the control contains a return statement, the graph is extended with the returned graph and the root of the current function is updated with the returned node:

$$\begin{array}{l}(G, ((\mathit{return}\ e, E), S, F) : Q, R) \ \rightarrow \ (G'[E(\mathit{ROOT}) \leftarrow n], (n, S, F) : Q, R) \\ \text{if } \mathit{extend}(G, E, e) = (G', n)\end{array}$$

Exempt statement: If the control contains an exempt statement, the current computation is removed from the list of tasks:

$$(G, ((\mathit{exempt}, E), S, F) : Q, R) \ \rightarrow \ (G, Q, R)$$

Case statement: If the control contains a case statement, the corresponding branch is selected (this is always possible since the case argument is demanded and was evaluated before invoking the function):

$$\begin{array}{l}(G, ((\mathit{case}\ x\ \mathit{of}\ \{c_1 \rightarrow b_1; \ldots; c_n \rightarrow b_n\}, E), S, F) : Q, R) \ \rightarrow \\ (G, ((b_i, E) : S, F) : Q, R) \\ \text{if } E(x) = n \text{ and } G[n] = c_i(\ldots)\end{array}$$

[5] Type-based generators can be implemented with type classes, as described in [23]. Thus, a compiler can easily attach appropriate generators to free variables in ICurry.

Constructor node: If the control contains a graph node labeled with a constructor symbol, we distinguish two cases. If the stack is empty, a result has been computed:

$$(G, (n, [], F) : Q, R) \;\to\; (G, Q, R \cup \{n\})$$
if $G[n] = c(\ldots)$ for some constructor c

If the stack is not empty, then it contains a function where an argument is demanded. Since this argument, which is the node in control, is evaluated, we invoke this function by putting its body into the control:

$$(G, (n, (n' : S, F) : Q, R) \;\to\; (G, ((b, \{ROOT \mapsto n'\}), S, F) : Q, R)$$
if $G[n] = c(\ldots)$ for some constructor c, $G[n'] = f^i(\ldots)$,
and $f^i = b$ is a declaration in the ICurry program

Choice node: If the control contains a choice node, we distinguish three cases. If the stack is empty, i.e., the choice is at the top, and the fingerprint already selects a branch for this choice, the choice node is replaced by the corresponding branch:

$$(G, (n, [], F) : Q, R) \;\to\; (G, (n_i, [], F) : Q, R)$$
if $G[n] = \mathtt{?}^c(n_1, n_2)$ and $F(c) = i$

If the stack is empty and the fingerprint does not contain a selection for this choice, we split the current task into two new tasks where the fingerprint is extended in each task:

$$(G, (n, [], F) : Q, R) \;\to\; (G, Q \mathbin{\texttt{++}} [(n_1, [], F[c \mapsto 1]), (n_2, [], F[c \mapsto 2])], R)$$
if $G[n] = \mathtt{?}^c(n_1, n_2)$ and $F(c)$ is undefined

Note that we can use any strategy to add the new tasks to the existing ones. Here we put them at the end which corresponds to a breadth-first strategy in the search tree. Putting them at the front of Q corresponds to a depth-first search strategy. Some Curry implementations, like KiCS2 [12], allow the user to select different search strategies.

The final case is a pull-tab step. If the choice is at a demanded argument position, then the stack is not empty, and the graph node identified by the top of the stack is replaced by a choice:

$$(G, (n_0, n : S, F) : Q, R) \;\to\; (G'[n \leftarrow \mathtt{?}^c(n_1', n_2')], (n, S, F) : Q, R)$$
if $G[n_0] = \mathtt{?}^c(n_1, n_2)$, $G[n] = f^i(n_1, \ldots, n_k)$,
and $G' = G \uplus \{n_1' : f^i(n_1, \ldots, n_{i-1}, n_1, n_{i+1}, \ldots, n_k),$
$\qquad n_2' : f^i(n_1, \ldots, n_{i-1}, n_2, n_{i+1}, \ldots, n_k)\}$

Since each transformation step performs only local changes, the implementation effort for these steps is limited when mapping ICurry into an imperative language. This will be shown in Sect. 5.4 where implementations of ICurry in various imperative languages are summarized.

5.3 ICurry Generation

Current Curry distributions such as PAKCS [21] or KiCS2 [12] provide a package with the definition of FlatCurry and a rich API for its construction and manipulation. Therefore, the ICurry format of a Curry program is conveniently obtained from the FlatCurry format of that program.

A fundamental difference between the two formats concerns expressions. Expressions in FlatCurry may contain *cases* and *lets* as the arguments of a function application. These are banned in ICurry which allows only nested functional application. The reason is that the latter can be directly translated into various imperative languages, where the former cannot. Therefore, any *case* and *let* constructs that are the arguments of a function application are replaced by calls to newly created functions. Thus, in ICurry, the replaced constructs are executed at the top level. We replace these constructs during the transformation from FlatCurry into ICurry. However, the same transformation could be performed from FlatCurry into itself, or even from source Curry into itself. Our contrived example below shows the latter for ease of understanding. The code of function `g` is irrelevant, therefore, it is not shown:

```
f x = g x (case x of ...)
```

is transformed into:

```
f x = g x (h x)
h x = case x of ...
```

The offending *case*, as an argument of the application of `g`, has been replaced by a call to a newly created function, `h`. In function `h`, the *case* is no longer an argument of a function application.

The second major difference between FlatCurry and ICurry concerns case expressions. FlatCurry matches a selector against shallow constructor expressions, where ICurry matches against constructor symbols. Furthermore, the set of these symbols is complete and ordered in ICurry. The transformation is relatively simple, except it may require non-local information. A function in a module M may pattern match on some instance of a type t that is not declared in M. Therefore, the constructors of t must be accessed in some module different from that being compiled.

A third significant difference between FlatCurry and ICurry concerns *let blocks*. They are banned in ICurry, and replaced by the explicit construction of nodes, and by the assignment of these nodes' references to local variables.

In the following, we show an algorithm to translate FlatCurry into ICurry. For this purpose, we define a *pure expression* as an expression that only contains literals, variables, constructor applications, and function applications. Any *or* expression and function application may only contain pure expressions. The scrutinee of a case expression must be a variable, literal, or constructor application. An assignment in a *let* expression must be a pure expression or an *or* expression. The branches of a case expression must match all constructors of a data type in an order fixed by the definition of that data type. Branches missing in the original Curry program contain $\perp$ in their right-hand side.

```
$\mathcal{F}(f(x_1,\ldots,x_n) = e) := f = \mathcal{B}(x_1,\ldots,x_n,e,ROOT)$

$\mathcal{B}(x_1,\ldots,x_n,\bot,root) := exempt$
$\mathcal{B}(x_1,\ldots,x_n,e,root) :=$
  $declare\ x_1$
  $\ldots$
  $declare\ x_n$
  $\mathcal{D}(e)$
  $x_1 = root[1]$
  $\ldots$
  $x_n = root[n]$
  $\mathcal{A}(e)$
  $return\ \mathcal{E}(e)$    (omit $return$ if $\mathcal{E}(e)$ is a $case$)

$\mathcal{D}(let\ x_1,\ldots,x_n\ free\ in\ e) :=$
  $free\ x_1$
  $\ldots$
  $free\ x_n$
$\mathcal{D}(let\ \{x_1 = e_1;\ldots;x_n = e_n\}\ in\ e) :=$
  $declare\ x_1$
  $\ldots$
  $declare\ x_n$
$\mathcal{D}(case\ e\ of\ \{p_1 \to e_1;\ldots;p_n \to e_n\}) := declare\ x_e$

$\mathcal{A}(let\ \{x_1 = e_1;\ldots;x_n = e_n\}\ in\ e) :=$
  $x_1 = \mathcal{E}(e_1)$
  $\ldots$
  $x_n = \mathcal{E}(e_n)$
  $x_1[p] = x_i$   (for each occurrence of $x_i, i \geq 1,$ in $e_1$ at position $p$)
  $\ldots$
  $x_n[p] = x_i$   (for each occurrence of $x_i, i \geq n,$ in $e_n$ at position $p$)
$\mathcal{A}(case\ e\ of\ \{p_1 \to e_1;\ldots;p_n \to e_n\}) := x_e = \mathcal{E}(e)$

$\mathcal{E}(x)$                                   $:=$ $x$
$\mathcal{E}(c(e_1,\ldots,e_n))$                   $:=$ $NODE(c,\mathcal{E}(e_1),\ldots,\mathcal{E}(e_n))$
$\mathcal{E}(f(e_1,\ldots,e_n))$                   $:=$ $NODE(f,\mathcal{E}(e_1),\ldots,\mathcal{E}(e_n))$
$\mathcal{E}(e_1\ or\ e_2)$                        $:=$ $\mathcal{E}(e_1)\ or\ \mathcal{E}(e_2)$
$\mathcal{E}(let\ \{x_1 = e_1;\ldots;x_n = e_n\}\ in\ e)$  $:=$ $\mathcal{E}(e)$
$\mathcal{E}(let\ \{x_1,\ldots,x_n\}\ free\ in\ e)$        $:=$ $\mathcal{E}(e)$
$\mathcal{E}(case\ e\ of\ \{c(x_{11},\ldots,x_{1m}) \to e_1;\ldots;c(x_{n1},\ldots,x_{nk}) \to e_n\})$ $:=$
   $case\ \mathcal{E}(e)\ of\ \{\ \mathcal{B}(x_{11},\ldots,x_{1m},e_1,x_e);$
                  $\ldots;$
                  $\mathcal{B}(x_{n1},\ldots,x_{nk},e_n,x_e);\ \}$
```

Fig. 5. Algorithm for translating FlatCurry into ICurry

The algorithm is divided into five functions which are described in Fig. 5. $\mathcal{F}$ translates a FlatCurry function into an ICurry function. $\mathcal{B}$ translates a FlatCurry expression into an ICurry block. $\mathcal{D}$ extracts all of the variables declared in a FlatCurry expression. $\mathcal{A}$ generates necessary assignments for ICurry variables. $\mathcal{E}$ translates a FlatCurry expression into an ICurry expression.

The functions $\mathcal{F}$ and $\mathcal{E}$ are straightforward translations. $\mathcal{F}$ simply makes a block, with the root of the block being set to the root of the function. $\mathcal{E}$ is almost entirely a straight translation, but there is one technical point. In a case expression, each branch must be translated into its own block. However, each of the variables in the pattern of a branch need to be related to the scrutinee of the case. This is achieved by setting the root of the block to the scrutinee of the case.

The function $\mathcal{B}$ creates an ICurry block. Blocks are more complicated to construct. Each block will have a root and a list of variables. The root is the root of the expression that created the block. For a function, the root is the root of the function expression. For a case branch, the root is the root of the scrutinee of the case. The variables of a block are the parameters of a function, or the pattern variables of a branch. After declaring variables, all variables in any *let* expressions are declared with $\mathcal{D}$. Then each variable is assigned an expression with $\mathcal{A}$. If either $\mathcal{D}$ or $\mathcal{A}$ is undefined for some expression, their application does not generate ICurry code. Finally, we translate the expression into an ICurry statement with $\mathcal{E}$.

The function $\mathcal{D}$ declares variables declared in a *let* or *free* expression. If there is a *case* expression, then a new variable x_e is declared.

The function $\mathcal{A}$ assigns variables in a *let* or *free* expression. The expression for all variables in a *let* is translated with $\mathcal{E}$. Next, if there are any variables declared in the *let* block that are used in one of the expressions, they need to be filled in. Finally, if there is a case expression, we assign x_e to be the root of the scrutinee.

A compiler from FlatCurry to ICurry implementing these translation rules is available as package `icurry`. It can easily be installed with the Curry package manager.[6] The tool provided by this package also contains an interpreter for ICurry, based on the small-step semantics specified in Sect. 5.2, which can visualize the graph and machine state during a computation.

5.4 ICurry Use

The stated goal of ICurry is to be a format of Curry programs suitable for translation into an imperative language. Below, we briefly report our experience in translating ICurry into various target languages. Table 1 shows the size of a Curry program that translates ICurry into a target language. The numerical values in the table, extracted from Wittorf's thesis [30], count the lines of code of the translator. The table is only indicative since "lines of code" is not an accurate measure, and some earlier compilers use older variants of ICurry that have

[6] `http://curry-lang.org/tools/cpm`

evolved over time. Each ICurry construct has a direct translation into the tar-

C	441
Python	342
Java	790
JavaScript	632

Table 1. Number of Curry source lines of code for various translators from ICurry to a target language.

get language. The following details refer to the translation into C. Declarations and assignments are the same as in C. The ICurry statements are translated as follows: (1) the ICurry *return* is the same as in C, (2) an ICurry *case statement* is translated into a C *switch statement* where the case selector is the tag of a node, and (3) the ICurry *exempt* statement is translated into code that, when executed, terminates the executing computation without producing any result. This is justified by the facts that the evaluation strategy executes only needed steps, and that failures in non-deterministic programs are natural and expected, therefore they should be silently ignored.

The ICurry case expressions of a function's code contain a branch for each constructor in the argument's type and a branch for each of the following: the choice symbol, the failure symbol, any function symbol [11, Fig. 2], and any free variable. A dispatch table, which is addressed by the argument's label's tag, efficiently selects the branch to be executed. The behavior of the additional branches is described below, and is the same across all the functions of a program. A choice symbol in a pattern matched position results in the execution of a pull-tabbing step [5, 13]. A failure is propagated to the context. A function symbol triggers the evaluation of the expression rooted by this symbol. Finally, a free variable is instantiated to a choice of *shallow* patterns of the same type as the variable. As an example, the evaluation of:

```
head x where x free
```

instantiates `x` to `[] ? (y:ys)` where `y` and `ys` are free variables. The alternative `[]` will result in failure. This can be determined at compile time and removed during optimizations.

Compilers from Curry to Python and other imperative languages can be implemented as described above. As Table 1 indicates, the compilers (written in Curry) are quite compact. We observe that FlatCurry covers the complete language, since it is the basis for robust Curry implementations, like PAKCS and KiCS2, and the natural/operational semantics of Curry is defined in FlatCurry [1]. ICurry contains the same information as FlatCurry except type information, since the type correctness of a program has been verified at the point of the compilation process in which ICurry is used.

We have also implemented a translator from ICurry programs into the JSON format. This translator is simpler and shorter than all the translators into imperative languages of Table 1. The translation into JSON is used by Sprite [11], a Curry system under development, whose target language is C++. The JSON format is more convenient than ICurry when the client of the ICurry format is not coded in Curry, hence it cannot read and parse ICurry program using Curry's library functions.

6 Concluding Remarks

Our work is centered on the compilation of Curry programs. As in many compilers, our approach is transformational. To compile a Curry program P, we translate P into a language, called *target*, for which a compiler already exists. This is the same route followed by other Curry compilers like PAKCS [21] and KiCS2 [12].

PAKCS translates source Curry code into Prolog, leveraging the existence of native free variables and non-determinism in Prolog. KiCS2 translates source Curry code into Haskell, leveraging the existence of first-class functions and their efficient demand-driven execution in Haskell. Both of these compilers use FlatCurry as an intermediate language. They have the same front end which translates Curry into FlatCurry. The use of FlatCurry simplifies the translation process, but is still appropriate to express Curry computations without much effort. FlatCurry has some relatively high-level constructs that can be mapped directly into Prolog and Haskell, because these languages are high-level, too.

In order to provide a better basis to compile Curry into low-level imperative languages, we presented ICurry as an intermediate language for this purpose. Before ICurry, a Curry compiler targeting a C-like language would handle certain high-level constructs of FlatCurry in whichever way each programmer would choose. This led to both duplications of code and unnecessary differences. ICurry originates from these efforts. It abstracts the ideas that, over time, proved to be simple and effective in a language-independent way. With ICurry, the effort to produce a Curry compiler targeting an imperative language is both shortened, because more of the front end can be reused, and simplified, because the starting point of the translation is independent of the target and is well understood.

Our work is complementary to, but independent of, other efforts toward the compilation of Curry programs. These efforts include the development of evaluation strategies [6], or the handling of non-determinism [5, 13].

There exist other functional logic languages, e.g., $\mathcal{TOY}$ [15, 25] whose operational semantics can be abstracted by needed narrowing steps of a constructor-based graph rewriting system. Some of our ideas seem applicable with little to no changes to the implementation of these languages.

Graph rewriting, often supported by graph machines [14, 24, 26], has been used for the implementation of functional languages. A comparison with these efforts is problematic at best. Despite the remarkable syntactic similarities—Curry's syntax extends Haskell's with a single construct, a free variable declaration—the semantic differences are profound. In particular, there is no textual order among the rewrite rules of a functional logic program, and the notion of laziness is based on needed steps modulo non-deterministic choices. As a consequence, there are purely functional programs whose execution produces a result as Curry but does not terminate as Haskell [8, Sect. 3]. Furthermore, most steps of a functional logic computation are functional steps, but the computation must be prepared to encounter non-determinism and/or free variables. Hence, situations and goals significantly differ.

Future work should investigate ICurry to ICurry transformations that are likely to optimize the generated code. For example, different orders of the declaration of variables in a let block lead to different numbers of assignments. Also, case expressions as arguments of function call can be moved outside the call in some situations rather than be replaced by a call to a new function.

Acknowledgments The authors are grateful to the anonymous reviewers for their helpful comments to improve the paper. This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

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. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
4. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
5. S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.
6. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
7. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
8. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
9. S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.
10. S. Antoy, M. Hanus, and S. Libby. Proving non-deterministic computations in Agda. In *Proc. of the 24th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2016)*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 180–195. Open Publishing Association, 2017.
11. S. Antoy and A. Jost. A new functional-logic compiler for Curry: Sprite. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, pages 97–113. Springer LNCS 10184, 2016.

12. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
13. B. Brassel and F. Huch. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*, pages 122–138, Berlin, Heidelberg, 2007. Springer-Verlag.
14. G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 244–258. ACM, 1988.
15. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.
16. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research report IMAG 985-I, IMAG-LSR, CNRS, Grenoble, 1997.
17. M. Hanus. FlatCurry: An intermediate representation for Curry programs, 2008. Available at `http://www.informatik.uni-kiel.de/~curry/flat/`.
18. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
19. M. Hanus. Combining static and dynamic contract checking for Curry. In *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*, pages 323–340. Spriner LNCS 10855, 2017.
20. M. Hanus. Verifying fail-free declarative programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming(PPDP 2018)*, pages 12:1–12:13. ACM Press, 2018.
21. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2018.
22. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
23. M. Hanus and F. Teegen. Adding `Data` to Curry. In *Proceedings of the Conference on Declarative Programming (Declare 2019)*. Springer LNCS, 2019.
24. R. Kieburtz. The G-machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture*, volume LNCS 201, pages 400–413. Springer, 1985.
25. F. J. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
26. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463, 1990.
27. M. J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.
28. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 3–61. World Scientific, 1999.
29. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Int. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
30. M.A. Wittorf. Generic translation of Curry programs into imperative programs (in German). Master's thesis, Kiel University, 2018.