UPV-Curry: An Incremental Curry Interpreter*

M. Alpuente S. Escobar S. Lucas

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46022 Valencia, Spain.
{alpuente,sescobar,slucas}@dsic.upv.es

Abstract. Functional logic programming integrates the best features of modern functional and logic languages. The multi-paradigm declarative language Curry is an extension of Haskell which is intended to become a standard in the area. In this paper, we present UPV-Curry, an efficient and quite complete implementation of Curry based on a new, incremental definition of its basic evaluation mechanism. We compare UPV-Curry with already existing implementations of other Curry interpreters.

Keywords: Curry, Functional logic languages, Implementation.

1 Introduction

Functional logic languages combine the best features of the most important declarative programming paradigms, namely functional and logic programming (see [Han94] for a survey). Throughout this decade, many practical proposals have been made to amalgamate functional and logic programming languages. However, these languages have not succeeded in becoming widely used by the functional or logic programming communities. The multi-paradigm language Curry [HKMN95,HAK+99] is an extension of Haskell [HPW92] which is supported by an international initiative to make it a standard in the area. In order to facilitate and extend the use of Curry, it is essential to make efficient and practical implementations available. Program transformation techniques whose goal is to derive better semantically equivalent programs have recently been adapted to their use in Curry [AAHV99,AFMV99,AHLV99]. In this paper, we consider a different approach for improving the execution of Curry programs, which relies on an incremental definition of the Curry operational machinery. We present UPV-Curry, a novel implementation of Curry which provides an almost complete implementation of the language. In Curry, each evaluation step for an expression e is performed on a *needed* redex of e, i.e., a subexpression of e which is an instance of a left-hand side l of a (possibly conditional) program equation $l \mid c = r$ and which is really necessary for a complete evaluation of e. As is usual in functional logic languages, it is allowed to adequately instantiate variable occurrences of e in order to reduce this redex. The overall process is called *needed*

* This work has been partially supported by CICYT TIC 98-0445-C03-01 and Acción Integrada hispano–alemana HA1997-0073.

narrowing [AEH94]. To implement UPV-Curry, we have developed an *incremental* optimization of the basic operational machinery of Curry which makes the pattern matching between program equations and goal expressions faster and is also able to *locally* resume the search for a new redex after performing each single evaluation step. We provide an experimental comparison of our implementation and other existing Curry interpreters.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [BN98] and functional logic programming [Han94]. We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{X}$ are denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of one variable. $root(t)$ denotes the symbol at the root of the term t. If $root(t) \in \mathcal{F}$ ($root(t) \in \mathcal{C}$), the term t is said to be operation-rooted (*constructor-rooted*). A *pattern* is a linear term of the form $f(\overline{d_n})$ where $f \in \mathcal{F}$ is n-ary and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

A *position* p in a term t is a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists p'.(p.p' = q)$. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing* $t|_p$ by the term s (see [BN98] for details). $\mathcal{P}os(t)$ ($\mathcal{FP}os(t)$) is the set of positions (of operation-rooted subterms) of t.

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The identity substitution is denoted by id. Substitutions are extended to morphisms on terms by $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ for every term $f(\overline{t_n})$. Given terms t, s, we write $t \leq s$ if $\exists \sigma. s = \sigma(t)$; a *unifier* of s and t is a substitution σ with $\sigma(s) = \sigma(t)$.

3 Curry

Curry is a functional logic programming language that combines the best ideas of declarative languages such as Haskell [HPW92] and SML [MTH90] (functional languages), Gödel [HL94] and λProlog [NM88] (logic languages), and Babel [KA96] and $\mathcal{TOY}$ [LS99] (functional logic languages). More specifically, Curry includes higher-order features, a type system, a module system, modern evaluation strategies, non-determinism, (encapsulated) search, partial data structures, existential variables, constraints, and declarative I/O.

We give a schematic description of the syntax of Curry. More details can be found in [HAK+99]. We first show a simple example of a Curry program.

Example 1. The following Curry program:

```
data Nat = Z | S Nat
plus :: Nat -> Nat
```

```
plus Z     x = x            plus (S x) y = S (plus x y)
```

contains the declaration of the *data type* `Nat`, an (optional) *type declaration* of the function `plus`, and the set of equations describing the function `plus`.

User defined data types are collections of *values* which are terms built from the data constructors which are associated to the data type being considered. For instance, `Z` and `S` are the data constructors for the data type `Nat`, and `Z`, `S Z`, and `S (S Z)` are examples of values of `Nat`. A Curry program mainly consists of a collection of data type declarations and function definitions given by (conditional) equations $f\ d_1\ \cdots\ d_n\ \{\texttt{|}\ c\}\ \texttt{=}\ r$ where f is the function symbol which the equation (partially) defines, $d_1, \ldots, d_n$ are constructor terms with variables, c is an (optional) condition, and r is the result expression. The condition c can either be a boolean expression (in the usual sense of functional languages) or a constraint expression (i.e., a conjunction of equational constraints e_1`=:=`e_2 where both e_1 and e_2 are expressions). An expression has the form:

$e ::=$	x	% variable
	$\mid c\ e_1\ \cdots\ e_k$	% application of n-ary constructor c $(0 \leq k \leq n)$
	$\mid f\ e_1\ \cdots\ e_k$	% application of n-ary function f $(0 \leq k \leq n)$
	$\mid$ `if` b `then` e_1 `else` e_2	% conditional expression
	$\mid$ `\` *pattern* `->` e	% a lambda abstraction
	$\mid e_1\ op\ e_2$	% an infix operator application

3.1 Operational Semantics of Curry

The operational principle of Curry uses definitional trees. Given a program $\mathcal{R}$, a *definitional tree* $\mathcal{T}$ with pattern π (notation, $pattern(\mathcal{T}) = \pi$) is an expression of the form[1] [AHLV99]:

$\mathcal{T} = rule(\pi = r')$ where $\pi = r'$ is a variant of a program equation $l = r \in \mathcal{R}$.
$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ where $\pi|_o \in \mathcal{X}$, $c_1, \ldots, c_n$ are different constructors for $n > 0$, and each $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of c_i and $x_1, \ldots, x_k$ are new variables.

A defined symbol f is called *inductively sequential* if there exists a definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_k)$ whose rule nodes contain all (and only) the program equations defining f. In this case, we say that $\mathcal{T}$ is a definitional tree of f. A program $\mathcal{R}$ is *inductively sequential* if all its defined function symbols are inductively sequential. An inductively sequential program can be viewed as a set of definitional trees, each defining a function symbol.

Example 2. Given the following function definition:

```
first Z     y      = []
first (S n) (x:xs) = x : (first n xs)
```

[1] Due to lack of space, we ignore the *or* nodes as well as the residuation of function calls used in Curry [HAK+99]. Of course, they have been considered in our implementation of the UPV-Curry interpreter.

The associated definitional tree is:

$branch($`first x y`$, 1,$
    $rule($`first Z y = []`$),$
    $branch($`first (S n) y`$, 2, rule($`first (S n) (m:ms) = m:(first n ms)`$)))$

Figure 1 shows the graphical representation of this definitional tree.

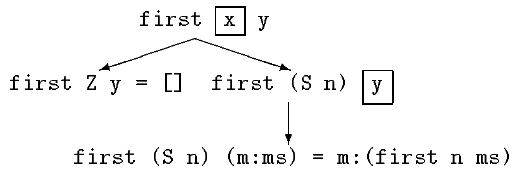

Fig. 1. Definitional tree for the function `first`

A function λ from terms and definitional trees to sets of triples (p, R, σ) determines each evaluation step. Here, p is a position in t, R is the rule to be applied, and σ is a substitution which should be applied to t before reducing it. Given a term t and a definitional tree $\mathcal{T}$ such that $pattern(\mathcal{T}) \leq t$ [AHLV99]:

$$\lambda(t, \mathcal{T}) \ni \begin{cases} (\Lambda, l \to r, id) & \text{if } \mathcal{T} = rule(l = r); \\ (p, l \to r, \sigma \circ \tau) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k),\ t|_o = x \in \mathcal{X}, \\ & \quad \tau = \{x \mapsto c_i(\overline{x_n})\},\ pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_n})]_o \text{ and} \\ & \quad (p, l \to r, \sigma) \in \lambda(\tau(t), \mathcal{T}_i); \\ (p, l \to r, \sigma) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k),\ t|_o = c_i(\overline{t_n}), \\ & \quad pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_n})]_o, \text{ and } (p, l \to r, \sigma) \in \lambda(t, \mathcal{T}_i); \\ (o.p, l \to r, \sigma) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k),\ t|_o = f(\overline{t_n}) \text{ for } f \in \mathcal{F}, \\ & \quad \mathcal{T}' \text{ is a definitional tree for } f \text{ and } (p, l \to r, \sigma) \in \lambda(t|_o, \mathcal{T}') \end{cases}$$

If $(p, l \to r, \sigma) \in \lambda(t, \mathcal{T})$, then $t \leadsto_{(p, l \to r, \sigma)} s$ is a valid *narrowing* step, i.e., $l \leq \sigma(t|_p)$ and $s = \sigma(t[r]_p)$, which is called a *needed* narrowing step.

4 UPV-Curry

UPV-Curry is a novel interpreter of Curry which is publicly available from the URL: `http://www.dsic.upv.es/users/elp/soft.html`. UPV-Curry provides an almost complete implementation of Curry according to [HAK$^+$99] (it lacks Curry modules and encapsulated search). The working environment of UPV-Curry provides the following facilities [EAL98]:

- a small, self-contained Curry implementation, which constitutes a portable stand-alone SICStus Prolog application.
- a read-eval-point loop for displaying the solution for each expression which is entered as an input to the interpreter.
- simple browsing commands, which are able to obtain the type and the definitional tree of each function (see below).
- a debugging command, which permits tracing the evaluation of expressions.

4.1 Incremental defininitional trees

The implementation of UPV-Curry is based on a simpler representation of definitional trees. Given a program $\mathcal{R}$, an *incremental definitional tree* $\mathcal{I}$ with pattern π is an expression of the form:

$\mathcal{I} = irule(\pi = r')$ where $\pi = r'$ is a variant of a program equation $l = r \in \mathcal{R}$.
$\mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n))$ where $\pi|_o \in \mathcal{X}$, $c_1, \ldots, c_n$ are constructors for $n > 0$, and each $\mathcal{I}_i$ is an incremental definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of c_i, and $x_1, \ldots, x_k$ are new variables.

The pattern of an incremental definitional tree can be obtained as follows:

$$pattern(\mathcal{I}) = \begin{cases} \pi & \text{if } \mathcal{I} = irule(\pi = r); \\ pattern(\mathcal{I}_1)[x]_o & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)) \text{ and} \\ & x \notin \mathcal{V}ar(pattern(\mathcal{I}_1)). \end{cases}$$

Standard and incremental definitional trees are related by a function ρ:

$$\rho(\mathcal{T}) = \begin{cases} irule(l = r) & \text{if } \mathcal{T} = rule(l = r); \\ ibranch(o, (c_1, \rho(\mathcal{T}_1)), \ldots, (c_k, \rho(\mathcal{T}_k))) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \text{ and} \\ & pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_n})]_o,\ 1 \leq i \leq k \end{cases}$$

Example 3. The definitional tree of Example 2 is represented as follows:

$ibranch(1, ($`Z`$, irule($`first Z y = []`$)),$
$\quad ($`S`$, ibranch(2, ($`:`$, irule($`first (S n) (m:ms) = m:(first n ms)`$))))$

Figure 2 shows the corresponding graphical representation.

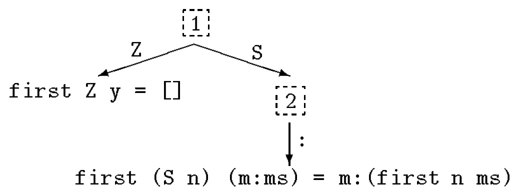

Fig. 2. Incremental definitional tree for the function `first`

4.2 Incremental evaluation

Let $e_1 \leadsto_{(p_1, R_1, \sigma_1)} \cdots e_i \leadsto_{(p_i, R_i, \sigma_i)} e_{i+1} \cdots e_n$ be a needed narrowing sequence, where $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$ and $\mathcal{T}_i$ is a definitional tree for $root(e_i)$, $i = 1, \cdots, n$. In the case when $e_{i+1}|_{p_i}$ is operation-rooted, we have that $p_{i+1} \geq p_i$, hence we take $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (p_i.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_{p_i}, \mathcal{T})$ and $\mathcal{T}$ is a definitional tree for $root(e_{i+1}|_{p_i})$. If $e_{i+1}|_{p_i}$ is constructor-rooted or a variable, then it is necessary to resume the evaluation someplace above p_i, hence we go back to $e_{i+1}|_q$ (where $q < p_i$ is the position of the defined symbol immediately above p_i) by taking the ('most defined') definitional subtree $\mathcal{T}_i'$ of the definitional tree of $root(e_i|_q)$ which has been used in the computation of $\lambda(e_i, \mathcal{T}_i)$ immediately before obtaining $\mathcal{T}_i$. Thus, we let $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (q.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_q, \mathcal{T}'_i)$. This completes an incremental definition of λ.

We use a list $\mathcal{L}$ which contains the information needed to perform the needed narrowing steps incrementally; $\mathcal{L}$ is a list of pairs $(p, \mathcal{I})$ where p is a position and $\mathcal{I}$ is an incremental definitional tree. The *incremental needed narrowing strategy* is denoted by λ^ι. Given an operation-rooted term t and a list $\mathcal{L} = [(q, \mathcal{I}), \ldots]$ with $pattern(\mathcal{I}) \leq t|_q$, we compute $(p, R, \sigma, \mathcal{L}') \in \lambda^\iota(t, \mathcal{L})$, where (p, R, σ) describes a needed narrowing step and $\mathcal{L}'$ is a list containing the information which allows us to proceed with the subsequent evaluation steps incrementally. Formally,

$$\lambda^\iota(t, [(q,\mathcal{I})|\mathcal{L}]) \ni \begin{cases} (q, l \to r, id, \mathcal{L}') & \text{if } \mathcal{I} = irule(l = r) \text{ and} \\ & \quad \mathcal{L}' = \begin{cases} [(q,\mathcal{I}')|\mathcal{L}] & \text{if } \theta(r) \text{ is operation-rooted} \\ \mathcal{L} & \text{otherwise} \end{cases} \\ & \quad \text{where } t|_q = \theta(l) \text{ and } \mathcal{I}' \text{ is an incremental} \\ & \qquad \text{definitional tree for } root(\theta(r)) \\ (p, l \to r, \sigma \circ \tau, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = x \in \mathcal{X},\ \tau = \{x \mapsto c_i(\overline{x_k})\}, \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(\tau(t), [(q, \mathcal{I}_i)|\mathcal{L}]); \\ (p, l \to r, \sigma, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = c_i(\overline{t_k}), \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(t, [(q, \mathcal{I}_i)|\mathcal{L}]); \\ (p, l \to r, \sigma, \mathcal{L}') & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)), \\ & \quad t|_{q.o} = f(\overline{t_k}) \text{ for } f \in \mathcal{F}, \\ & \quad \mathcal{I}' \text{ is an incremental definitional tree for } f, \\ & \quad \text{and } (p, l \to r, \sigma, \mathcal{L}') \in \lambda^\iota(t, [(q.o, \mathcal{I}'), (q, \mathcal{I})|\mathcal{L}]) \end{cases}$$

The needed narrowing steps performed by λ^ι and the computation steps carried out by the original needed narrowing strategy λ are equivalent.

Theorem 1. *Let $\mathcal{R}$ be an inductively sequential program, $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ be a needed narrowing derivation where $\mathcal{T}_i$ is a definitional tree such that $pattern(\mathcal{T}_i) \leq e_i$ for all $1 \leq i \leq n$, and $\mathcal{L}_0 = [(\Lambda, \rho(\mathcal{T}_1))]$. Then, for all $1 \leq i \leq n$, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$ if and only if $(p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, \mathcal{L}_{i-1})$.*

5 Experimental results

In this section, we compare UPV-Curry and other Curry interpreters. Table 1 provides the runtimes of the benchmarks for UPV-Curry, TasteCurry, and PACS-TasteCurry interpreters[2]. Times were measured on a SUN SparcStation, running under UNIX System V Release 4.0. They are expressed in milliseconds and are the average of 10 executions. Most of the benchmarks used for the analysis

[2] TasteCurry and PACS-TasteCurry are available from `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/`.

Benchmark	Goal	UPV-Curry	TasteCurry	PACS-TasteCurry
`iter`	`iter 100 sub1 100`	323	1174	909
`ackermann`	`ackermann 20`	640	2505	1203
`mergesort`	`sort (intMerge) [3,1,2] xs`	313	870	442
`quicksort`	`qsort` [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]	2476	1829	1511
`fibonacci`	`fibonacci 10`	576	2262	1106
`horseman`	`horseman x y 8 20`	2655	3542	1658
`last`	`last` $[1]_{100}$	990	58736	4256

Table 1. Runtime goals and comparison with TasteCurry (in ms.)

(namely, `quicksort`, `last`, `horseman`, and `mergesort`) are standard Curry test programs[3]. The benchmarks `ackermann`, `fibonacci`, and `iter` are given in Table 2. The functions `ackermann`, `fibonacci`, `last`, `quicksort` and `mergesort`, are well-known; `horseman` computes the number of men and horses that have a certain number of heads and feet; finally, `iter` produces a sequence of n nested calls to a given function. Natural numbers are implemented using `Z`/`S`-terms, and lists are shown in goals using a subindex which represents its size. The figures in Table 1 show that our UPV-Curry implementation performs very well in comparison to the other Curry interpreters. In overall, it takes about 80% the time needed by PACS-TasteCurry and about 38% the time needed by TasteCurry to evaluate the queries. These results seem to substantiate the advantages of using the proposed incremental techniques. A more detailed comparison among Curry implementations -including also more benchmark examples- is underway.

ackermann	

`ackermann n = ack (S (S Z)) n`	`ack Z n = S n`
	`ack (S m) Z = ack m (S Z)`
	`ack (S m) (S n) = ack m (ack (S m) n)`

fibonacci	

`sum Z y = y`
`sum (S x) y = S (sum x y)`
	`fib Z = S Z`
	`fib (S (S x)) = sum (fib (S x)) (fib x)`
	`fib (S x) = S x`

iter	

`iter Z f x = x`
`iter (S y) f x = f (iter y f x)`
	`sub1 (S Z) = Z`
	`sub1 (S (S Z)) = S Z`
	`sub1 (S (S (S Z))) = S (S Z)`
	`sub1 (S (S (S (S x)))) = S (S (S x))`

Table 2. Benchmark code

6 Conclusions

We have presented UPV-Curry, a novel and effective implementation of Curry which provides an almost complete implementation of the language. In order to

[3] Look at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/examples/` for a collection of examples containing these and other available Curry test programs.

improve performance, we have implemented (and proved correct) an *incremental* (re)definition of the basic evaluation strategy of Curry which is able to *locally* resume the search for a new redex after performing a previous incremental narrowing step, and also reduces the number of term positions to be considered in a needed narrowing derivation. Finally, we have provided the (first) experimental comparison of different implementations of existing Curry interpreters.

References

[AAHV99] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of LPAR'99*.

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. of POPL'94*, pages 268–279, 1994.

[AFMV99] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs In *Proc. of FLOPS'99*.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of ICFP'99*.

[BN98] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.

[EAL98] S. Escobar, M. Alpuente, and S. Lucas. UPV-Curry User's Manual. TR DSIC-II/38/98. `http://www.dsic.upv.es/users/elp/papers.html`.

[HAK+99] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, and F. Steiner. Curry An Integrated Functional Logic Language (version 0.5). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, Jan 1999.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[HKMN95] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[HL94] P. Hill and J.W. Lloyd. *The Gödel Programming Language.* The MIT Press, Cambridge, MA, 1994.

[HPW92] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27(5), 1992.

[KA96] H. Kuchen and J. Anastasiadis. Higher Order Babel: Language and implementation. In *Proc. of ELP'96*, LNAI 1050:193–207, 1996.

[LS99] F.J. López Fraguas and J. Sánchez Hernández. $\mathcal{TOY}$: A multiparadigm Declarative System. In *Proc. of RTA'99*, LNCS 1631, 1999.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of ML.* The MIT Press, 1990.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In *Proc. of ICLP'88*, pages 810–827. The MIT Press, Cambridge, MA, 1988.