

# On a Tighter Integration of Functional and Logic Programming\*

Bernd Braßel and Frank Huch

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

**Abstract.** The integration of functional and logic programming is a well developed field of research. We discuss that the integration could be improved significantly in two separate aspects: sharing computations across non-deterministic branching and the declarative encapsulation of logic search. We then present a formal approach which shows how these improvements can be realized and prove the soundness of our approach.

## 1 Introduction

There are two main streams in declarative programming: functional and logic programming. For several years now a third stream aims at combining the key advantages of these two paradigms. This third stream is often called “functional logic programming” but could also be simply denoted as “declarative programming”. Declarative programming languages try to bridge the chasm between the deterministic world of (lazy) functional programming and the non-deterministic multiverse of logic programming. By now the theory is well developed and among the many approaches we name only three works we consider the main theoretic fundament: A denotational semantics was developed in [9] and extended in many subsequent publications. A formal base on narrowing was given in [3] and an operational semantics was introduced in [1].

There are, however, reasons to believe that the integration of both paradigms could be tighter, demonstrated by two examples in the functional logic language Curry:

*Example 1 (Sharing across Non-Determinism).* We consider parser combinators which can elegantly make use of the non-determinism of functional logic languages to implement the different rules of a grammar. A simple set of parser combinators, consisting of the always successful parser `succ`, a parser for a single character `sym` and the sequential composition of two parsers (`<*>`), can be defined as follows:

```
type Parser a = String -> (String,a)

succ :: a -> Parser a
succ r cs = (cs,r)
```

---

\* This work has been partially supported by DFG grant Ha 2457/5-1.

```

sym :: Char -> Parser Char
sym c (c':cs) | c==c' = (cs,c)

(<*>) :: Parser (a -> b) -> Parser a -> Parser b
(p1 <*> p2) str = case p1 str of
  (str1,f) -> case p2 str1 of
    (str2,x) -> (str2,f x)

parse :: Parser a -> String -> a
parse p str = case p str of ("",r) -> r

```

As an example, we construct a non-deterministic parser for the inherent ambiguous language of palindromes without marked center  $L = \{w\overleftarrow{w} \mid w \in \{a,b\}^*\}$ <sup>1</sup>, which, if parsing is possible, returns the word  $w$  and fails otherwise:

```

pal :: Parser String
pal = succ (\ c str _ -> c:str) <*> sym 'a' <*> pal <*> sym 'a'
  ? succ (\ c str _ -> c:str) <*> sym 'b' <*> pal <*> sym 'b'
  ? succ ""

```

where  $(?) :: a \rightarrow a \rightarrow a$  induces non-deterministic branching and is defined by:

```

x ? _ = x
_ ? y = y

```

In all Curry implementations the parser `pal` analyses a `String` of length 100 within milliseconds. We call this time  $t_{parse}$ . Unfortunately, this program does not scale well with respect to the time it takes to compute the argument string, which we consider to be a list of expressions  $[e_1, \dots, e_{100}]$  where each  $e_i$  evaluates to a character taking an amount of time  $t \gg t_{parse}$  and constructing the string  $[e_1, \dots, e_{100}]$  takes time  $100 \cdot t$ . Then one would expect the total time to compute `parse pal [e1, ..., e100]` is  $100 \cdot t + t_{parse} \approx 100 \cdot t$ . But measurements for the Curry implementation PAKCS ([10]) show that, e.g., for  $t = 0.131s$  it takes more than  $5000 \cdot t$  to generate a solution for a palindrome  $[e_1, \dots, e_{100}]$  and  $9910 \cdot t$  to perform the whole search for all solutions. We obtain similar results for other implementations of lazy functional logic languages, like the Münster Curry Compiler [15].

The reason is that these systems do not provide sharing across non-determinism. For all non-deterministic branches in the parser the elements of the remaining list are computed again and again. Only values which are evaluated before non-determinism occurs are shared over the non-deterministic branches. This behavior is not only a minor implementation detail but poses problems concerning the fundamentals of declarative languages. These problems have never been treated at a formal level before.

The consequence of Example 1 is that programmers have to avoid using non-determinism for functions that might be applied to expensive computations. But in connection with *laziness* a programmer cannot know which arguments are already computed because evaluations are suspended until their value is demanded. Hence, the connection of laziness with logic search always threatens to perform with the considerable slowdown discussed above. Thus, sadly, when

<sup>1</sup> We restrict to this small alphabet for simplicity.

looking for possibilities to improve efficiency, the programmer in a lazy functional logic language is well advised to either try to eliminate the logic features he might have employed or to strictly evaluate expressions prior to each search. Both alternatives show that he still follows either the functional or the logic paradigm but cannot profit from a seamless integration of both.

A second important topic when connecting logic search with the lazy functional paradigm is *encapsulated search*. Encapsulated search is employed whenever different values of one expression have to be related in some way, e.g., to compute a list of all values or to find the minimal value and also to formulate a search strategy. But again we find that the connection of this feature with laziness is problematic as illustrated by the following example.

*Example 2 (Encapsulated Search)*. Reconsider the notion of palindrome and the parser definitions of Example 1. We call a palindrome *prime*, if it does not contain a proper, non-empty prefix which is also a palindrome. For a given string  $s$  we can check this property by inspecting the result of applying our parser.  $s$  is a prime palindrome if (`pal s`) meets two conditions: 1)  $s$  is a palindrome, i.e., the parser succeeds consuming the whole input and 2) there are only two successful parses as the empty prefix is a palindrome by definition. Hence, we can use the operation `allValues`, which performs encapsulated search to yield a list containing all values of its argument, to count the number of successful parses and we define:

```
prime : (String,a) -> Success
prime r = fst r :=> "" & length (allValues r) :=> 2
```

To express the conditions, we use the strict equality operator (`:=>`) which implements unification. The two conditions are expressed as constraints on the parse results connected by the operator (`&`). (`&`) is called “concurrent conjunction” and is a primitive, i.e., an externally defined operator. The adjective “concurrent” suggests that the result should not depend on the order in which the constraints are evaluated. But if `allValues` is based on encapsulated search as available in PAKCS, the result of, e.g., `prime (pal "abba")` does indeed depend on the order of evaluation. If the constraint `fst r :=> ""` is solved first then PAKCS yields no solution and if the second constraint is preferred, the computation is successful. We will not explain how this behavior comes to pass and refer to [5] for a detailed discussion. Here, it is only important that this problem also stems from the connection of laziness with logic search and is caused by the sharing of `r` in both constraints.

In an alternative approach as implemented in the Münster Curry Compiler (MCC) [14] the result does not depend on the order in which the two constraints are evaluated. The evaluation of `(prime (pal "abba"))` fails in any case, again cf. [5] for details. Although the approach of [14] does not require any knowledge about the order of evaluation, detailed knowledge about the compiler and the executed optimizations are needed to successfully employ encapsulated search in the MCC. For instance, a program can yield different values if one writes `(\x -> x:=>(0 ? 1))` instead of `(:=>(0 ? 1))` although by the very definition of the language Curry [11] the latter expression is just an abbreviation of the former.

Example 2 shows that encapsulated search is a second area on which the integration of functional and logic programming could be tighter. The work pre-

$P ::= D_1 \dots D_m$	
$D ::= f(x_1, \dots, x_n) = e$	
$e ::= x$	(variable)
$c(x_1, \dots, x_n)$	(constructor call)
$f(x_1, \dots, x_n)$	(function call)
$\text{case } e \text{ of } \{p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n\}$	(rigid case)
$\text{fcase } e \text{ of } \{p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n\}$	(flexible case)
$e_1 \text{ or } e_2$	(disjunction)
$\text{let } x_1 = e_1, \dots, x_n = e_n \text{ in } e$	(let binding)
$p ::= c(x_1, \dots, x_n)$	

where  $P$  denotes a program,  $D$  a function definition,  $p$  a pattern and  $e \in \text{Expr}$  an arbitrary expression.

**Fig. 1.** Syntax for Normalized Flat Curry Programs

sented in this paper does not suffer from the illustrated problems: The time to parse a string which is expensive to construct is not multiplied by the number of non-deterministic branches and the evaluation of, e.g., `(prime (pal "abba"))` is successful regardless of the order of evaluation. In general, the use of encapsulated search does neither depend on evaluation order nor on the way the compiler transforms the programs. The approach has been successfully implemented in our Curry to Haskell Compiler, the Kiel Curry System (*KiCS*), available at [www-ps.informatik.uni-kiel.de/~bbr/download/kics\\_src.tgz](http://www-ps.informatik.uni-kiel.de/~bbr/download/kics_src.tgz).

functions of arity 0. In the following sections we show that both problems, sharing across non-determinism and encapsulation, can be interleaved in such a way that a solution to one also solves the other. While we are not the first to present an approach to sharing across non-determinism, cf. Section 5, the seamless integration of both aspects, and especially the purity of encapsulation, is the contribution of the presented work. We propose a natural (big step) semantics for functional logic languages which features sharing across non-determinism. This semantics can be seen as an extension of the one presented in [1] and we formally relate our results to that work. The key idea of the extension is to treat non-deterministic branching as constructors, so called “or-nodes”. This enables sharing beneath these nodes just as beneath any other constructor. As a consequence, however, an or-node is also in head normal form. As the semantics of a given expression in [1] is normally its head normal form, we need a further concept to yield values in the sense of [1]. Surprisingly, this additional concept is encapsulated search. Adding this feature, we can show a strong relationship between the values computed by our semantics and the one of [1].

The paper mainly consists of enhancing the semantics proposed in [1] step-wise. First, the semantics is substantially simplified and then extended to cover sharing across non-determinism and encapsulated search. We only sketch the ideas of the proofs. The complete proofs are presented in [7].

## 2 Preliminaries

For the syntax of functional logic programs we consider the first-order core language *Flat Curry*. Furthermore, we restrict to *normalized Flat Curry*, in which

(VarCons)	$\Gamma[x \mapsto t] : x \Downarrow_0 \Gamma[x \mapsto t] : t$	where $t$ is constructor-rooted
(VarExp)	$\frac{\Gamma \setminus \{(x, \Gamma(x))\} : \Gamma(x) \Downarrow_0 \Delta : v}{\Gamma : x \Downarrow_0 \Delta[x \mapsto v] : v}$	where $\Gamma(x)$ is not constructor-rooted and $\Gamma(x) \neq x$
(Val)	$\Gamma : v \Downarrow_0 \Gamma : v$	where $v$ is constructor-rooted or a variable with $\Gamma(v) = v$
(Fun)	$\frac{\Gamma : \sigma(e) \Downarrow_0 \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow_0 \Delta : v}$	where $f(\overline{y_n}) = e \in P$ and $\sigma = \{\overline{y_n} \mapsto \overline{x_n}\}$
(Let)	$\frac{\Gamma[\overline{y_k} \mapsto \sigma(e_k)] : \sigma(e) \Downarrow_0 \Delta : v}{\Gamma : \text{let } \{\overline{x_k} \equiv \overline{e_k}\} \text{ in } e \Downarrow_0 \Delta : v}$	where $\sigma = \{\overline{x_k} \mapsto \overline{y_k}\}$ and $\overline{y_k}$ are fresh variables
(Or)	$\frac{\Gamma : e_i \Downarrow_0 \Delta : v}{\Gamma : e_1 \text{ or } e_2 \Downarrow_0 \Delta : v}$	where $i \in \{1, 2\}$
(Select)	$\frac{\Gamma : e \Downarrow_0 \Delta : c(\overline{y_n}) \quad \Delta : \sigma(e_i) \Downarrow_0 \Theta : v}{\Gamma : (\text{f})\text{case } e \text{ of } \{\overline{p_k} \rightarrow \overline{e_k}\} \Downarrow_0 \Theta : v}$	where $p_i = c(\overline{x_n})$ and $\sigma = \{\overline{x_n} \mapsto \overline{y_n}\}$
(Guess)	$\frac{\Gamma : e \Downarrow_0 \Delta : x \quad \Delta[x \mapsto \sigma(p_i), \overline{y_n} \mapsto \overline{y_n}] : \sigma(e_i) \Downarrow_0 \Theta : v}{\Gamma : \text{fcase } e \text{ of } \{\overline{p_k} \rightarrow \overline{e_k}\} \Downarrow_0 \Theta : v}$	where $p_i = c(\overline{x_n})$ , $\sigma = \{\overline{x_n} \mapsto \overline{y_n}\}$ , and $\overline{y_n}$ are fresh variables

**Fig. 2.** Natural Semantics for Functional Logic Programs

only variables are allowed as arguments of constructors and functions. The normalization of arbitrary Flat Curry programs is defined in [1]. This normalization is the key idea to express sharing, the main concept of laziness in the functional setting [13] and call-time choice in the functional logic setting. The syntax is presented in Figure 1. Free variables are introduced as circular *let* bindings of the form *let*  $x=x$  *in*  $e$ . To keep programs containing multiple *ors* more readable, we omit brackets for *or* expressions and assume that *or* binds left associatively. We also omit argument brackets for constructors and

The semantics is similarly defined to the semantics in [1], with the exception that a black hole detection (like present in [13]) is added in the rule (VarExp). Without this black hole detection, a non-deterministic choice might produce a proof tree in which the same variable is updated with different values in the heap. [7] presents such an undesired derivation. Hence, this slight modification can be seen as a correction of [1]. We refer to this semantics as  $\Downarrow_0$  and will stepwise modify it. For lack of space, we cannot explain the ideas of the rules in this paper and refer to [1]. Nevertheless, we want to introduce some notations used in the semantics.

**Definition 1 (Heap, Update  $\Gamma[\mapsto]$ ).** Let  $Var = \{x, y, z, \dots\}$  be a finite set of variables,  $Expr$  the set of expressions, as defined in Figure 1, and  $Heap \subset Var \times Expr$  a finite set, called heap, where each element  $x \in Var$  appears at most once in a pair  $(x, e)$  within the set, i.e. a heap represents a partial function from  $Var$  to  $Expr$ . Heaps will be denoted with upper case greek letters (e.g.  $\Gamma, \Delta, \Theta$ ) and we adopt the usual notation for functions to write  $\Gamma(x) = e$  for  $(x, e) \in \Gamma$ . A heap update  $\Gamma[x \mapsto e]$  is an abbreviation for  $(\Gamma \setminus \{(x, \Gamma(x))\}) \cup \{(x, e)\}$ . We

will also make use of the usual notations  $Dom(\Gamma)$  and  $Rng(\Gamma)$  to denote the domain and range of a heap, respectively.

Note that an updated heap is again a heap and that for all heaps  $\Gamma$  the equation  $\Gamma[x \mapsto e][x \mapsto e] = \Gamma[x \mapsto e]$  holds. Like in the rules (Let), (Select), and (Guess) of Figure 2, we often refer to a sequence of arguments, bindings, expressions, or similar objects. We usually write  $\overline{o_n}$  as an abbreviation for a sequence of  $n$  objects  $o_1, \dots, o_n$ . Finally, in example programs, we will often use non-normalized programs and Curry-like notations in which we may use pattern matching instead of case expressions and write function and constructor application in curried form.

### 3 Simplifications of the Semantics

Before we extend the semantics, we apply some further simplifications.

#### 3.1 Elimination of Free Variables

In [4], Antoy and Hanus presented the surprising result that under certain circumstances one can replace free variables by generator functions. Unfortunately, they prove their result only for a term-rewriting based semantics not considering sharing. A similar result was presented in [8] in the context of the *Constructor-based ReWriting Logic (CRWL)*, a different semantic framework for functional logic languages. Unfortunately, this result cannot be transferred to our framework so easily. Since, on the other hand, this technique is crucial for our semantics extensions, we transfer this result to the setting with sharing. To do this we change the setting from  $\Downarrow_0$  as follows and refer to the new semantics as  $\Downarrow_1$ .

1. Replace in each heap and each expression program bindings of the form  $x = x$  by  $x = \text{generate}$ , cf. Definition 2 below.
2. Add to each program the definition of the special function generate:

$$\begin{aligned} \text{generate} &= (\text{let } \{\overline{x_{n_1}} = \text{generate}\} \text{ in } c_1(\overline{x_{n_1}})) \\ &\text{or } \dots \\ &\text{or } (\text{let } \{\overline{x_{n_k}} = \text{generate}\} \text{ in } c_k(\overline{x_{n_k}})) \end{aligned}$$

where  $\overline{c_k}$  are all program constructors and  $c_i$  has arity  $n_i$  for all  $1 \leq i \leq k$ .

Note that normally, the generator is type oriented, i.e., it generates only values of the correct type. This greatly prunes the search space and can be implemented by approaches analogous to those used for type classes.

Now all free variables in a program can be replaced by such generators.

**Definition 2 (Free Variable Eliminations  $e^\dagger$  and  $\Gamma^\otimes$ ).** *We eliminate free variables in expressions by replacing them with a call to the special function generate.*

$$\begin{aligned} x^\dagger &= x \\ s(\overline{x_n})^\dagger &= s(\overline{x_n}) \quad s \text{ function or constructor} \\ (e_1 \text{ or } e_2)^\dagger &= e_1^\dagger \text{ or } e_2^\dagger \\ ((f)\text{case } e \text{ of } \{\overline{p_k \rightarrow e_k}\})^\dagger &= (f)\text{case } e^\dagger \text{ of } \{\overline{p_k \rightarrow e_k^\dagger}\} \\ (\text{let } \{\overline{x_k = e_k}\} \text{ in } e)^\dagger &= \text{let } \{\overline{(x_k = e_k)^\dagger}\} \text{ in } e^\dagger \\ (x = e)^\dagger &= \begin{cases} x = \text{generate} & , \text{ if } e = x \\ x = e^\dagger & , \text{ otherwise} \end{cases} \end{aligned}$$

Likewise, we replace free variables in heaps by defining:

$$\Gamma^\otimes = \{(x, e^\dagger) \mid (x, e) \in \Gamma \wedge x \neq e\} \cup \{(x, \text{generate}) \mid (x, x) \in \Gamma\}$$

We also write  $P^\dagger$  for the result of transforming all expressions in a program  $P$  by means of  $\dagger$ .

Note that both  $e^\dagger$  and  $\Gamma^\otimes$  are unambiguously invertible.

The evaluation of the special function `generate` is a linear proof tree which non-deterministically chooses one of the constructors of the program as a value. In order to be able to refer to such a tree, we define:

**Definition 3 (Generator Tree  $gT$ ).** For an arbitrary heap  $\Gamma$ , a variable  $x$  with  $\Gamma(x) = x$  and an  $n_i$ -ary constructor  $c_i$  the generator tree is defined as:

$$(\Gamma, x)gT(\Theta, c_i(\overline{x_n})) = \frac{\frac{\frac{\Delta^\otimes := \Gamma'^\otimes[\overline{x_{n_i}} \mapsto \text{generate}] : c_i(\overline{x_n}) \Downarrow_1 \Delta^\otimes : c_i(\overline{x_n})}{\Gamma'^\otimes : e_i = \text{let } \{\overline{y_{n_i}} = \text{generate}\} \text{ in } c_i(\overline{y_n}) \Downarrow_1 \Delta^\otimes : c_i(\overline{x_{n_i}})}}{\times_{j=1}^{i-1} \left( \frac{\Gamma'^\otimes : e_{j+1} \text{ or } \dots \text{ or } e_k \Downarrow_1 \Delta^\otimes : c_i(\overline{x_n})}{\Gamma'^\otimes : e_j \text{ or } e_{j+1} \text{ or } \dots \text{ or } e_k \Downarrow_1 \Delta^\otimes : c_i(\overline{x_n})} \right)}{\frac{\Gamma'^\otimes := \Gamma^\otimes \setminus \{(x, \text{generate})\} : \text{generate} \Downarrow_1 \Delta^\otimes : c_i(\overline{x_n})}{\Gamma^\otimes : x \Downarrow_1 \Theta^\otimes := \Delta^\otimes[x \mapsto c_i(\overline{x_n})] : c_i(\overline{x_n})}}$$

where  $e_j := \text{let } \{\overline{x_{n_j}} = \text{generate}\} \text{ in } c_j(\overline{x_{n_j}})$ ,  $j \in \{1, \dots, k\}$  and  $\overline{x_k}$  are all constructors of the program. Note that, by construction,  $\Theta = \Gamma[x \mapsto c_i(\overline{x_n}), \overline{x_n} \mapsto \overline{x_n}]$

Now we are ready to establish the link between  $\Downarrow_0$  and  $\Downarrow_1$ .

**Theorem 1.** Let  $P$  be a program,  $e$  an expression, and  $C$  the set of all constructors used in  $P$ . Then the following properties hold:

$$\begin{aligned} \text{If } \Gamma : e \Downarrow_0 \Delta : c(\overline{x_n}) \quad , \text{ then } \Gamma^\otimes : e^\dagger \Downarrow_1 \Delta^\otimes : c(\overline{x_n}). \\ \text{If } \Gamma : e \Downarrow_0 \Delta : x \quad , \text{ then } \forall c \in C : \Gamma^\otimes : e^\dagger \Downarrow_1 \Delta_1^\otimes : c(\overline{x_n}) \\ \text{and } \Delta_1 = \Delta[x \mapsto c(\overline{x_n}), \overline{x_n} \mapsto \overline{x_n}]. \\ \text{If } \Gamma^\otimes : e^\dagger \Downarrow_1 \Delta_1^\otimes : c(\overline{x_n}), \text{ then } \Gamma : e \Downarrow_0 \Delta_1 : c(\overline{x_n}) \\ \text{or } \Gamma : e \Downarrow_0 \Delta : x \text{ and } \Delta_1 = \Delta[x \mapsto c(\overline{x_n}), \overline{x_n} \mapsto \overline{x_n}]. \end{aligned}$$

*Proof (Central Idea).* Whenever the intermediate result of a sub computation in  $\Downarrow_0$  is a free variable, the corresponding application of **(Val)**  $\Gamma : x \Downarrow_0 \Gamma : x$  is replaced by the generator tree  $gT(\Gamma, x, \Gamma[x \mapsto c(\overline{x_n}), \overline{x_n} \mapsto \overline{x_n}], c(\overline{x_n}))$ . The differences between the resulting heaps is effectively eliminated when the rule **(Guess)** is applied for  $\Downarrow_0$ . The remaining proof is concerned with showing that the mappings  $\otimes$  and  $\dagger$  correctly replace free variables by calls to the generator function.

According to this theorem, we can eliminate the rule **(Guess)**, the distinction between *case* and *fcase* and also all conditions concerned with free variables in the remaining rules. Note, however, that for each free variable used in the proof tree for  $\Downarrow_0$  one generator tree is used in the proof tree for  $\Downarrow_1$ . I.e., variable elimination does not imply loss of efficiency. Furthermore, another simplification is possible, as the next section shows.

### 3.2 Elimination of (VarCons)

After eliminating variables from the semantics, the rule (VarCons) is not needed anymore, because now it is a simple short-cut for applying rules (VarExp) directly followed by (Val). We only have to omit the restrictions, when (Val) and (VarExp) can be applied. We replace (VarExp) by the similar rule (Lookup) and refer to this new semantics without (VarCons) as  $\downarrow$ :

$$\text{(Lookup)} \frac{\Gamma \setminus \{(x, \Gamma(x))\} : \Gamma(x) \downarrow \Delta : v}{\Gamma : x \downarrow \Delta[x \mapsto v] : v}$$

We obtain the following theorem, which can easily be proven by a direct derivation by rule (Lookup) and (Val).

**Theorem 2.**  $\Gamma : e \downarrow_1 \Delta : v$  iff  $\Gamma : e \downarrow \Delta : v$

### 3.3 Summarization of the Simplified Semantics

In the semantics considered so far, it was not necessary to normalize the arguments of *or*. We want to introduce sharing over non-determinism, for which the main idea is to handle *or* as a kind of constructor. Hence, it is necessary to normalize *or* expressions as well and introduce variables by means of a *let* expression for *or*.

**Definition 4 (Stronger Normalization  $e^*$ ).** *Stronger normalization of an expression  $e$  flattens the arguments of “or” by means of the mapping  $e^*$  which is defined inductively as follows:*

$$\begin{aligned} x^* &= x \\ s(\overline{x_n})^* &= s(\overline{x_n}) \\ (e_1 \text{ or } e_2)^* &= \text{let } \{x_1 = e_1^*, x_2 = e_2^*\} \text{ in } (x_1 \text{ or } x_2) \\ ((f)\text{case } e \text{ of } \{p_k \rightarrow e_k\})^* &= (f)\text{case } e^* \text{ of } \{p_k \rightarrow e_k^*\} \\ (\text{let } \{x_k = e_k\} \text{ in } e)^* &= \text{let } \{x_k = e_k^*\} \text{ in } e^* \end{aligned}$$

It is easy to see that the stronger normalization conserves all values of the semantics. With this last simplification we obtain a condensed semantics which we have shown to be equivalent to the one defined in [1]. Since this semantics is the basis for our extensions, we summarize its rules again in Figure 3 and refer to it as  $\downarrow$ .

## 4 Extending the Semantics

### 4.1 Constructors representing Non-Determinism

There have been several attempts to define libraries for logical features for lazy functional programming, e.g. [12]. All these approaches encode the non-deterministic search as a kind of lazily constructed data structure, e.g. a list (embedded in some backtracking monad). The context demands elements from this list (requires their evaluation) which relates to searching solutions within non-deterministic computations in the logical setting.

Our idea is similar in the sense that we employ a data structure representing the values computed in all non-deterministic computations. Since non-deterministic branching may result in an infinite search-space, this data structure

$$\begin{array}{l}
\text{(Lookup)} \quad \frac{\Gamma \setminus \{(x, \Gamma(x))\} : \Gamma(x) \downarrow \Delta : v}{\Gamma : x \downarrow \Delta[x \mapsto v] : v} \\
\text{(Val)} \quad \Gamma : v \downarrow \Gamma : v \quad \text{where } v \text{ is constructor-rooted} \\
\text{(Fun)} \quad \frac{\Gamma : \sigma(e) \downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \downarrow \Delta : v} \quad \text{where } f(\overline{y_n}) = e \in P \\
\quad \quad \quad \text{and } \sigma = \{\overline{y_n} \mapsto \overline{x_n}\} \\
\text{(Let)} \quad \frac{\Gamma[\overline{y_k} \mapsto \sigma(e_k)] : \sigma(e) \downarrow \Delta : v}{\Gamma : \text{let } \{\overline{x_k} = \overline{e_k}\} \text{ in } e \downarrow \Delta : v} \quad \text{where } \sigma = \{\overline{x_k} \mapsto \overline{y_k}\} \\
\quad \quad \quad \text{and } \overline{y_k} \text{ are fresh variables} \\
\text{(Or)} \quad \frac{\Gamma : x_i \downarrow \Delta : v}{\Gamma : x_1 \text{ or } x_2 \downarrow \Delta : v} \quad \text{where } i \in \{1, 2\} \\
\text{(Select)} \quad \frac{\Gamma : e \downarrow \Delta : c(\overline{y_n}) \quad \Delta : \sigma(e_i) \downarrow \Theta : v}{\Gamma : \text{case } e \text{ of } \{\overline{p_k} \rightarrow \overline{e_k}\} \downarrow \Theta : v} \quad \text{where } p_i = c(\overline{x_n}) \\
\quad \quad \quad \text{and } \sigma = \{\overline{x_n} \mapsto \overline{y_n}\}
\end{array}$$

**Fig. 3.** Simplified Natural Semantics for Functional Logic Programs

may be infinite which is no problem in a lazy language. We just need to ensure that it is built only if demanded by the surrounding computation.

But what is an appropriate structure to represent the non-determinism of our semantics? Since it has to reflect the branching of non-determinism, a tree is most appropriate. The nodes in this structure (labeled with **OR**) relate to the evaluation of or expressions. Since we use binary ors in Flat Curry, we obtain a binary tree as well.<sup>2</sup> The leaves in this tree contain either values or failed computations.

*Example 3.* We consider the following Flat Curry program (**Zero**, **One**, and **Two** are constructors):

```

coin = Zero or One
add x y = case x of { Zero -> y;
                    One  -> case y of { Zero -> One;
                                        One  -> Two } }

main = add coin coin

```

Computing the semantics of **coin** yields the tree **OR [Zero,One]**. Now in the computation of **main** the result of **coin** has to be combined twice. We obtain **OR [OR [Zero,One], OR [One,Two]]**. But consider the definition

```

main = let { c = coin } in add c c

```

In a call-time choice semantics, like  $\Downarrow_0$  and  $\downarrow$  as well, it is important that there are only two results for this computation: **Zero** and **Two**. But how can we guarantee call-time choice when we twice combine the **OR** tree representation of **c** in **add**?

We identify each **OR** by a special reference. In the example with sharing, this is **OR  $r_1$  [OR  $r_1$  [Zero,One], OR  $r_1$  [One,Two]]** with the same reference  $r_1$  for every **OR** node. When later the value is presented to the user or consumed by the context we can consider which branching was chosen for which reference and choose exactly the same branching for this reference everywhere within the **OR structure**. Especially, in the example the result **One** is not reachable anymore.

<sup>2</sup> Note however, that we present more general or nodes with a list of branches. This becomes useful when considering the fairness of search strategies, cf. Section 4.3.

## 4.2 Sharing Across Non-Determinism

In the semantics we replace the non-deterministic *or* by introducing an *internal constructor* OR which may not appear in any Flat Curry program. Later, we will introduce another internal constructor FAIL. To distinguish internal constructors from other constructors, we write them with all upper case letters.

(OR)  $\Gamma : x_1 \text{ or } x_2 \Downarrow \Gamma : \text{OR } r [x_1, x_2]$  where  $r$  fresh

(Lift) 
$$\frac{\Gamma : e \Downarrow \Delta : \text{OR } r [\bar{x}_n]}{\Gamma : \text{case } e \text{ of } \{bs\} \Downarrow \Delta[\bar{y}_n \mapsto \text{case } x_n \text{ of } \{bs\}] : \text{OR } r [\bar{y}_n]}$$
 where  $\bar{y}_n$  fresh

By replacing the rule (Or) by the rule (OR), we obtain a deterministic semantics. Beside its children (the variables  $x_1$  and  $x_2$ ), the introduced OR constructor also contains a reference which identifies it in more complex OR structures and which is used to realize call-time choice. The rule (Lift) is used to push computations inside an OR-branching, i.e. lift the OR constructor one level up.

But, how does this modification relate to the original semantics  $\Downarrow$ ? The original semantics  $\Downarrow$  computes the head-normal form of the expression on the control. The same holds for the new semantics  $\Downarrow$ , but now also OR expressions are constructor terms. Hence, the semantics stops whenever the original semantics branches non-deterministically. To retrieve values in the original sense, we have to add a special computation yielding head-normal forms underneath OR nodes for the new semantics. We extend the relation  $\Downarrow$  as follows:

(Hnf-Val) 
$$\frac{\Gamma : e \Downarrow \Delta : c(\bar{x}_n)}{\Gamma : \text{hnf } \rho e \Downarrow \Delta : c(\bar{x}_n)}$$
 where  $c \neq \text{OR}$

(Hnf-Choose) 
$$\frac{\Gamma : e \Downarrow \Delta : \text{OR } r [\bar{x}_n] \quad \Delta : \text{hnf } \rho x_{\rho(r)} \Downarrow \Theta : v}{\Gamma : \text{hnf } \rho e \Downarrow \Theta : v}$$
 if  $r \in \text{Dom}(\rho)$

The *branching information*  $\rho$  is a partial function from OR references to branching positions, i.e. the natural numbers (for binary ors:  $\{1, 2\}$ ). It expresses which branch is to be selected for which OR reference. Later, it will be computed by the surrounding computation. For the moment, it is chosen arbitrarily and corresponds exactly to one non-deterministic computation of  $\Downarrow$ . When the branching information  $\rho$  is given, it is straight forward to prune a heap constructed in a  $\Downarrow$ -proof to a simple heap constructed by the corresponding  $\Downarrow$ -proof. Such a pruning is defined next.

**Definition 5 (Heap Pruning  $\text{cut}(\rho, \Gamma)$ ).**

$$\text{cut}(\rho, \Gamma)(x) = \begin{cases} \text{cut}(\rho, \Gamma)(x_{\rho(r)}), & \text{if } \Gamma(x) = \text{OR } r [\bar{x}_n] \\ \Gamma(x), & \text{otherwise} \end{cases}$$

$\text{cut}$  is not well defined for all possible arguments  $\rho$  and  $\Gamma$ . However, this is not a problem since we only use the definition to prove the existence of a heap with certain properties. Hence,  $\text{cut}$  will only be applied to appropriate arguments.

**Theorem 3 (Completeness of  $\Downarrow$ ).**

If  $\Gamma : e \Downarrow \Delta : v$  then there exist a heap  $\Delta'$  and a sequence  $s$  of branching updates such that  $\Gamma : \text{hnf } \rho e \Downarrow \Delta' : v$  and  $\Delta = \text{cut}(\rho, \Delta')$  where  $\rho := \emptyset s$ .

*Proof (Central Idea).* The main difference between  $\downarrow$  and  $\Downarrow$  is the treatment of expressions ( $x_1$  or  $x_2$ ). In  $\downarrow$  such an expression is derived from the value of one of the  $x_i$  whereas in  $\Downarrow$  the expression directly yields the value  $\text{OR } r [x_1, x_2]$ . The key idea to construct a corresponding proof tree in  $\Downarrow$  from one in  $\downarrow$  is to update  $\rho$  with  $[r \mapsto i]$  whenever  $x_i$  is chosen in  $\downarrow$ . Furthermore, the rule (Lift) of  $\Downarrow$  makes sure that each case is finally applied to a head normal form according to the choice represented by  $\rho$ . The rest of the proof consists of showing that cut correctly maps between corresponding heaps.

We achieve sharing across non-determinism as the following example shows.

*Example 4 (Sharing in action).* In Example 1 we showed the importance of sharing across. As a simpler example for sharing across non-determinism, we consider the function `ins` which non-deterministically inserts an element to arbitrary position in a list:

```
ins x ys = case ys of {[]      -> x:[];
                    (a:as) -> (x:a:as) or (a:ins x as)}
```

Similar to our parser example, the evaluation of the element added to the list would be performed in every non-deterministic computation in most functional logic languages. To demonstrate how our new semantics shares these evaluation, we consider the following simple expression `(let {x = long} in ins(x, [x]))`, which computes only two results non-deterministically. Usually, the demands for different results of `ins` come from the context `ins` is used in. Here, we somewhat artificially force the demand by the context `(&&(hnf  $\rho$  h, hnf  $\tau$  h))` where  $h$  denotes the application of the `head` function to the above expression and  $\rho, \tau$  contain branching information such that  $\rho(1) = 1, \tau(1) = 2$ . This keeps the example manageable while proving the main point.

A linearization of the proof tree for this example is presented in Figure 4. For space constraints and sake of readability, the computation is somewhat abbreviated. Constructors are not normalized and sub computations are only introduced, if a look up in the heap occurs.

### 4.3 Encapsulated Search

The goal of this section is to provide a primitive function which computes the internal branching information and represents the search space as a *search tree* according to the definition (cf. [5]):

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

The programmer should be able to access the search tree of a given expression via the function `searchTree :: a -> SearchTree a`, which should provide the tree in a lazy manner (cf. [5]) such that different search strategies like breadth first and depth first search can easily be formulated just by matching the tree structure. Although we use the same definition for a search tree, in comparison to [5] the approach in this paper provides sharing across non-determinism and employs a much simpler mechanism with just a single heap.

Before we can reach our goal, we have to extend the setting by information about failure. To do this, we add the following rule:

$$\begin{array}{l}
\emptyset : \text{let } \{x=long, i=ins(x, x : []), h=head(i), n=hnf \rho h, m=hnf \tau h\} \text{ in } \&\&(n, m) \\
\Gamma := [x \mapsto long, i \mapsto ins(x, x : []), h \mapsto head(i), n \mapsto hnf \rho h, m \mapsto hnf \tau h] : \&\&(n, m) \\
\Gamma : \text{case } n \text{ of } \{T \rightarrow m, F \rightarrow F\} \\
\left[ \begin{array}{l}
\Gamma : n \\
\Gamma : hnf \rho h \\
\left[ \begin{array}{l}
\Gamma : h \\
\Gamma : head(i) \\
\Gamma : \text{case } i \text{ of } \{z : zs \rightarrow z\} \\
\left[ \begin{array}{l}
\Gamma : i \\
\Gamma : ins(x, x : []) \\
\Gamma : \text{case } x : [] \text{ of } \{[] \rightarrow x : [], a : as \rightarrow \text{let } \{l=ins(x, as)\} \text{ in } (x : a : as) \text{ or } (a : l)\} \\
\Gamma : \text{let } \{l=ins(x, [])\} \text{ in } (x : x : []) \text{ or } (x : l) \\
\Gamma' := \Gamma[l \mapsto ins(x, [])] : (x : x : []) \text{ or } (x : l) \\
\Gamma' : \text{OR } 1 [x : x : [], x : l] \\
\Gamma'' := \Gamma'[i \mapsto \text{OR } 1 [x : x : [], x : l]] : \text{OR } 1 [x : x : [], x : l] \\
\Delta := \Gamma''[y_1 \mapsto \text{case } x : x : [] \text{ of } \{z : zs \rightarrow z\}, \\
\quad y_2 \mapsto \text{case } x : l \text{ of } \{z : zs \rightarrow z\}] : \text{OR } 1 [y_1, y_2] \\
\Delta' := \Delta[h \mapsto \text{OR } 1 [y_1, y_2]] : \text{OR } 1 [y_1, y_2] \\
\Delta' : hnf \rho y_{\rho(1)} \\
\left[ \begin{array}{l}
\Delta' : y_1 \\
\Delta' : \text{case } x : x : [] \text{ of } \{z : zs \rightarrow z\} \\
\left. \begin{array}{l}
\Delta' : x \\
\Delta' : long \\
\dots \\
\Delta' : T
\end{array} \right\} \text{ here the long deterministic evaluation takes place} \\
\Theta := \Delta'[x \mapsto T] : T \\
\Theta' := \Theta[y_1 \mapsto T] : T \\
\Theta'' := \Theta'[n \mapsto T] : T \\
\Theta'' : m \\
\Theta'' : hnf \tau h \\
\left[ \begin{array}{l}
\Theta'' : h \\
\Theta'' : \text{OR } 1 [y_1, y_2] \\
\Theta'' : hnf \tau y_{\tau(1)} \\
\left[ \begin{array}{l}
\Theta'' : y_2 \\
\Theta'' : \text{case } x : l \text{ of } \{z : zs \rightarrow z\} \\
\left. \begin{array}{l}
\Theta'' : x \\
\Theta'' : T
\end{array} \right\} \text{ here the result of the long computation is looked up in the heap} \\
\Omega := \Theta''[y_1 \mapsto T] : T \\
\Omega[m \mapsto T] : T
\end{array} \right.
\end{array} \right.
\end{array} \right.
\end{array}
\end{array}
\end{array}$$

**Fig. 4.** Semantics of Example 4 – Proof Tree presented in Linearized Form

$$\text{(Fail)} \frac{\Gamma : e \Downarrow \Delta : c(\overline{x_n})}{\Gamma : \text{case } e \text{ of } \{\overline{p_k \Rightarrow e_k}\} \Downarrow \Delta : \text{FAIL}} \quad \text{where for all } i \text{ with } 1 \leq i \leq k \text{ holds: } p_i \neq c(\overline{y_n}).$$

Note, that especially, since FAIL is an internal constructor, it cannot be a pattern of any case expression. Furthermore, FAIL is a valid result of an application of rule (Hnf-Val), as defined above. By adding rule (Fail), we do neither introduce new results other than FAIL nor lose existing results:

**Theorem 4.**

- a) If  $\Gamma : e \Downarrow \Delta : \text{FAIL}$  then there exists no  $\Delta', v \neq \text{FAIL}$  such that  $\Gamma : e \Downarrow \Delta' : v$ .
- b) If  $\Gamma : e \Downarrow \Delta : v$  with  $v \neq \text{FAIL}$  then there exists no  $\Delta'$  such that  $\Gamma : e \Downarrow \Delta' : \text{FAIL}$ .

*Proof.*

- a) The internal constructor FAIL is only introduced by the rule (Fail). This rule is applicable only if no rule of  $\Downarrow$  can be applied.
- b) The only rule that changes the result of a derivation is (Select). Since FAIL is an internal constructor there can be no pattern matching it in any program. Thus, FAIL is the final result whenever it appears.

So far we have presented how a head normal form can be computed if the branching information  $\rho$  is already given. The next step is to define how this information is introduced. This is the responsibility of *encapsulation*. The first thing to do is to define how the hnf function should behave if the reference of a computed OR node is not contained in the given branching information  $\rho$ . In this case the evaluation to head normal form should stop:

$$\text{(Hnf-Stop)} \frac{\Gamma : e \Downarrow \Delta : \text{OR } r \ [\overline{x_n}]}{\Gamma : \text{hnf } \rho \ e \Downarrow \Delta : \text{OR } r \ [\overline{x_n}]} \quad \text{if } r \notin \text{Dom}(\rho)$$

We are now ready to define encapsulated search. We introduce the function `st` which mostly translates internal (untyped) OR and FAIL constructors with call-time choice references to the typed search trees without references, as defined at the beginning of this section. There is only one more thing, `st` does: the OR references are added to the branching information for hnf:

$$\text{(St-Val)} \frac{\Gamma : \text{hnf } \rho \ x \Downarrow \Delta : c(\overline{x_n})}{\Gamma : \text{st } \rho \ x \Downarrow \Delta [y \mapsto c(\overline{x_n})] : \text{Value}(y)} \quad \text{where } c \notin \{\text{OR}, \text{FAIL}\} \text{ and } y \text{ fresh}$$

$$\text{(St-Fail)} \frac{\Gamma : \text{hnf } \rho \ x \Downarrow \Delta : \text{FAIL}}{\Gamma : \text{st } \rho \ x \Downarrow \Delta : \text{Fail}}$$

$$\text{(St-Or)} \frac{\Gamma : \text{hnf } \rho \ x \Downarrow \Delta : \text{OR } r \ [\overline{x_n}]}{\Gamma : \text{st } \rho \ x \Downarrow \Delta \left[ \overline{y_n \mapsto \text{st}(\rho \cup \{(r, n)\})} \ x_n, y \mapsto (y_1 : \dots : y_n : [])^* \right] : \text{Or}(y)} \quad \text{where } y, \overline{y_n} \text{ fresh}$$

As mentioned above, we provide the function `searchTree` which can now be defined such that each expression of the form `(searchTree x)` is replaced by `(st  $\emptyset$  x)`. Our final theorem states that the programmer is thus provided with a *complete* representation of the search which he can traverse according to his needs.

**Theorem 5 (Completeness of Representation).** *If  $\Gamma : e \Downarrow \Delta : v$  then there exist a heap  $\Delta'$  and a case expression  $c := \text{case } x \text{ of } \{bs\}$  such that  $\Gamma : \text{let } \{x = \text{st } \emptyset e\} \text{ in } c \Downarrow \Delta' : v$ .*

*Proof (Central Idea).* By Theorem 3 there exists a sequence of updates  $s$  such that if  $\Gamma : e \Downarrow \Delta : v$  then  $\Gamma : \text{hnf } (\emptyset s) e \Downarrow \Delta' : v$ . As the function `st` invokes the function `hnf` systematically with all possible alternatives such that the  $i$ th element beneath an `Or` constructor contains the evaluation of `hnf` with the update  $[r \mapsto i]$  we need only to construct a case expression which chooses that  $i$ th element beneath `Or` while also eliminating the `Value` constructor at the end of the evaluation.

There are various reasons to employ encapsulated search. One is to express properties about all possible non-deterministic branches (e.g., their number) as given in the introduction. Other reasons are pruning the search space like done in the branch and bound method, to encapsulate non-determinism for the integration of external functions, which normally are defined on ground terms only, or to ensure that I/O operations do not perform conflicting actions. This latter use, called *complete encapsulation*, requires that arguments of the encapsulation's `Value` constructor are solutions, i.e. do not contain the internal constructors `OR` or `FAIL`. This can only be ensured by an evaluation to normal form. We include a simple example for computing a normal form mainly because the importance of complete encapsulation was convincingly discussed in [5].

*Example 5 (Complete Encapsulation).* We restrict ourselves to the simplest form of a recursive data structure. The extension to more complex structures is straight forward.

```
data Nat = Z | S Nat
nf Z = Z
nf (S x) = case nf x of {Z -> S Z; S y -> S (S y)}
```

Now the expression `(searchTree (nf x))` will compute a complete encapsulation of  $x$ , i.e. a search tree where the arguments of the `Value` constructor are *solutions*, they do not contain any internal constructor `OR` or `FAIL`. Note that `nf` copies the given data structure regardlessly. There are of course ways to do this more cleverly by defining a suitable primitive function. Also it could be convenient to define a polymorphic function which ensures the required property for any data structure, maybe using an overloaded function, like possible with Haskell's type classes, or use `=. .`-like term deconstruction methods as in Prolog.

Whether or not the encapsulation is complete, an important point about the presented encapsulation is that the result of the function `searchTree` is *generated lazily*. This implies that different search strategies can easily be defined on the level of the source language. E.g., for Curry we can easily define depth-first and breadth-first search, from which we only present the more interesting breadth-first search:

```
allValuesB :: SearchTree a -> [a]
allValuesB t = all [t]
  where all ts | null ts = []
        | otherwise = [x | Val x <- ts] ++
                      all [t | Or ts' <- ts, t <- ts']
```

Also, a *fair* search can be implemented by an *action fair* :: `SearchTree a -> IO [a]`, which forks a thread for each child of an `Or` node. As such a search essentially realizes a *committed choice* by computing the results in arbitrary order, it destroys the purity of the encapsulations presented so far. In consequence, we regard this function as an I/O-Operation.

## 5 Related work

There has been only one other approach prior to this work to formalize sharing deterministic computations across non-deterministic branches which is called bubbling [2]. Bubbling is defined as a graph rewriting technique and the call-time choice semantics is realized by manipulating the graph globally. We, in contrast, do only the local manipulation of lifting or-nodes and realize call-time choice by storing branching information and comparing or-references later on. This definitely speeds up deterministic computations in comparison, putting the whole overhead on branching. Since the implementation of bubbling is not yet finished, it remains difficult to judge which approach performs better in practice. The amount of sharing, however, is the same.

Our previous work [6] exhaustively discussed all aspects of encapsulation, but did not solve it as elegantly as the work presented here. Furthermore, sharing across non-determinism was not covered. However, it formulates a “wish list” for implementations of encapsulation, which is fully met by the presented approach. The general considerations of [6] also show that bubbling prohibits to reach the level of purity achieved here, because it executes a fair search beneath or-nodes prior to induce the non-determinism globally. This can be expected to increase the efficiency of a certain class of programs. But it can not lead to deterministic encapsulation as fair search is essentially a committed choice.

## 6 Conclusion

We have presented a new operational semantics for functional logic languages, like Curry [11]. It covers in a clear and seamless way two main problems of the integration of functional and logic languages: encapsulated search and sharing across non-determinism. Both are in our opinion key issues for the applicability of functional logic languages in practice. The key idea is to handle non-determinism as a (lazily constructed) data structure. We obtain a deterministic semantics in which search strategies, like depth-first and breadth-first search, can easily be defined as pure functions on top of the resulting `SearchTree` by the user. The up to now unreachable purity of encapsulation enables new programming methods like the delaying of non-deterministic choices.

At the moment we have two implementations based on this semantics. The first is an interpreter, exactly implementing the operational semantics  $\Downarrow$ . It is very useful for analyzing how the operational semantics works and for computing small examples. We also developed a compiler for this semantics. The key feature of the implementation is the representation of the search tree as a data structure in the heap, which is not provided by the existing Curry implementation and makes the extension of these systems very difficult. Hence, we

implemented a new compiler which translates Curry to Haskell (the Kiel Curry System, *KiCS*). It implements Curry's non-determinism by means of extended Haskell data structures providing special constructors for representing the internal OR and FAIL constructors of our semantics. The implementation handles encapsulated search and sharing across non-determinism similar to the semantics presented in this paper. In many practical applications this compiler has proven that the presented approach is feasible for general use.

## 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, D. Brown, and S.-H. Chiang. On the correctness of bubbling. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications*, pages 35–49, Seattle, WA, Aug. 2006. Springer LNCS 4098.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
4. 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.
5. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
6. B. Braßel and F. Huch. Translating Curry to Haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 60–65. ACM Press, 2005.
7. B. Braßel and F. Huch. On the tighter integration of functional and logic programming. Technical Report 0710, Institute of Computer Science, CAU Kiel, 2007.
8. J. Dios and F.J. López-Fraguas. Elimination of extra variables from functional logic programs. In P. Lucio and F. Orejas, editors, *VI Jornadas sobre Programación y Lenguajes (PROLE 2006)*, pages 121–135. CINME, 2006.
9. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
10. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at <http://www.informatik.uni-kiel.de/~pakcs/>, 2006.
11. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at <http://www.informatik.uni-kiel.de/~curry>, 2006.
12. R. Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.
13. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
14. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
15. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer, 1999.