Curry

An Integrated Functional Logic Language

Version 0.9.0

January 13, 2016

Michael Hanus[1] [editor]

Additional Contributors:

Sergio Antoy[2]
Bernd Braßel[3]
Herbert Kuchen[4]
Francisco J. López-Fraguas[5]
Wolfgang Lux[6]
Juan José Moreno Navarro[7]
Björn Peemöller[8]
Frank Steiner[9]

(1) University of Kiel, Germany, mh@informatik.uni-kiel.de
(2) Portland State University, USA, antoy@cs.pdx.edu
(3) University of Kiel, Germany, bbr@informatik.uni-kiel.de
(4) University of Münster, Germany, kuchen@uni-muenster.de
(5) Universidad Complutense de Madrid, Spain, fraguas@dia.ucm.es
(6) University of Münster, Germany, wlux@uni-muenster.de
(7) Universidad Politécnica de Madrid, Spain, jjmoreno@fi.upm.es
(8) University of Kiel, Germany, bjp@informatik.uni-kiel.de
(9) University of Munich, Germany, fst@bio.informatik.uni-muenchen.de

Contents

1 Introduction

Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic evaluation of functions). Moreover, it also amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [8, 22, 25] for surveys on functional logic programming).

The development of Curry is an international initiative intended to provide a common platform for the research, teaching[1] and application of integrated functional logic languages. This document describes the features of Curry, its syntax and operational semantics.

[1] Actually, Curry has been successfully applied to teach functional and logic programming techniques in a single course without switching between different programming languages. More details about this aspect can be found in [23].

2 Programs

A Curry program specifies the semantics of expressions. Executing a Curry program means simplifying an expression until a value (together with bindings of free variables) is computed. To distinguish between values and reducible expressions, Curry has a strict distinction between (data) *constructors* and *operations* or *defined functions* on these data. Hence, a Curry program consists of a set of type and function declarations. The type declarations define the computational domains (constructors) and the function declarations the operations on these domains. Predicates in the logic programming sense can be considered as functions with result type `Bool`. Goals, which are the main expressions in logic programming, are Boolean expressions that are intended to be evaluated to `True`.

Modern functional languages (e.g., Haskell [41], SML [37]) allow the detection of many programming errors at compile time by the use of polymorphic type systems. Similar type systems are also used in modern logic languages (e.g., Gödel [28], λProlog [39]). Curry follows the same approach, i.e., it is a strongly typed language with a Hindley/Milner-like polymorphic type system [14].[2] Each object in a program has a unique type, where the types of variables and operations can be omitted and are reconstructed by a type inference mechanism.

2.1 Datatype Declarations

A datatype declaration has the form

$$\texttt{data}\ T\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ C_1\ \tau_{11} \ldots \tau_{1n_1}\ \texttt{|}\ \cdots\ \texttt{|}\ C_k\ \tau_{k1} \ldots \tau_{kn_k}$$

and introduces a new n-ary *type constructor* T and k new (data) *constructors* $C_1, \ldots, C_k$, where each C_i has the type

$$\tau_{i1}\ \texttt{->}\ \cdots\ \texttt{->}\ \tau_{in_i}\ \texttt{->}\ T\ \alpha_1 \ldots \alpha_n$$

($i = 1, \ldots, k$). Each τ_{ij} is a *type expression* built from the *type variables* $\alpha_1, \ldots, \alpha_n$ and some type constructors. Curry has a number of built-in type constructors, like `Bool`, `Int`, `->` (function space), or, lists and tuples, which are described in Section 4.1. Since Curry is a higher-order language, the types of functions (i.e., constructors or operations) are written in their curried form τ_1 `->` τ_2 `->` $\cdots$ `->` τ_n `->` τ where τ is not a functional type. In this case, n is called the *arity* of the function. For instance, the datatype declarations

```
data Bool = True | False
data List a = [] | a : List a
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

introduces the datatype `Bool` with the 0-ary constructors (*constants*) `True` and `False`, and the polymorphic types `List a` and `Tree a` of lists and binary trees. Here, "`:`" is an infix operator, i.e., "`a:List a`" is another notation for "`(:) a (List a)`". Lists are predefined in Curry, where the notation "`[a]`" is used to denote list types (instead of "`List a`"). The usual convenient notations for lists are supported, i.e., `[0,1,2]` is an abbreviation for `0:(1:(2:[]))` (see also Section 4.1).

A *data term* is a variable x or a constructor application $c\, t_1 \ldots t_n$ where c is an n-ary constructor

[2] The extension of this type system to Haskell's type classes is not included in the kernel language but could be considered in a future version.

and $t_1, \ldots, t_n$ are data terms. An *expression* is a variable or a (partial) application $\varphi\ e_1 \ldots e_m$ where φ is a function or constructor and $e_1, \ldots, e_m$ are expressions. A data term or expression is called *ground* if it does not contain any variable. Ground data terms correspond to values in the intended domain, and expressions containing operations should be evaluated to data terms. Note that traditional functional languages compute on ground expressions, whereas logic languages also allow non-ground expressions.

2.2 Type Synonym Declarations

To make type definitions more readable, it is possible to specify new names for type expressions by a *type synonym declaration*. Such a declaration has the general form

`type` $T\ \alpha_1 \ldots \alpha_n$ `=` τ

which introduces a new n-ary type constructor T. $\alpha_1, \ldots, \alpha_n$ are pairwise distinct type variables and τ is a type expressions built from type constructors and the type variables $\alpha_1, \ldots, \alpha_n$. The type $(T\ \tau_1 \ldots \tau_n)$ is equivalent to the type $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}(\tau)$, i.e., the type expression τ where each α_i is replaced by τ_i. Thus, a type synonym and its definition are always interchangeable and have no influence on the typing of a program. For example, we can provide an alternative notation for list types and strings by the following type synonym declarations:

```
type List a = [a]
type Name = [Char]
```

Since a type synonym introduces just another name for a type expression, recursive or mutually dependent type synonym declarations are not allowed. Therefore, the following declarations are *invalid*:

```
type RecF a = a -> RecF a   -- recursive definitions not allowed
type Place = [Addr]         -- mutually recursive definitions not allowed
type Addr = [Place]
```

However, recursive definitions with an intervening datatype are allowed, since recursive datatype definitions are also allowed. For instance, the following definitions are valid:

```
type Place = [Addr]
data Addr = Tag [Place]
```

2.3 Function Declarations

A function is defined by a type declaration (which can be omitted) followed by a list of defining equations. A *type declaration* has the form

f `::` τ_1 `->` τ_2 `->` $\cdots$ `->` τ_k `->` τ

where $\tau_1, \ldots, \tau_k, \tau$ are type expressions.

The simplest form of a *defining equation* (or *rule*) for an n-ary function f ($n \geq 0$) is

$f\ t_1 \ldots t_n$ `=` e

where $t_1, \ldots, t_n$ are data terms (also called *patterns*) and the *right-hand side* e is an expression. Functions can also be defined by *conditional equations* which have the form

$f\ t_1 \ldots t_n$ `|` c `=` e

where the *condition* c is a Boolean expression A conditional equation can be applied if its condition can be evaluated to `True`.

If the *left-hand side* $f\ t_1 \ldots t_n$ of a defining equation contains multiple occurrences of a variable, these occurrences are considered as an equational constraint (cf. Section 2.6) between the arguments. Thus, a function declaration

```
f x x = g x
```

is considered as an abbreviation of the conditional equation

```
f x y | x=:=y = g x
```

Thus, each multiple variable occurrence is replaced by a new variable and an equational constraint is added. These constraints are the first constraints to be solved. For instance, the conditional equation

```
f x y x y | c x y = g x
```

is considered as an abbreviation for

```
f x y x1 y1 | (x=:=x1 & y=:=y1) &> c x y = g x
```

An n-ary function f can be defined by more than one (conditional) equation (where each rule must have the same number of arguments on the left-hand side). Note that one can define functions with a non-determinate behavior by providing several rules with overlapping left-hand sides or *free variables* (i.e., variables which do not occur in the left-hand side) in the conditions or right-hand sides of rules. For instance, the following non-deterministic function inserts an element at an arbitrary position in a list:

```
insert x []     = [x]
insert x (y:ys) = x : y : ys
insert x (y:ys) = y : insert x ys
```

Such *non-deterministic functions* can be given a perfect declarative semantics [19] and their implementation causes no overhead in Curry since techniques for handling non-determinism are already contained in the logic programming part (see also [5]). However, deterministic functions are advantageous since they provide for more efficient implementations (like the *dynamic cut* [35]). If one is interested only in defining deterministic functions, this can be ensured by the following restrictions:

1. Each variable occurring in the right-hand side of a rule must also occur in the corresponding left-hand side.

2. If $f\ t_1 \ldots t_n$ `|` c `=` e and $f\ t'_1 \ldots t'_n$ `|` c' `=` e' are rules defining f and σ is a *substitution*[3] with $\sigma(t_1 \ldots t_n) = \sigma(t'_1 \ldots t'_n)$, then at least one of the following conditions holds:

[3] A *substitution* σ is a mapping from variables into expressions which is extended to a homomorphism on expressions by defining $\sigma(f\ t_1 \ldots t_n) = f\ \sigma(t_1) \ldots \sigma(t_n)$. $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}$ denotes the substitution σ with $\sigma(x_i) = e_i$ $(i = 1, \ldots, k)$ and $\sigma(x) = x$ for all variables $x \notin \{x_1, \ldots, x_k\}$.

(a) $\sigma(e) = \sigma(e')$ (*compatibility of right-hand sides*).

(b) $\sigma(c)$ and $\sigma(c')$ are not simultaneously satisfiable (*incompatibility of conditions*). A decidable approximation of this condition can be found in [31].

These conditions ensure the confluence of the rules if they are considered as a conditional term rewriting system [46]. Implementations of Curry may check these conditions and warn the user if they are not satisfied. There are also more general conditions to ensure confluence [46] which can be checked instead of the above conditions.

2.3.1 Functions vs. Variables

In lazy functional languages, different occurrences of the same variable are shared to avoid multiple evaluations of identical expressions. For instance, if we apply the rule

```
double x = x+x
```

to an expression `double`t, we obtain the new expression t`+`t but both occurrences of t denote the identical expression, i.e., these subterms will be simultaneously evaluated. Thus, *several occurrences of the same variable are always shared*, i.e., if one occurrence of an argument variable, which might bound to an evaluable expression when the function is applied, is evaluated to some value, all other occurrences of this variable are replaced by the same value (without evaluating these other occurrences again). This sharing is necessary not only for efficiency reasons but it has also an influence on the soundness of the operational semantics in the presence of non-deterministic functions (see also [19]). For instance, consider the non-deterministic function `coin` defined by the rules

```
coin = 0
coin = 1
```

Thus, the expression `coin` evaluates to `0` or `1`. However, the result values of the expression `(double coin)` depend on the sharing of the two occurrences of `coin` after applying the rule for `double`: if both occurrences are shared (as in Curry), the results are `0` or `2`, otherwise (without sharing) the results are `0`, `1`, or `2`. The sharing of argument variables corresponds to the so-called "call time choice" in the declarative semantics [19]: if a rule is applied to a function call, a unique value must be assigned to each argument (in the example above: either `0` or `1` must be assigned to the occurrence of `coin` when the expression `(double coin)` is evaluated). This does not mean that functions have to be evaluated in a strict manner but this behavior can be easily obtained by sharing the different occurrences of a variable.

Since different occurrences of the same variable are always shared but different occurrences of (textually identical) function calls are not shared, it is important to distinguish between variables and functions. Usually, all symbols occurring at the top level in the left-hand side of some rule are considered as functions and the non-constructor symbols in the arguments of the left-hand sides are considered as variables. But note that there is a small exception from this general rule in local declarations (see Section 2.4).

2.3.2 Rules with Multiple Guards

One can also write conditional equations with multiple guards

$f\ t_1 \ldots t_n$ | b_1 = e_1
⋮
| b_k = e_k

where $b_1, \ldots, b_k$ ($k > 0$) are expressions of type `Bool`. Such a rule is interpreted as in Haskell: the guards are successively evaluated and the right-hand side of the first guard which is evaluated to `True` is the result of applying this equation. Thus, this equation can be considered as an abbreviation for the rule

$f\ t_1 \ldots t_n$ = if b_1 then e_1 else
⋮
if b_k then e_k else failed

(where `failed` is a non-reducible function defined in the prelude). To write rules with several Boolean guards more nicely, there is a Boolean function `otherwise` which is predefined as `True`. For instance, the factorial function can be declared as follows:

```
fac n | n==0      = 1
      | otherwise = fac(n-1)*n
```

After performing a simple optimization, this definition is equivalent to

```
fac n = if n==0 then 1 else fac(n-1)*n
```

Note that multiple guards in a single rule are always sequentially tested, whereas multiple rules for a function are non-deterministically applied (see the definition of `insert` in in Section 2.3). Thus, the following definition of `fac` has a different meaning than the definition with multiple guards above, since the second rule is always applicable:

```
fac n | n==0      = 1
fac n | otherwise = fac(n-1)*n
```

2.3.3 As-Patterns

The patterns in a defining equation, i.e., the arguments of the left-hand sides, are required to be data terms without multiple occurrences of variables. Patterns are useful to split the definition of functions into easily comprehensible cases. Thus, a pattern denotes some part of a structure of the actual argument. Sometimes one wants to reuse this structure in the right-hand side of the defining equation. This can be expressed by an *as-pattern* which allows to identify this structure by a variable. An as-pattern has the form v`@`pat where the variable v identifies the structure matched by the pattern pat. For instance,

```
dropFalse (False:ys)  = ys
dropFalse xs@(True:_) = xs
```

is equivalent to

```
dropFalse0 xs = dropFalse' xs
 where
  dropFalse' (False:ys)  = ys
  dropFalse' (True:_)    = xs
```

(local declarations introduced by `where`-clauses are discussed in Section 2.4). However, as-patterns are usually implemented more efficiently without introducing an auxiliary function.

2.4 Local Declarations

Since not all auxiliary functions should be globally visible, it is possible to restrict the scope of declared entities. Note that the scope of parameters in function definitions is already restricted since the variables occurring in parameters of the left-hand side are only visible in the corresponding conditions and right-hand sides. The visibility of other entities can be restricted using `let` in expressions or `where` in defining equations.

An expression of the form `let` *decls* `in` *exp* introduces a set of local names. The list of local declarations *decls* can contain function definitions as well as definitions of constants by pattern matching. The names introduced in these declarations are visible in the expression *exp* and the right-hand sides of the declarations in *decls*, i.e., the local declarations can be mutually recursive. For instance, the expression

```
let a=3*b
    b=6
in 4*a
```

reduces to the value `72`.

Auxiliary functions which are only introduced to define another function should often not be visible outside. Therefore, such functions can be declared in a `where`-clause added to the right-hand side of the corresponding function definition. This is done in the following definition of a fast exponentiation where the auxiliary functions `even` and `square` are only visible in the right-hand side of the rule for `exp`:

```
exp b n = if n==0
          then 1
          else if even n then square (exp b (n `div` 2))
                         else b * (exp b (n-1))

       where even n = n `mod` 2 == 0
             square n = n*n
```

Similarly to `let`, `where`-clauses can contain mutually recursive function definitions as well as definitions of constants by pattern matching. The names declared in the `where`-clauses are only visible in the corresponding conditions and right-hand sides. As a further example, the following Curry program implements the quicksort algorithm with a function `split` which splits a list into two lists containing the smaller and larger elements:

```
split e []            = ([],[])
split e (x:xs) | e>=x = (x:l,r)
               | e<x  = (l,x:r)
               where
                 (l,r) = split e xs

qsort []     = []
qsort (x:xs) = qsort l ++ (x:qsort r)
```

```
where
  (l,r) = split x xs
```

To distinguish between locally introduced functions and variables (see also Section 2.3.1), we define a *local pattern* as a (variable) identifier or an application where the top symbol is a data constructor. If the left-hand side of a local declaration in a `let` or `where` is a pattern, then all identifiers in this pattern that are not data constructors are considered as variables. For instance, the locally introduced identifiers `a`, `b`, `l`, and `r` in the previous examples are variables whereas the identifiers `even` and `square` denote functions. Note that this rule exclude the definition of 0-ary local functions since a definition of the form "`where f = ...`" is considered as the definition of a local variable `f` by this rule which is usually the intended interpretation (see previous examples). Appendix D.7 contains a precise formalization of the meaning of local definitions.

2.5 Free Variables

Since Curry is intended to cover functional as well as logic programming paradigms, expressions might contain free (unbound, uninstantiated) variables. The idea is to compute values for these variables such that the expression is reducible to a data term. For instance, consider the definitions

```
data Person = John | Christine | Alice | Andrew

mother John    = Christine
mother Alice   = Christine
mother Andrew  = Alice
```

Then we can compute a child of `Alice` by solving the equation (see Section 2.6)

```
mother x == Alice
```

Here, `x` is a free variable which is instantiated to `Andrew` in order to reduce the equation's left-hand side to `Alice` and compute the result `True` for this equation. Note that we can also compute the result `False` by instantiating `x` to `John` or `Alice`. If we are interested to obtain only positive solutions, we can wrap the initial expression with the operation `solve`, which is defined in the prelude by the single rule

```
solve True = True
```

Hence, the equation

```
solve (mother x == Alice)
```

reduces only to `True` and yields the value `Andrew` for `x`. Using the infix application operator `$` defined in the prelude, we can omit the parentheses and write this equation as

```
solve $ mother x == Alice
```

Similarly, we can compute a grandchild of `Chistine` by solving the equation

```
mother (mother x) == Christine
```

which yields the value `Andrew` for `x`.

In logic programming languages like Prolog, all free variables are considered as existentially

quantified at the top level. Thus, they are always implicitly declared at the top level. In a language with different nested scopes like Curry, it is not clear to which scope an undeclared variable belongs (the exact scope of a variable becomes particularly important in the presence of search operators, see Section 8, where existential quantifiers and lambda abstractions are often mixed). Therefore, Curry requires that *each free variable x must be explicitly declared* using a local declaration of the form x `free`. The variable is then introduced as unbound with the same scoping rules as for all other local entities (see Section 2.4). For instance, we can define

```
isGrandmother g | let c free in mother (mother c) == g  = True
```

An exception are *anonymous free variables*, i.e., free variables occurring only once in an expression, which can be denoted by "`_`". Thus, each occurrence of "`_`" in an expression can be considered as an abbreviation of "`let x free in x`".

As a further example, consider the definition of the concatentation of two lists:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Then we can define the function `last` which computes the last element of a list by the rule

```
last zs | append xs [x] == zs  = x  where x,xs free
```

Since the variable `xs` occurs in the condition but not in the right-hand side, the following definition is also possible:

```
last zs | let xs free in append xs [x] == zs  = x  where x free
```

which can abbreviated to

```
last zs | append _ [x] == zs  = x  where x free
```

Note that the `free` declarations can be freely mixed with other local declarations after a `let` or `where`. The only difference is that a declaration like "`let x free`" introduces an unbound variable whereas other `let` declarations introduce local functions or parameters. Since all local declarations can be mutually recursive, it is also possible to use local variables in the bodies of the local functions in one `let` declarations. For instance, the following expression is valid (where the functions `h` and `k` are defined elsewhere):

```
let f x = h x y
    y free
    g z = k y z
 in c y (f (g 1))
```

Similarly to the usual interpretation of local definitions by lambda lifting [30], this expression can be interpreted by transforming the local definitions for `f` and `g` into global ones by adding the non-local variables of the bodies as parameters:

```
f y x = h x y
g y z = k y z
...
  let y free in c y (f y (g y 1))
```

See Appendix D.7 for more details about the meaning and transformation of local definitions.

2.6 Equality and Constraints

In order to compare the values of two expressions, there is a predefined predicate "==" to test the convertibility of expressions to identical data terms. The expression e_1==e_2 reduces to `True` if e_1 and e_2 are reducible to identical ground data terms, and it reduces to `False` if e_1 and e_2 are reducible to different data terms. If we ignore the strong type system of Curry,[4] we can consider the equality predicate as defined by the following rules:

```
C          == C          = True                    -- for all 0-ary constructors C
C x1...xn  == C y1...yn  = x1==y1 &&...&& xn==yn   -- for all n-ary constructors C
C x1...xn  == D y1...ym  = False                   -- for all different constructors C and D
```

For instance, consider the datatype `Person` as defined in Section 2.5 and lists as defined in Section 2.1. Then the equality on these types is defined by the following rules (where "`&&`" is the Boolean conjunction, see Section 4.1.1):

```
John       == John       = True
Christine  == Christine  = True
Alice      == Alice      = True
Andrew     == Andrew     = True
John       == Christine  = False
⋮
Andrew     == Alice      = False

[]         == []         = True
(x:xs)     == (y:ys)     = x==y && xs==ys
[]         == (_:_)      = False
(_:_)      == []         = False
```

This equality test, which is sometimes called *strict equality* and also used in functional languages, is the only sensible notion of equality in the presence of non-terminating functions [18, 38]. For instance, "`[John]==[John]`" reduces to `True` and "`[John]==[John,Andrew]`" reduces to `False`.

If one expression contains free variables, they are instantiated in order to evaluate the equality. For instance, "`[John]==x`" binds `x` to `[John]` and reduces to `True`, but it also reduces to `False` with the bindings of `x` to `[]`, `(Alice:_)` etc.

If one is interested only in bindings for positive solutions, i.e., reductions to `True`, one can also use the predefined operation "`=:=`". Conceptually, it can be considered as defined by the "positive" "`==`"-rules:

```
C          =:= C          = True                     -- for all 0-ary constructors C
C x1...xn  =:= C y1...yn  = x1=:=y1 &...& xn=:=yn    -- for all n-ary constructors C

True & True = True
```

With these rules, "`[John]=:=x`" has a unique result, i.e., `x` is bound to `[John]` so that the value `True` is returned. Since e_1=:=e_2 can be interpreted as a constraint between e_1 and e_2 that must be solved, it is also called an *equational constraint* between the expressions e_1 and e_2. Operationally,

[4] Although "`==`" is defined like a polymorphic function, it should be better considered as an overloaded function symbol. This could be more precisely expressed using type classes which might be included in a future extensions of Curry.

the equational constraint e_1`=:=`e_2 is solved by evaluating e_1 and e_2 to unifiable data terms and computing the unifier between these data terms. For instance, "`[John,x]=:=[y,z]`" is solved by binding `y` to `John` and `x` to `z` (or vice versa). This is in contrast to "`[John,x]==[y,z]`" which is solved by binding `y` to `John` and both `x` and `z` to `John`, `Christine`, `Alice`, or `Andrew`. Hence, the equational constraint yields one solution whereas the equality "`==`" yields four solutions (as well as many more non-solutions). Therefore, "`=:=`" can be considered as an optimization of "`==`", but it can only be used in contexts where positive solutions are required, e.g., in conditions of program rules.

An equational constraint could also be solved in an incremental manner by an interleaved lazy evaluation of the expressions and binding of variables to constructor terms (see [34] or Section D.3 in the appendix). Note that the basic kernel of Curry only provides strict equations e_1`=:=`e_2 between expressions as elementary constraints. Since it is conceptually fairly easy to add other constraint structures [36], extensions of Curry may provide richer constraint systems to support constraint logic programming applications.

As shown in the definition above, equational constraints are combined into a *conjunction* with the operator "`&`". Since this operators demands that both arguments must be evaluated to `True` in order to obtain any result (otherwise, its evaluation fails), this conjunction is interpreted *concurrently*: if the combined constraint c_1 `&` c_2 should be solved, c_1 and c_2 are solved concurrently. In particular, if the evaluation of c_1 suspends (see Sections 4.1.3 or 5.6), the evaluation of c_2 can proceed which may cause the reactivation of c_1 at some later time (due to the binding of common variables). In a sequential implementation, the evaluation of c_1 `&` c_2 could be started by an attempt to solve c_1. If the evaluation of c_1 suspends, an evaluation step is applied to c_2.

It is interesting to note that parallel functional computation models [12, 13] are covered by the use of concurrent constraints. For instance, a constraint of the form

```
x =:= f t1  &  y =:= g t2  &  z =:= h x y
```

specifies a potentially concurrent computation of the functions `f`, `g` and `h` if the function `h` can proceed its computation only if the arguments have been bound by evaluating the expressions `f t1` and `g t2`. Since constraints can be passed as arguments or results of functions (like any other data object or function), it is possible to specify general operators to create flexible communication architectures similarly to Goffin [13]. Thus, the same abstraction facilities could be used for sequential as well as concurrent programming. On the other hand, the clear separation between sequential and concurrent computations (e.g., a program without any occurrences of concurrent conjunctions is purely sequential) supports the use of efficient and optimal evaluation strategies for the sequential parts [5, 7], where similar techniques for the concurrent parts are not available.

2.7 Higher-order Features

Curry is a higher-order language supporting common functional programming techniques by partial function applications and lambda abstractions. *Function application* is denoted by juxtaposition the function and its argument. For instance, the well-known `map` function is defined in Curry by

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

However, there is an important difference w.r.t. to functional programming. Since Curry is also a logic language, it allows logical variables also for functional values, i.e., it is possible to evaluate the equation `map f [1 2] =:= [2 3]` which has, for instance, a solution {`f=inc`} if `inc` is the increment function on natural numbers. In principle, such solutions can be computed by extending (first-order) unification to *higher-order unification* [26, 39, 43]. Since higher-order unification is a computationally expensive operation, Curry delays the application of unknown functions until the function becomes known [2, 45].[5]

In situations where a function is only used a single time, it is tedious to assign a name to it. For such cases, anonymous functions (λ-*abstractions*), denoted by

$\backslash x_1 \ldots x_n$`->`$e$

are provided.

[5] Note that an unbound functional variable can never be instantiated if all program rules are constructor-based and the equational constraint `=:=` denotes equality between data terms. However, extensions of Curry might overcome this weakness by instantiating unbound functional variables to (type-conform) functions occurring in the program in order to evaluate an application (as in [20]), or by considering partial applications (i.e., functions calls with less than the required number of arguments) as data terms (as in [48]).

3 Operational Semantics

Curry's operational semantics is based on the lazy evaluation of expressions combined with a possible instantiation of free variables occurring in the expression. If the expression is ground, the operational model is similar to lazy functional languages, otherwise (possibly non-deterministic) variable instantiations are performed as in logic programming. If an expression contains free variables, it may be reduced to different values by binding the free variables to different expressions. In functional programming, one is interested in the computed *value*, whereas logic programming emphasizes the different bindings (*answers*). Thus, we define for the integrated functional logic language Curry an *answer expression* as a pair "$\sigma \,[]\, e$" consisting of a substitution σ (the answer computed so far) and an expression e. An answer expression $\sigma \,[]\, e$ is *solved* if e is a data term. Usually, the identity substitution in answer expressions is omitted, i.e., we write e instead of $\{\} \,[]\, e$ if it is clear from the context.

Since more than one answer may exist for expressions containing free variables, in general, initial expressions are reduced to disjunctions of answer expressions. Thus, a *disjunctive expression* is a (multi-)set of answer expressions $\{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$. For the sake of readability, we write concrete disjunctive expressions like

$$\{\{\mathtt{x} = 0, \mathtt{y} = 2\} \,[]\, 2 \;,\; \{\mathtt{x} = 1, \mathtt{y} = 2\} \,[]\, 3\}$$

in the form `{x=0,y=2} 2 | {x=1,y=2} 3`. Thus, substitutions are represented by lists of equations enclosed in curly brackets, and disjunctions are separated by vertical bars.

A single *computation step* performs a reduction in exactly one unsolved expression of a disjunction (e.g., in the leftmost unsolved answer expression in Prolog-like implementations). If the computation step is deterministic, the expression is reduced to a new one. If the computation step is non-deterministic, the expression is replaced by a disjunction of new expressions. The precise behavior depends on the function calls occurring in the expression. For instance, consider the following rules:

```
f 0 = 2
f 1 = 3
```

The result of evaluating the expression `f 1` is `3`, whereas the expression `f x` should be evaluated to the disjunctive expression

```
{x=0} 2 | {x=1} 3 .
```

To avoid superfluous computation steps and to apply programming techniques of modern functional languages, nested expressions are evaluated lazily, i.e., the leftmost outermost function call is primarily selected in a computation step. Due to the presence of free variables in expressions, an expression may have a free variable at a position where a value is demanded. A value is *demanded* in an argument of a function call if the left-hand side of some rule has a constructor at this position, i.e., in order to apply the rule, the actual value at this position must be the constructor. A value is also demanded in case expressions (see Section 5.3) or in arguments of external functions (see Section 9), in particular, in the primitive operation `ensureNotFree` (see Section 5.6). In this situation there are two possibilities to proceed:

1. Delay the evaluation of this expression until the corresponding free variable is bound (this

corresponds to the *residuation* principle which is the basis of languages like Escher [32, 33], Le Fun [2], Life [1], NUE-Prolog [40], or Oz [45]).

2. If the value is demanded by the left-hand sides of some defined function, (non-deterministically) instantiate the free variable to the different values required by these left-hand sides and apply reduction steps using the different rules (this corresponds to *narrowing* principle which is the basis of languages like ALF [21], Babel [38], K-Leaf [18], LPG [9], or SLOG [17]).

Since Curry is an attempt to provide a common platform for different declarative programming styles and both are reasonable in different contexts, Curry supports both alternatives. If the value of a free variable is demanded by the left-hand sides of a defined function, the second alternative (instantiation) is used, otherwise the evaluation is suspended and the evaluation proceeds with another subexpression (usually, another constraint). Free variables can also be instantiated by constraint solving, e.g., by evaluating an equational constraint e_1`=:=`e_2 (see Section 2.6). Most external functions, i.e., functions that are not defined by explicit program rules but implemented by primitive code (see Section 9), like arithmetic operators (see Section 4.1.3), suspend if some argument is an unbound variable.

For instance, consider the function `f` as defined above. The expression `f x` is evaluated by instantiating `x` to `0` or `1` and applying a reduction step in both cases. This yields the disjunctive expression

```
{x=0} 2 | {x=1} 3 .
```

On the other hand, primitive arithmetic functions like "`+`" or "`*`" suspend if some argument is a free variable, e.g., "`2+x`" can not be evaluated to some result. However, the use of such functions is reasonable if there is a "generator" for values for `x`. This is the case in the following constraint:

```
2+x =:= y & f x =:= y
```

Here, the first constraint `2+x =:= y` cannot be evaluated and suspends, but the second constraint `f x =:= y` can be solved by binding `x` to `1`, evaluating `f 1` to `3`, and binding `y` to `3`. After this binding (or even after binding `x`), the first constraint can be evaluated and the entire constraint is solved. Thus, the constraint is solved by the following steps (note that this is not the only possible order of steps):

```
2+x =:= y & f x =:= y
⇝ {x=1} 2+1 =:= y & 3 =:= y
⇝ {x=1,y=3} 2+1 =:= 3
⇝ {x=1,y=3} 3 =:= 3
⇝ {x=1,y=3}
```

(The empty constraint is omitted in the final answer.)

4 Types

4.1 Built-in Types

This section describes the types that are predefined in Curry. For each type, some important operations are discussed. The complete definition of all operations can be found in the standard prelude (see Appendix B).

4.1.1 Boolean Values

Boolean values are predefined by the datatype declaration

```
data Bool = True | False
```

The (sequential) conjunction is predefined as the left-associative infix operator `&&`:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

Similarly, the (sequential) disjunction "`||`" and the negation `not` are defined as usual (see Appendix B). Furthermore, the function `otherwise` is predefined as `True` to write rules with multiple guards more nicely.

Boolean values can be used in conditionals, i.e., the conditional function `if_then_else` is predefined as

```
if_then_else :: Bool -> a -> a -> a
if_then_else b t f = case b of True  -> t
                               False -> f
```

where "`if b then x else y`" is syntactic sugar for the application `(if_then_else b x y)`. Due to the definition based on a case expression (see Section 5.3), the evaluation of a conditional suspends of the discriminating Boolean argument is unknown.

A function with result type `Bool` is often called a *predicate*.[6] The predefined standard predicates "`==`" and "`=:=`" have already been discussed in Section 2.6. Furthermore, there are also a number of built-in predicates for comparing objects, like `compare`, "`<`", etc. User-defined data types are compared in the order of their definition in the datatype declarations and recursively in the arguments. For instance, if `Coin` is defined by

```
data Coin = Head | Tail
```

then the operation `compare` can be considered as defined by the rules

```
compare Head Head = EQ
compare Head Tail = LT
compare Tail Head = GT
compare Tail Tail = EQ
```

Note that one can also compare expressions containing free variables. For instance, the evaluation of "`x < [Tail]`" yields `True` for the bindings {`x=[]`} and {`x=(Head:_)`}. For built-in types like numbers

[6] Predicates in the logic programming sense are often partially defined, i.e., they reduce to `True` or their evaluation just fails.

or characters, some Curry systems might suspend when comparing unknown values.

The prelude defines also an operator "`&>`" to attach a constraint to an arbitrary expression:

```
(&>) :: Bool -> a -> a
True &> x = x
```

Hence, "`c &> e`" can be considered as a *constrained expression*, i.e., it is evaluated by first solving the constraint `c` and then evaluating `e`. This operator can be used to enforce a sequential order for constraint evaluation, e.g., the combined constraint c_1 `&>` c_2 will be evaluated by first completely evaluating c_1 and then c_2.

4.1.2 Functions

The type `t1 -> t2` is the type of a function which produces a value of type `t2` for each argument of type `t1`. A function f is applied to an argument x by writing "$f\ x$". The type expression

t_1 `->` t_2 `->` $\cdots$ `->` t_{n+1}

is an abbreviation for the type

t_1 `->` (t_2 `->` ($\cdots$ `->` t_{n+1}))

and denotes the type of a (currified) n-ary function, i.e., `->` associates to the right. Similarly, the expression

$f\ e_1\ e_2\ \ldots\ e_n$

is an abbreviation for the expression

(...(($f\ e_1$) e_2) ... e_n)

and denotes the application of a function f to n arguments, i.e., the application associates to the left.

The prelude also defines a right-associative application operator "`$`" which is sometimes useful to avoid brackets. Since `$` has low, right-associative binding precedence, the expression "`f $ g $ 3+4`" is equivalent to "`f (g (3+4))`".

Furthermore, the prelude also defines right-associative application operators that enforces the evaluation of the argument to a particular degree. For instance, the definition of "`$!`" is based on a predefined (infix) operator `seq` that evaluates the first argument and returns the value of the second argument:

```
($!)    :: (a -> b) -> a -> b
f $! x  = x `seq` f x
```

Thus, if `inf` is a non-terminating function (e.g., defined by "`inf = inf`") and `f` a constant function defined by "`f _ = 0`", then the evaluation of the expression "`f $! inf`" does not terminate whereas the expression "`f $ inf`" evaluates to `0`. Similarly, the operator "`$!!`" completely evaluates its second argument (i.e., to a normal form), "`$#`" evaluates its second argument to a non-variable term (by `ensureNotFree`, see Section 5.6) and suspends, if necessary, and "`$##`" completely evaluates its second argument to a term without variables.

4.1.3 Integers

The common integer values, like "42" or "-15", are considered as constructors (constants) of type `Int`. The usual operators on integers, like `+` or `*`, are predefined functions that are evaluated only if both arguments are integer values, otherwise such function calls are suspended. Thus, these functions can be used as "passive constraints" which become active after binding their arguments. For instance, if the constraint `digit` is defined by the equations

```
digit 0 = True
...
digit 9 = True
```

then the conjunction `x*x=:=y & x+x=:=y & digit x` is solved by delaying the two equations which will be activated after binding the variable `x` to a digit by the constraint `digit x`. Thus, the corresponding computed solution is

```
{x=0,y=0} | {x=2,y=4}
```

4.1.4 Floating Point Numbers

Similarly to integers, values like "3.14159" or "5.0e-4" are considered as constructors of type `Float`. Since overloading is not included in the kernel version of Curry, the names of arithmetic functions on floats are different from the corresponding functions on integers.

4.1.5 Lists

The type `[t]` denotes all lists whose elements are values of type `t`. The type of lists can be considered as predefined by the declaration

```
data [a] = [] | a : [a]
```

where `[]` denotes the empty list and `x:xs` is the non-empty list consisting of the first element `x` and the remaining list `xs`. Since it is common to denote lists with square brackets, the following convenient notation is supported:

`[`e_1`,`e_2`,...,`e_n`]`

denotes the list e_1`:`e_2`:`$\cdots$`:`e_n`:[]` (which is equivalent to e_1`:(`e_2`:(`$\cdots$`:(`e_n`:[])...)`) since ":" is a right-associative infix operator). Note that there is an overloading in the notation `[t]`: if `t` is a type, `[t]` denotes the type of lists containing elements of type `t`, where `[t]` denotes a single element list (with element `t`) if `t` is an expression. Since there is a strong distinction between occurrences of types and expressions, this overloading can always be resolved.

For instance, the following predefined functions define the concatenation of two lists and the application of a function to all elements in a list:

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

map :: (a -> b)  -> [a] -> [b]
```

```
map f []     = []
map f (x:xs) = f x : map f xs
```

4.1.6 Characters

Values like `'a'` or `'0'` denote constants of type `Char`. Special characters can be denoted with a leading backslash, e.g., `'\n'` for the character with ASCII value 10, or `'\228'` for the character `'ä'` with ASCII value 228.

There are two conversion functions between characters and their corresponding ASCII values:

```
ord :: Char -> Int
chr :: Int -> Char
```

4.1.7 Strings

The type `String` is an abbreviation for `[Char]`, i.e., strings are considered as lists of characters. String constants are enclosed in double quotes. Thus, the string constant `"Hello"` is identical to the character list `['H','e','l','l','o']`. A term can be converted into a string by the function

```
show :: a -> String
```

For instance, the result of `(show 42)` is the character list `['4','2']`.

4.1.8 Tuples

If $t_1, t_2, \ldots, t_n$ are types and $n \geq 2$, then $(t_1,t_2,\ldots,t_n)$ denotes the type of all n-tuples. The elements of type $(t_1,t_2,\ldots,t_n)$ are $(x_1,x_2,\ldots,x_n)$ where x_i is an element of type t_i $(i = 1,\ldots,n)$. Thus, for each n, the tuple-building operation $(,\ldots,)$ (with $n-1$ commas) can be considered as an n-ary constructor introduced by the pseudo-declaration

$$\texttt{data } (a_1,a_2,\ldots,a_n) \texttt{ = } (,\ldots,)\ a_1\ a_2 \ldots a_n$$

where $(x_1,x_2,\ldots,x_n)$ is equivalent to the constructor application "$(,\ldots,)\ x_1\ x_2 \ldots x_n$".

The unit type `()` has only a single element `()` and can be considered as defined by

```
data () = ()
```

Thus, the unit type can also be interpreted as the type of 0-tuples.

4.2 Type System

Curry is a strongly typed language with a Hindley/Milner-like polymorphic type system [14].[7] Each variable, constructor and operation has a unique type, where only the types of constructors have to be declared by datatype declarations (see Section 2.1). The types of functions can be declared (see Section 2.3) but can also be omitted. In the latter case they will be reconstructed by a type inference mechanism.

[7] The extension of the type system to Haskell's type classes is not included in the kernel language but may be considered in a future version.

Axiom: $\dfrac{}{\mathcal{A} \vdash x :: \tau}$ if τ is a generic instance of $\mathcal{A}(x)$

Application: $\dfrac{\mathcal{A} \vdash e_1 :: \tau_1 \to \tau_2 \quad \mathcal{A} \vdash e_2 :: \tau_1}{\mathcal{A} \vdash e_1\, e_2 :: \tau_2}$

Abstraction: $\dfrac{\mathcal{A}[x/\tau] \vdash e :: \tau'}{\mathcal{A} \vdash \backslash x\texttt{->}e :: \tau \to \tau'}$ if τ is a type expression

Existential: $\dfrac{\mathcal{A}[x/\tau] \vdash e :: \tau'}{\mathcal{A} \vdash \texttt{let}\ x\ \texttt{free in}\ e :: \tau'}$ if τ is a type expression

Figure 1: Typing rules for Curry programs

Note that Curry is an explicitly typed language, i.e., each function has a type. The type can only be omitted if the type inferencer is able to reconstruct it and to insert the missing type declaration. In particular, the type inferencer can reconstruct only those types which are visible in the module (cf. Section 6). Each type inferencer of a Curry implementation must be able to insert the types of the parameters and the free variables (cf. Section 2.5) for each rule. The automatic inference of the types of the defined functions might require further restrictions depending on the type inference method. Therefore, the following definition of a well-typed Curry program assumes that the types of all defined functions are given (either by the programmer or by the type inferencer). A Curry implementation must accept a well-typed program if all types are explicitly provided but should also support the inference of function types according to [14].

A *type expression* is either a type variable, a basic type like `Bool`, `Int`, `Float`, `Char` (or any other type constructor of arity 0), or a type constructor application of the form $(T\,\tau_1 \ldots \tau_n)$ where T is a type constructor of arity n, as defined by a datatype declaration (cf. Section 2.1),[8] and $\tau_1, \ldots, \tau_n$ are type expressions (note that list, tuple and function types have the special syntax `[`$\cdot$`]`, `(`$\cdot$`,`$\cdots$`,`$\cdot$`)`, and $\cdot$`->`$\cdot$ as described above). For instance, `[(Int,a)]->a` is a type expression containing the type variable `a`. A *type scheme* is a type expression with a universal quantification for some type variables, i.e., it has the form $\forall\alpha_1, \ldots, \alpha_n.\tau$ ($n \geq 0$; in case of $n = 0$, the type scheme is equivalent to a type expression). A function type declaration f`::`τ is considered as an assignment of the type scheme $\forall\alpha_1, \ldots, \alpha_n.\tau$ to f, where $\alpha_1, \ldots, \alpha_n$ are all type variables occurring in τ. The type τ is called a *generic instance* of the type scheme $\forall\alpha_1, \ldots, \alpha_n.\tau'$ if there is a substitution $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ on the types with $\sigma(\tau') = \tau$.

The types of all defined functions are collected in a *type environment* $\mathcal{A}$ which is a mapping from identifiers to type schemes. It contains at least the type schemes of the defined functions and an assignment of types for some local variables. An expression e is *well-typed* and has type τ w.r.t. a type environment $\mathcal{A}$ if $\mathcal{A} \vdash e :: \tau$ is derivable according to the inference rules shown in Figure 1.

A defining equation $f\ t_1 \ldots t_n$ `=` e [`where` x `free`] is well-typed w.r.t. a type environment $\mathcal{A}$ if $\mathcal{A}(f) = \forall\alpha_1, \ldots, \alpha_m.\tau$ [and $\mathcal{A}(x)$ is a type expression] and $\mathcal{A} \vdash \backslash t_1\texttt{->} \cdots \backslash t_n\texttt{->}e :: \tau$ is derivable according to the above inference rules. A conditional equation l `|` c `=` r is considered (for the purpose of typing) as syntactic sugar for l `=` c `&>` r where "`&>`" (see Section 4.1.1) has the type

[8] We assume here that all type constructors introduced by type synonyms (cf. Section 2.2) are replaced by their definitions.

scheme $\mathcal{A}(\texttt{\&>}) = \forall\alpha.\texttt{Bool} \to \alpha \to \alpha$.

A *program is well-typed* if all its rules are well-typed with a unique assignment of type schemes to defined functions.[9]

Note that the following recursive definition is a well-typed Curry program according to the definition above (and the type definitions given in the prelude, cf. Appendix B):

```
f :: [a] -> [a]
f x = if length x == 0 then fst (g x x) else x

g :: [a] -> [b] -> ([a],[b])
g x y = (f x, f y)

h :: ([Int],[Bool])
h = g [3,4] [True,False]
```

However, if the type declaration for `g` is omitted, the usual type inference algorithms are not able to infer this type.

[9] Here we assume that all local declarations are eliminated by the transformations described in Appendix D.7.

5 Expressions

Expressions are a fundamental notion of Curry. As introduced in Section 2, functions are defined by equations defining expressions that are equivalent to specific function calls. For instance, the program rule

```
square x = x*x
```

defines that the function call `(square 3)` is equivalent to the expression `(3*3)`.

Expressions occur in conditions and right-hand sides of equations defining functions. A *computation* evaluates an expression to a data term (see Section 3). Expressions are built from constants of a specific data type (e.g., integer constants, character constants, see Section 4), variables, or applications of constructors or functions to expressions. Furthermore, Curry provides some syntactic extensions for expressions that are discussed in this section.

5.1 Arithmetic Sequences

Curry supports two syntactic extensions to define list of elements in a compact way. The first one is a notation for arithmetic sequences. The *arithmetic sequence*

`[` e_1 `,` e_2 `..` e_3 `]`

denotes a list of integers starting with the first two elements e_1 and e_2 and ending with the element e_3 (where e_2 and e_3 can be omitted). The precise meaning of this notation is defined by the following translation rules:

Arithmetic sequence notation:	**Equivalent to:**
`[`e`..]`	`enumFrom` e
`[`e_1`,`e_2`..]`	`enumFromThen` e_1 e_2
`[`e_1`..`e_2`]`	`enumFromTo` e_1 e_2
`[`e_1`,`e_2`..`e_3`]`	`enumFromThenTo` e_1 e_2 e_3

The functions for generating the arithmetic sequences, `enumFrom`, `enumFromThen`, `enumFromTo`, and `enumFromThenTo`, are defined in the prelude (see page 62). Thus, the different notations have the following meaning:

- The sequence `[`e`..]` denotes the infinite list `[`e`,`e`+1,`e`+2,...]`.
- The sequence `[`e_1`..`e_2`]` denotes the finite list `[`e_1`,`e_1`+1,`e_1`+2,...,`e_2`]`. Note that the list is empty if $e_1 > e_2$.
- The sequence `[`e_1`,`e_2`..]` denotes the infinite list `[`e_1`,`e_1`+`i`,`e_1`+2*`i`,...]` with $i = e_2 - e_1$. Note that i could be positive, negative or zero.
- The sequence `[`e_1`,`e_2`..`e_3`]` denotes the finite list `[`e_1`,`e_1`+`i`,`e_1`+2*`i`,...,`e_3`]` with $i = e_2 - e_1$. Note that e_3 is not contained in this list if there is no integer m with $e_3 = e_1 + m * i$.

For instance, `[0,2..10]` denotes the list `[0,2,4,6,8,10]`.

5.2 List Comprehensions

The second compact notation for lists are *list comprehensions*. They have the general form

`[` e `|` q_1 `,\ldots,` q_k `]`

where $k \geq 1$ and each q_i is a qualifier that is either

- a *generator* of the form p`<-`l, where p is a local pattern (i.e., an expression without defined function symbols and without multiple occurrences of the same variable, compare Section 2.4) of type t and l is an expression of type `[`t`]`, or
- a *guard*, i.e., an expression of type `Bool`.

The variables introduced in a local pattern can be used in subsequent qualifiers and the element description e. Such a list comprehension denotes the list of elements which are the result of evaluating e in the environment produced by the nested and depth-first evaluation of the generators satisfying all guards. For instance, the list comprehension

```
[ x | x <- [1..50], x `mod` 7 == 0 ]
```

denotes the list `[7,14,21,28,35,42,49]`, and the list comprehension

```
[ (x,y) | x<-[1,2,3], y<-[4,5] ]
```

denotes the list `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`.

The precise meaning of list comprehensions is defined by the following translation rules (for the purpose of this definition, we also consider list comprehensions with an empty qualifier list):

List comprehension:	**Equivalent to:**
`[`e `|` `]`	`[`e`]`
`[`e `|` b`,` q`]`	`if` b `then [`e`|`q`] else []` (b Boolean guard)
`[`e `|` `let` $decls$`,` q`]`	`let` $decls$ `in [`e`|`q`]`

`[`e `|` p`<-`l`,` q`]`	`let ok` p `= [`e`|`q`]`
	`ok` p_1 `= []`
	`...`
	`ok` p_n `= []`
	`in concatMap ok l`
	where $\{p_1, \ldots, p_n\} = complPat(p)$

In the last translation rule, `concatMap` is defined in the prelude (see page 60) and $complPat(p)$ denotes the set of patterns that are incompatible with p. This set is defined by

$$complPat(x) = \{\}$$

for all variables x and

$$\begin{aligned} complPat(C\ p_1 \ldots p_n) \;=\; & \{C_1\ x_{11} \ldots x_{1n_1}, \ldots, C_k\ x_{k1} \ldots x_{kn_k}\} \\ & \cup \{C\ p_1 \ldots p_{i-1}\ p'\ y_{i+1} \ldots y_n \mid 1 \leq i \leq n,\ p' \in complPat(p_i)\} \end{aligned}$$

where $C, C_1, \ldots, C_k$ are all the constructors of the result type of C and all x_{ij}, y_i are fresh variables. For instance, $complPat($`[True]`$) = \{$`[]`, `False:zs`, `True:y:ys`$\}$ and $complPat($`(x,y)`$) = \{\}$.

Note that this translation does not work for patterns containing number constants since in this case the complement set of patterns is infinite. Therefore, we modify this scheme as follows. If the pattern p in a list comprehension `[`e `|` p`<-`l`,` q`]` contains the integer, float or character constants $c_1, \ldots, c_k$, we replace them by corresponding fresh variables $z_1, \ldots, z_k$ in the pattern and apply for this modified pattern p' the above translation scheme but with the following modified first rule for `ok`:

`ok` p' `=` `if` z_1`==`c_1 `&&`...`&&` z_k`==`c_k `then [`e`|`q`] else []`

For instance, the list comprehension

```
[x | (2,x) <- [(1,3),(2,4),(3,6)]]
```

(which is evaluated to `[4]`) is translated into the following expression:

```
let ok (y,x) = if y==2 then [x] else []
 in concatMap ok [(1,3),(2,4),(3,6)]
```

5.3 Case Expressions

Case expressions are a convenient notation of sequential pattern matching with defaults. The simplest form of a case expression is as follows ($e, e_1, \ldots, e_n$ are expressions and the patterns $p_1, \ldots, p_n$ are data terms):

`case` e `of`
  p_1 `->` e_1
  ...
  p_n `->` e_n

Note that case expressions use the layout rule (see Section C.3). Thus, the patterns $p_1, \ldots, p_n$ must be vertically aligned. The informal operational meaning of the case expression is as follows. Evaluate e so that it matches a pattern p_i. If this is possible, replace the entire case expression by the corresponding alternative e_i (after replacing the pattern variables occurring in p_i by their actual expressions). If none of the patterns $p_1, \ldots, p_n$ matches, the computation fails. The pattern matching is tried *sequentially*, from top to bottom, and rigid, i.e., without binding free variables occurring in e. In particular, the evaluation of a case expression suspends if the discriminating expression e evaluates to a free variable.

Case expressions are a convenient notation for functions with default cases. For instance, the function

```
swap z = case z of
           [x,y] -> [y,x]
           _     -> z
```

returns a list with swapped elements in case of an input list with exactly two elements and is the identity in all other cases. Moreover, it suspends if the input is a free variable. If we ignore this latter property, we can define `swap` by the following rules:

```
swap [] = []
swap [x] = [x]
swap [x,y] = [y,x]
```

```
swap (x1:x2:x3:xs) = x1:x2:x3:xs
```

The latter definition shows the improved readability obtained by case expressions.

Case expressions might also contain guards and local declarations in the alternatives. For instance, the following expression is legal:

```
case y of
  Left z
    | z >= 0    -> sqr z
    | otherwise -> - sqr z
    where sqr x = x * x
  _ -> 0
```

Guard expressions must be of type `Bool`. Case alternatives with guards have a "fall-through" semantics: if all guards of a case alternative evaluate to `False`, the match is continued with the next alternative, i.e., this is handled as if the pattern did not match at all. For instance,

```
case (1,3) of
  (x,y) | x < 0 -> (0,y)
  z -> z
```

returns the pair `(1,3)`.

5.4 Flexible Case Expressions

Similarly to the rigid pattern matching of case expressions, there is also a notation for the standard flexible pattern matching of defined functions without explicitly defining a function for it. Such a flexible case expression has the general form

`fcase` e `of`
  p_1 `|` g_1 `->` $e1$
  $\vdots$
  p_n `|` g_n `->` en

where $e, e_1, \ldots, e_n$ are expressions, the patterns $p_1, \ldots, p_n$ are data terms, and the (optional) guards $g_1, \ldots, g_n$ are expressions of type `Bool`. The operational meaning obeys the flexible pattern matching of defined functions, i.e., the alternatives do not have the "fall-through" semantics of case expressions. Actually, this expression corresponds to the expression

`let` f p_1 `|` g_1 `=` $e1$
    $\vdots$
    f p_n `|` g_n `=` en
 `in` f e

where f is a fresh auxiliary function symbol. Hence, more than one alternative can be taken during the evaluation of a flexible case expression. For instance, the expression

```
fcase () of
  _ -> False
  _ -> True
```

non-deterministically evaluates to `False` or `True`, i.e., it has the same meaning as `False ? True`.

5.5 Datatypes with Field Labels

A datatype declaration may optionally define data constructors with field labels.[10] These field labels can be used to construct, select from, and update fields in a manner that is independent of the overall structure of the datatype.

5.5.1 Declaration of Constructors with Labeled Fields

A data constructor of arity n creates an object with n components. These components are normally accessed positionally as arguments to the constructor in expressions or patterns. For large datatypes it is useful to assign *field labels* to the components of a data object. This allows a specific field to be referenced independently of its location within the constructor. A constructor definition in a data declaration may assign labels to the fields of the constructor, using the *record syntax* `C` $\{\ldots\}$. Constructors using field labels may be freely mixed with constructors without them. A constructor with associated field labels may still be used as an ordinary constructor. The various use of labels (see below) are simply a shorthand for operations using an underlying positional constructor. The arguments to the positional constructor occur in the same order as the labeled fields.

Translation:
$[\![C\ \{\ lts\ \}]\!] = C\ [\![lts]\!]$
$[\![lt,\ lts]\!] = [\![lt]\!]\ [\![lts]\!]$
$[\![l,\ ls :: t]\!] = t\ [\![ls :: t]\!]$
$[\![l :: t]\!] = t$

For example, the definition using field labels

```
data Person = Person { firstName, lastName :: String, age :: Int }
            | Agent  { firstName, lastName :: String, trueIdentity :: Person }
```

is translated to

```
data Person = Person String String Int
            | Agent  String String Person
```

A data declaration may use the same field label in multiple constructors as long as the typing of the field is the same in all cases after type synonym expansion. A label cannot be shared by more than one type in scope. Field names share the top-level name space with ordinary definition of functions and must not conflict with other top-level names in scope.

Consider the following example:

```
data S = S1 { x :: Int } | S2 { x :: Int  } -- OK
data T = T1 { y :: Int } | T2 { y :: Bool } -- BAD
```

Here S is legal but T is not, because y is given inconsistent typings in the latter.

5.5.2 Field Selection

Field labels are used as *selector functions*, i.e., each field label serves as a function that extracts the field from an object. Selectors are top-level bindings and so they may be shadowed by local

[10] Field labels are quite similar to Haskell [41] so that we adopt most of the description of Haskell here.

variables but cannot conflict with other top-level bindings of the same name. This shadowing only affects selector functions; in record construction (Section 5.5.3) and update (Section 5.5.4), field labels cannot be confused with ordinary variables.

Translation: A field label *lab* introduces a selector function defined as:
lab `(`C_1 p_{11} `...`p_{1k_1}`) = x`
`...`
lab `(`C_n p_{n1} `...`p_{nk_n}`) = x`
where $C_1 \ldots C_n$ are all the constructors of the datatype containing a field labeled with *lab*, p_{ij} is `x` when *lab* labels the jth component of C_i or `_` otherwise.

For example the definition of `Person` above introduces the selector functions

```
firstName :: Person -> String
firstName (Person x _ _) = x
firstName (Agent  x _ _) = x

lastName :: Person -> String
lastName (Person _ x _) = x
lastName (Agent  _ x _) = x

age :: Person -> Int
age (Person _ _ x) = x

trueIdentity :: Person -> Person
trueIdentity (Agent  _ _ x) = x
```

5.5.3 Construction Using Field Labels

A constructor with labeled fields may be used to construct a value in which the components are specified by name rather than by position. In this case, the components are enclosed by braces. Construction using field labels is subject to the following constraints:

- Only field labels declared with the specified constructor may be mentioned.
- A field label may not be mentioned more than once.
- Fields not mentioned are initialized to different free variables.

The expression `C{}`, where `C` is a data constructor, is legal *whether or not* `C` *was declared with record syntax*; it denotes `C Prelude.unknown`$_1$ `...` `Prelude.unknown`$_n$ where n is the arity of `C`. Note that this will introduce the constructor `C` with n *different* free variables as arguments.

Translation: In the binding $f = v$, the field f labels v.
C `{` *bs* `} = C (`$pick_1^C$ *bs* `Prelude.unknown) ... (`$pick_k^C$ *bs* `Prelude.unknown)`
where k is the arity of C.
The auxiliary function $pick_i^C$ *bs d* is defined as follows:
If the ith component of a constructor C has the field label f and $f = v$ appears in the binding list *bs*, then $pick_i^C$ *bs d* is v. Otherwise, $pick_i^C$ *bs d* is the default value *d*.

For example, a `Person` can be constructed by

```
smith = Agent { lastName = "Smith", firstName = "Agent" }
```

which is equivalent to the following agent, whose true identity might be any person:

```
smith = Agent "Agent" "Smith" _
```

5.5.4 Updates Using Field Labels

Values belonging to a datatype with field labels may be non-destructively updated. This creates a new value in which the specified field values replace those in the existing value. Updates are restricted in the following ways:

- All labels must be taken from the same datatype.
- No label may be mentioned more than once.
- The computation fails when the value being updated does not contain all of the specified labels.

Translation: Using the prior definition of *pick*,
e `{` bs `}` = `fcase` e `of`

$$\begin{aligned} & C_1\ v_1 \ldots v_{k_1}\ \texttt{->}\ C_1\ (pick_1^{C_1}\ bs\ v_1)\ \ldots\ (pick_{k_1}^{C_1}\ bs\ v_{k_1}) \\ & \quad \ldots \\ & C_j\ v_1 \ldots v_{k_j}\ \texttt{->}\ C_j\ (pick_1^{C_j}\ bs\ v_1)\ \ldots\ (pick_{k_j}^{C_j}\ bs\ v_{k_j}) \end{aligned}$$

where $\{C_1, \ldots, C_j\}$ is the set of constructors containing *all* labels in bs, k_i is the arity of C_i.

For example, after watching a few more movies, we might want to update our information about `smith`. We can do so by writing

```
smith { trueIdentity = complement neo }
```

which is equivalent to

```
fcase smith of
  Agent fn ln _ -> Agent fn ln (complement neo)
```

5.5.5 Pattern Matching Using Field Labels

A constructor with labeled fields may be used to specify a pattern in which the components are identified by name rather than by position. Matching against a constructor using labeled fields is the same as matching ordinary constructor patterns except that the fields are matched in the order they are named in the field list. All listed fields must be declared by the constructor; fields may not be named more than once. Fields not named by the pattern are ignored (matched against `_`).

Translation: Using the prior definition of *pick*,
C `{` bs `}` = `(`C `(`$pick_1^C$ bs `_)` `...` `(`$pick_k^C$ bs `_))`
where k is the arity of C.

For example, we could define a Smith-tester by writing:

```
isSmith Agent { lastName = "Smith" } = True
```

which is equivalent to

```
isSmith (Agent _ "Smith" _) = True
```

5.5.6 Field Labels and Modules

As described in Section 6, there are two forms of exporting a data type T: The simple name T exports only the types name without constructors, whereas T`(..)` also exports all constructors. Analogously, the form T does not export any field labels, whereas T`(..)` exports all constructors and all field labels.

5.6 Ensuring Instantiation

In some situations one is interested in defining functions that suspend on unknown inputs, e.g., in concurrent programming (see Appendix A.6). This behavior can be enforced by case expressions for particular user-defined datatypes. In order to support a similar definition for primitive types or types with a large number of constructors, there is a predefined primitive function

```
ensureNotFree :: a -> a
```

that evaluates the argument head normal form (i.e., to an expression without a defined function at the top) and suspends if the result is a free variable. The computation proceeds if the free variable has been instantiated to a non-variable term. In this case, the evaluated argument is returned (i.e., semantically it is the identity function).

Based on this function, one can also define more complex operations that evaluate larger parts of an expression and suspend as long as some parts are instantiated. For instance, the following function (which is defined in the standard prelude, see Appendix B), evaluates the argument to spine form, i.e., the skeleton of the list without the elements, and returns it. It suspends if the result is a non-variable spine.

```
ensureSpine :: [a] -> [a]
ensureSpine l = ensureList (ensureNotFree l)
 where ensureList []     = []
       ensureList (x:xs) = x : ensureSpine xs
```

This function is useful to model objects that process list of messages (see Appendix A.6 for an example).

6 Modules

A *module* defines a collection of datatypes, constructors and functions which we call *entities* in the following. A module exports some of its entities which can be imported and used by other modules. An entity which is not exported is not accessible from other modules.

A Curry *program* is a collection of modules. Every module named `M` (where `M` does not contain a dot, see below) must be stored in a file named `M.curry`. There is one main module which is loaded into a Curry system. The modules imported by the main module are implicitly loaded but not visible to the user. After loading the main module, the user can evaluate expressions which contain entities exported by the main module.

There is one distinguished module, named `Prelude`, (see Appendix B) which is implicitly imported into each module provided that this module does not contain an explicit import declaration for the module `Prelude`. Thus, the entities defined in the standard prelude (basic functions for arithmetic, list processing etc.) can be always used (if they are not hidden by a `hiding` declaration, see below).

A module always starts with the head which contains at least the name of the module followed by the keyword `where`, like

```
module Stack where ...
```

If a module stored in a directory in file `M.curry` does not contain a module head, the *standard module head* "`module M where`" is implicitly inserted. *Module names* can be given a hierarchical structure by inserting dots which is useful if larger applications should be structured into different subprojects. For instance,

```
company.project1.subproject2.Mod4
```

is a valid module name. The dots may reflect the hierarchical file structure where modules are stored. For instance, the module `compiler.symboltable` could be stored in the file `symboltable.curry` in the directory `compiler`. To avoid such long module names when referring to entities of this module by qualification, imported modules can be renamed by the use of "`as`" in an import declaration (see below).

Without any further restrictions in the module head, all entities which are defined by top-level declarations in this module are exported (but not those entities that are imported into this module). In order to modify this standard set of exported entities of a module, an *export list* can be added to the module head. For instance, a module with the head

```
module Stack(stackType, push, pop, newStack) where ...
```

exports the entities `stackType`, `push`, `pop`, and `newStack`. An export list can contain the following entries:

1. *Names of datatypes*: This exports only the datatype, whose name must be accessible in this module, *but not* the constructors of the datatype. The export of a datatype without its constructors allows the definition of *abstract datatypes*.

2. *Datatypes with all constructors*: If the export list contains the entry `t(..)`, then `t` must be a datatype whose name is in the module. In this case, the datatype `t` and all constructors of this datatype, which must be also accessible in this module, are exported.

3. *Names of functions*: This exports the corresponding functions whose names must be accessible in this module. The types occurring in the argument and result type of this function are implicitly exported, otherwise the function may not be applicable outside this module.

4. *Modules*: The set of all accessible entities imported from a module M into the current module (see below) can be exported by a single entry "`module` M" in the export list. For instance, if the head of the module `Stack` is defined as above, the module head

   ```
   module Queue(module Stack, enqueue, dequeue) where ...
   ```

   specifies that the module `queue` exports the entities `stackType`, `push`, `pop`, `newStack`, `enqueue`, and `dequeue`.

The unqualified names of the exported entities of a module must be pairwise different to avoid name clashes in the use of these entities. For instance, the module

```
module M(f,Ma.g) where
import Ma
f x = Ma.g x
```

exports the names `M.f` and `M.g`, i.e., a *qualified entity* consists always of the name of the exported module followed by a dot and the unqualified name of the entity. Spaces are not allowed between the dot and the module/function name. Thus, an identifier immediately followed by a dot and a variable or infix operator identifier is always interpreted as a qualified name.

If module `Ma` also exports the entity `f`, then the export declaration

```
module M(f,Ma.f) where
import Ma
f x = Ma.f x
```

is not allowed since the exported name `M.f` cannot be uniquely resolved.

All entities defined by top-level declarations in a module are *always accessible* in this module, i.e., there is no need to qualify the names of top-level declarations. However, they can be qualified for the sake of readability, i.e., inside module `M`, it is always possible to refer to an entity `f` defined at the top level of this module with the qualified name `M.f`. Additionally, the entities exported by another module can be also made accessible in the module by an `import` declaration. An import declaration consists of the name of the imported module and (optionally) a list of entities imported from that module. If the list of imported entities is omitted, all entities exported by that module are imported. For instance, the import declaration

```
import Stack
```

imports all entities exported by the module `Stack`, whereas the declaration

```
import Family(father, grandfather)
```

imports only the entities `father` and `grandfather` from the module `Family`, provided that they are exported by `Family`. Similarly to export declarations, a datatype name `t` in an import list imports only the datatype without its constructors whereas the form `t(..)` imports the datatype together with all its constructors (provided that they are also exported).

The names of all imported entities are accessible in the current module, i.e., they are equivalent to top-level declarations, provided that their names are not in conflict with other names. For instance, if a function `f` is imported from module `M` but the current module contains a top-level declaration for `f` (which is thus directly accessible in the current module), the imported function is not accessible (without qualification). Similarly, if two identical names are imported from different modules and denote different entities, none of these entities is accessible (without qualification). It is possible to refer to such imported but not directly accessible names by prefixing them with the module identifier (*qualification*). For instance, consider the module `M1` defined by

```
module M1 where
f :: Int -> Int
...
```

and the module `m2` defined by

```
module M2 where
f :: Int -> Int
...
```

together with the main module

```
module Main where
import M1
import M2
...
```

Then the names of the imported functions `f` are not directly accessible in the main module but one can refer to the corresponding imported entities by the qualified identifiers `M1.f` or `M2.f`.

Another method to resolve name conflicts between imported entities is the qualification of an imported module. If we change the `Main` module to

```
module Main where
import qualified M1
import M2
...
```

then the name `f` refers to the entity `M2.f` since all entities imported from `M1` are *only accessible by qualification* like `M1.f`.

A further method to avoid name conflicts is the hiding of imported entities. Consider the following definition:

```
module Main where
import M1 hiding (f)
import M2
...
```

The name `f` in the `Main` module refers to the entity `M2.f` since the name `f` is not imported from `M1` by the hiding declaration. The hiding clause effects only unqualified names, i.e., the entity `M1.f` is still accessible in the body of the `Main` module. Therefore, a hiding clause has no effect in combination with a `qualified` import. Similarly to export declarations, a datatype `t` in a hiding clause hides only the datatype (but not its constructors) whereas the form `t(..)` hides the complete datatype

including its constructors.

The effect of several `import` declarations is cumulative, i.e., if an entity is hidden in one import declaration, it can still be imported by another import declaration. For instance, if module `Mt` exports a datatype `t` together with its constructors, then the import declarations

```
import Mt hiding (t(..))
import Mt (t)
```

imports all entities exported by `Mt` but only the name `t` of the datatype without its constructors, since the first hiding clause imports everything from `Mt` except the complete datatype `t` and the second import specification additionally imports the name of the datatype `t`.

Imported modules can also be given a new local name in the import declaration. For instance, the declaration

```
import M(f) as foo
```

enables access to the name `f` (provided that it is not in conflict with another entity with the same name) and `foo.f` but not to `M.f`. This local renaming enables the abbreviation of long module names and the substitution of different modules without changing the qualifiers inside a module.

Although each name refers to exactly one entity, it is possible that the same entity is referred by different names. For instance, consider the modules defined by

```
module M(f) where
f :: Int -> Int
...
module M1(M.f) where
import M
...
module M2(M.f) where
import M
...
```

together with the main module

```
module Main where
import M1
import M2
...
```

Now the names `f`, `M1.f`, and `M2.f` refer to the identical entity, namely the function `f` defined in module `M`. Note that there is no need to qualify `f` in the module `Main` since this name is unambiguously resolved to the function `f` defined in module `M`, although it is imported via two different paths.

Qualified names are treated syntactically like unqualified names. In particular, a qualified infix operator like `Complex.+` has the same fixity as the definition of `+` in the module `Complex`, i.e., the expression "`x Complex.+ y`" is syntactically valid. To distinguish between a function composition like "`f . g`", where "." is an infix operator (see Section 20, page 54), and a module qualification, spaces are not allowed in the module qualification while there must be at least one space before the "." if it is used as an infix operator. Thus, "`f . g`" is interpreted as the composition of `f` and `g` whereas "`f.g`" is interpreted as the object `g` imported from module `f`.

The import dependencies between modules must be *non-circular*, i.e., it is not allowed that module m_1 imports module m_2 and module m_2 also imports (directly or indirectly) module m_1.

7 Input/Output

Curry provides a declarative model of I/O by considering I/O operations as transformations on the outside world. In order to avoid dealing with different versions of the outside world, it must be ensured that at each point of a computation only one version of the world is accessible. This is guaranteed by using the monadic I/O approach [42] of Haskell and by requiring that I/O operations are not allowed in program parts where non-deterministic search is possible.

7.1 Monadic I/O

In the monadic I/O approach, the outside "world" is not directly accessible but only through actions which change the world. Thus, the world is encapsulated in an abstract datatype which provides actions to change the world. The type of such actions is `IO t` which is an abbreviation for

```
World -> (t,World)
```

where `World` denotes the type of all states of the outside world. If an action of type `IO t` is applied to a particular world, it yields a value of type `t` and a new (changed) world. For instance, `getChar` is an action which reads a character from the standard input when it is executed, i.e., applied to a world. Therefore, `getChar` has the type `IO Char`. The important point is that values of type `World` are not accessible for the programmer—she/he can only create and compose actions on the world. Thus, a program intended to perform I/O operations has a sequence of actions as the result. These actions are computed and executed when the program is connected to the world by executing it. For instance,

```
getChar :: IO Char
getLine :: IO String
```

are actions which read the next character or the next line from the standard input. The functions

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

take a character (string) and produces an action which, when applied to a world, puts this character (string) to the standard output (and a line-feed in case of `putStrLn`.

Since an interactive program consists of a sequence of I/O operations, where the order in the sequence is important, there are two operations to compose actions in a particular order. The function

```
(>>) :: IO a -> IO b -> IO b
```

takes two actions as input and yields an action as output. The output action consists of performing the first action followed by the second action, where the produced value of the first action is ignored. If the value of the first action should be taken into account before performing the second action, the actions can be composed by the function

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

where the second argument is a function taking the value produced by the first action as input and performs another action. For instance, the action

```
getLine >>= putStrLn
```

is of type `IO ()` and copies, when executed, a line from standard input to standard output.

The `return` function

```
return :: a -> IO a
```

is sometimes useful to terminate a sequence of actions and return a result of an I/O operation. Thus, `return v` is an action which does not change the world and returns the value `v`.

To execute an action, it must be the main expression in a program, i.e., interactive programs have type `IO ()`. Since the world cannot be copied (note that the world contains at least the complete file system), non-determinism in relation with I/O operations must be avoided. Thus, the applied action must always be known, i.e., `>>` and `>>=` suspend if the arguments are free variables. Moreover, it is a runtime error if a disjunctive expression (cf. Section 3) $\sigma_1 \,[\!]\, e_1 \;|\cdots|\; \sigma_n \,[\!]\, e_n$, where the e_i's are of type `IO ()` and $n > 1$, occurs as the top-level expression of a program, since it is unclear in this case which of the disjunctive actions should be applied to the current world. Thus, all possible search must be encapsulated between I/O operations (see Section 8). The compiler may warn the user about non-deterministic computations which may occur in I/O actions so that the programmer can encapsulate them.

7.2 Do Notation

To provide a more conventional notation for programming sequences of I/O operations, Curry has a special piece of syntax for writing such sequences. For instance, consider the following expression to read a line form the standard input and to print them together with the string `"Your input: "` on the standard output:

```
getLine >>= \line -> putStr "Your input: " >> putStrLn line
```

Using the do notation, this expression can be written in a more traditional style:

```
do line <- getLine
   putStr "Your input: "
   putStrLn line
```

Note that the do notation is just another syntax for sequences of I/O actions. Thus, `do` expressions can be used wherever an expression of type I/O is required (but note that `do` expressions use the layout rule (see Section C.3) and, therefore, they are usually written in separate program lines). The following table specifies the translation of `do` expressions into the kernel language:

Do notation:	**Equivalent to:**
`do` *expr*	*expr*
`do` *expr* *stmts*	*expr* `>>` `do` *stmts*
`do` *p* `<-` *expr* *stmts*	*expr* `>>=` `\`*p* `->` `do` *stmts*
`do let` *decls* *stmts*	`let` *decls* `in` `do` *stmts*

8 Encapsulated Search

Global search, possibly implemented by backtracking, must be avoided in some situations (user-control of efficiency, concurrent computations, non-backtrackable I/O). Hence it is sometimes necessary to encapsulate search, i.e., non-deterministic computations in parts of larger programs. Non-deterministic computations might occur in Curry whenever a function must be evaluated with a free variable at a flexible argument position. In this case, the computation must follow different branches with different bindings applied to the current expression which has the effect that the entire expression is split into (at least) two independent disjunctions. To give the programmer control on the actions taken in this situation, Curry provides a primitive search operator which is the basis to implement sophisticated search strategies. This section sketches the idea of encapsulating search in Curry and describes some predefined search strategies.

8.1 Search Goals and Search Operators

Since search is used to find solutions to some constraint, search is always initiated by a constraint containing a *search variable* for which a solution should be computed.[11] Since the search variable may be bound to different solutions in different disjunctive branches, it must be abstracted. Therefore, a *search goal* has the type `a->Bool` where `a` is the type of the values which we are searching for. In particular, if c is a constraint containing a variable x and we are interested in solutions for x, i.e., values for x such that c is satisfied, then the corresponding search goal has the form `(\`x`->`c`)`. However, any other expression of the same type can also be used as a search goal.

To control the search performed to find solutions to search goals, Curry has a predefined operator

```
try :: (a->Bool) -> [a->Bool]
```

which takes a search goal and produces a list of search goals. The search operator `try` attempts to evaluate the search goal until the computation finishes or performs a non-deterministic splitting. In the latter case, the computation is stopped and the different search goals caused by this splitting are returned as the result list. Thus, an expression of the form `try (\`x`->`c`)` can have three possible results:

1. An empty list. This indicates that the search goal `(\`x`->`c`)` has no solution. For instance, the expression

   ```
   try (\x -> 1=:=2)
   ```

   reduces to `[]`.

2. A list containing a single element. In this case, the search goal `(\`x`->`c`)` has a single solution represented by the element of this list. For instance, the expression

   ```
   try (\x->[x]=:=[0])
   ```

   reduces to `[\x->x=:=0]`. Note that a solution, i.e., a binding for the search variable like the substitution $\{\texttt{x} \mapsto \texttt{0}\}$, can always be presented by an equational constraint like `x=:=0`.

[11] The generalization to more than one search variable is straightforward by the use of tuples.

Generally, a one-element list as a result of `try` has always the form `[\`x`->`x`=:=`e`]` (plus some local variables, see next subsection) where e is fully evaluated, i.e., e does not contain defined functions. Otherwise, this goal might not be solvable due to the definition of equational constraints.

3. A list containing more than one element. In this case, the evaluation of the search goal `(\`x`->`c`)` requires a non-deterministic computation step. The different alternatives immediately after this non-deterministic step are represented as elements of this list. For instance, if the function `f` is defined by

   ```
   f a = c
   f b = d
   ```

   then the expression

   ```
   try (\x -> f x =:= d)
   ```

   reduces to the list `[\x->x=:=a & f a =:= d, \x->x=:=b & f b =:= d]`. This example also shows why the search variable must be abstracted: the alternative bindings cannot be actually performed (since a free variable is only bound to at most one value in each computation thread) but are represented as equational constraints in the search goal.

Note that the search goals of the list in the last example are not further evaluated. This provides the possibility to determine the behavior for non-deterministic computations. For instance, the following function defines a depth-first search operator which collects all solutions of a search goal in a list:

```
solveAll :: (a->Bool) -> [a->Bool]
solveAll g = collect (try g)
      where collect []         = []
            collect [g]        = [g]
            collect (g1:g2:gs) = concat (map solveAll (g1:g2:gs))
```

(`concat` concatenates a list of lists to a single list). For instance, if `append` is the list concatenation, then the expression

```
solveAll (\l -> append l [1] =:= [0,1])
```

reduces to `[\l->l=:=[0]]`.

The value computed for the search variable in a search goal can be easily accessed by applying it to a free variable. For instance, the evaluation of the applicative expression

```
solveAll (\l->append l [1] =:= [0,1]) =:= [g] & g x
```

binds the variable `g` to the search goal `[\l->l=:=[0]]` and the variable `x` to the value `[0]` (due to solving the constraint `x=:=[0]`). Based on this idea, there is a predefined function

```
findall :: (a->Bool) -> [a]
```

which takes a search goal and collects all solutions (computed by a depth-first search like `solveAll`) for the search variable into a list.

Due to the laziness of the language, search goals with infinitely many solutions cause no problems if one is interested only in finitely many solutions. For instance, a function which computes only the first solution w.r.t. a depth-first search strategy can be defined by

```
first g = head (findall g)
```

Note that `first` is a partial function, i.e., it is undefined if `g` has no solution.

8.2 Local Variables

Some care is necessary if free variables occur in the search goal, like in the goal

```
\l2 -> append l1 l2 =:= [0,1]
```

Here, only the variable `l2` is abstracted in the search goal but `l1` is free. Since non-deterministic bindings cannot be performed during encapsulated search, *free variables are never bound inside encapsulated search*. Thus, if it is necessary to bind a free variable in order to proceed an encapsulated search operation, the computation suspends. For instance, the expression

```
first (\l2 -> append l1 l2 =:= [0,1])
```

cannot be evaluated and will be suspended until the variable `l1` is bound by another part of the computation. Thus, the constraint

```
s =:= first (\l2->append l1 l2 =:= [0,1])  &  l1 =:= [0]
```

can be evaluated since the free variable `l1` in the search goal is bound to `[0]`, i.e., this constraint reduces to the answer

```
{l1=[0], s=[1]}
```

In some cases it is reasonable to have unbound variables in search goals, but these variables should be treated as local, i.e., they might have different bindings in different branches of the search. For instance, if we want to compute the last element of the list `[3,4,5]` based on `append`, we could try to solve the search goal

```
\e -> append l [e] =:= [3,4,5]
```

However, the evaluation of this goal suspends due to the necessary binding of the free variable `l`. This can be avoided by declaring the variable `l` as *local* to the constraint by the use of `let` (see Section 2.5), like in the following expression:

```
first (\e -> let l free in append l [e] =:= [3,4,5])
```

Due to the local declaration of the variable `l` (which corresponds logically to an existential quantification), the variable `l` is only visible inside the constraint and, therefore, it can be bound to different values in different branches. Hence this expression evaluates to the result `5`.

In order to ensure that an encapsulated search will not be suspended due to necessary bindings of free variables, the search goal should be a closed expression when a search operator is applied to it, i.e., the search variable is bound by the lambda abstraction and all other free variables are existentially quantified by local declarations.

8.3 Predefined Search Operators

There are a number of search operators which are predefined in the prelude. All these operators are based on the primitive `try` as described above. It is also possible to define other search strategies in a similar way. Thus, the `try` operator is a a powerful primitive to define appropriate search strategies. In the following, we list the predefined search operators.

`solveAll :: (a->Bool) -> [a->Bool]`
: Compute all solutions for a search goal via a depth-first search strategy. If there is no solution and the search space is finite, the result is the empty list, otherwise the list contains solved search goals (i.e., without defined operations).

`once :: (a->Bool) -> (a->Bool)`
: Compute the first solution for a search goal via a depth-first search strategy. Note that `once` is partially defined, i.e., if there is no solution and the search space is finite, the result is undefined.

`findall :: (a->Bool) -> [a]`
: Compute all solutions for a search goal via a depth-first search strategy and unpack the solution's values for the search variable into a list.

`best :: (a->Bool) -> (a->a->Bool) -> [a->Bool]`
: Compute the best solution via a depth-first search strategy, according to a specified relation (the second argument) that can always take a decision which of two solutions is better (the relation should deliver `True` if the first argument is a better solution than the second argument).

  As a trivial example, consider the relation `shorter` defined by

```
shorter l1 l2 = length l1 <= length l2
```

  Then the expression

```
best (\x -> let y free in append x y =:= [1,2,3]) shorter
```

  computes the shortest list which can be obtained by splitting the list `[1,2,3]` into this list and some other list, i.e., it reduces to `[\x->x=:=[]]`. Similarly, the expression

```
best (\x -> let y free in append x y =:= [1,2,3])
     (\l1 l2 -> length l1 > length l2)
```

  reduces to `[\x->x=:=[1,2,3]]`.

`browse :: (a->Bool) -> IO ()`
: Show the solution of a *solved* constraint on the standard output, i.e., a call `browse g`, where `g` is a solved search goal, is evaluated to an I/O action which prints the solution. If `g` is not an abstraction of a solved constraint, a call `browse g` produces a runtime error.

`browseList :: [a->Bool] -> IO ()`
: Similar to `browse` but shows a list of solutions on the standard output. The `browse` operators are useful for testing search strategies. For instance, the evaluation of the expression

```
browseList (solveAll (\x -> let y free in append x y =:= [0,1,2]))
```

produces the following result on the standard output:

```
[]
[0]
[0,1]
[0,1,2]
```

Due to the laziness of the evaluation strategy, one can also browse the solutions of a goal with infinitely many solutions which are printed on the standard output until the process is stopped.

9 Interface to External Functions and Constraint Solvers

Since Curry has only a small number of builtins, it provides a simple interface to connect external functions (functions programmed in another language) and external constraint solvers. External functions must be free of side-effects in order to be compatible with Curry's computation model. An external constraint solver consists of a constraint store which can be accessed and manipulated via a few primitive operations.

9.1 External Functions

External functions are considered as an implementation of a potentially infinite set of equations (describing the graph of the functions). In particular, they have no side effects, i.e., identical function calls at different points of time yield identical results. Since the implementation of an external function is unknown, a call to an external function is suspended until all arguments are fully known, i.e., until they are evaluated to ground data terms. This view is compatible with the residuation principle covered by Curry's computation model and very similar to the connection of external functions to logic programs [10, 11].

An external function is declared by a type declaration followed by `external(FName,OF,Lang)` where `FName` is the name of the external function and its code is contained in the object file `OF`. `Lang` is the implementation language of the external function. For instance, if the addition and multiplication on integers are defined as C functions named `add` and `mult` where the compiled code is contained in the file "`arith.o`", we can connect these functions to a Curry program by providing the declarations

```
add  :: Int -> Int -> Int  external("add","arith.o",C)
mult :: Int -> Int -> Int  external("mult","arith.o",C)
```

Implementations of Curry might have restrictions on the interface. A reasonable requirement is that the implementation language is constrained to C and the argument and result types are only simple types like `Bool`, `Int`, `Float`, `Char`, or `String`.

9.2 External Constraint Solvers

A constraint solver can be viewed as an abstract datatype consisting of a constraint store together with a few operations to check the entailment of a constraint or to add a new constraint. In order to connect a constraint solver to Curry, like a solver for arithmetic or Boolean constraints, the external solver must provide the following operations (cs denotes the type of the constraint store and c the type of constraints):

new	:		$\rightarrow\ cs$	(*create and initialize a new constraint store*)
ask	:	$cs \times c$	$\rightarrow\ \{True, False, Unknown\}$	(*check entailment of a constraint*)
$tell$	:	$cs \times c$	$\rightarrow\ cs$	(*add a new constraint*)
$clone$	:	cs	$\rightarrow\ cs \times cs$	(*copy the constraint store*)

Using these operations, it is relatively easy to extend Curry's computation model to include constraints by adding the constraint store as a new component to the substitution part in answer expressions (cf. Section 3).[12]

[12] This section will be modified or extended in a later version.

10 Literate Programming

To encourage the use of comments for the documentation of programs, Curry supports a literate programming style (inspired by Donald Knuth's "literate programming") where comments are the default case. Non-comment lines containing program code must start with "`>`" followed by a blank. Using this style, we can define the concatenation and reverse function on lists by the following literate program:

```
The following function defines the concatenation of two lists.
Note that this function is predefined as `++' in the standard prelude.

> append :: [a] -> [a] -> [a]
> append []     x  = x
> append (x:xs) ys = x : append xs ys

As a second example for list operations, we define a
function to reverse the order of elements in a list:

> rev :: [a] -> [a]
> rev []     = []
> rev (x:xs) = append (rev xs) [x]
```

To distinguish literature from non-literate programs, literate programs must be stored in files with the extension "`.lcurry`" whereas non-literate programs are stored in files with the extension "`.curry`".

11 Interactive Programming Environment

In order to support different implementations with a comparable user interface, the following commands should be supported by each interactive programming environment for Curry (these commands can also be abbreviated to `:`c where c is an unambiguous prefix string of the command):

`:load file` Load the program stored in `file.curry`.

`:reload` Repeat the last load command.

`expr` Evaluate the expression `expr` and show all computed results. Since an expression could be evaluated to a disjunctive expression (cf. Section 3), the expression could be automatically wrapped in some search operator like depth-first search or a fair breadth-first search, depending on the implementation.

`:debug expr` Debug the evaluation of the expression `expr`, i.e., show the single computation steps and ask the user what to do after each single step (like proceed, abort, etc.).

`:type expr` Show the type of the expression `expr`.

`:quit` Exit the system.

A Example Programs

This section contains a few example programs together with some initial expressions and their computed results.

A.1 Operations on Lists

Here are simple operations on lists. Note that, due to the logic programming features of Curry, `append` can be used to split lists. We exploit this property to define the last element of a list in a very simple way.

```
-- Concatenation of two lists:
append :: [t] -> [t] -> [t]
append []     ys = ys
append (x:xs) ys = x:append xs ys

-- Naive reverse of all list elements:
rev :: [t] -> [t]
rev []     = []
rev (x:xs) = append (rev xs) [x]

-- Last element of a list:
last :: [t] -> t
last xs | append _ [x] =:= xs  = x  where x free
```

Expressions and their evaluated results:

```
append [0,1] [2,3]   ⇝   [0,1,2,3]

append l m =:= [0,1]
  ⇝   {l=[],m=[0,1]} | {l=[0],m=[1]} | {l=[0,1],m=[]}

rev [0,1,2,3]   ⇝   [3,2,1,0]

last (append [1,2] [3,4])   ⇝   4
```

A.2 Higher-Order Functions

Here are some "traditional" higher-order functions to show that the familiar functional programming techniques can be applied in Curry. Note that the functions `map`, `foldr`, and `filter` are predefined in Curry (see Appendix B).

```
-- Map a function on a list (predefined):
map :: (t1->t2) -> [t1] -> [t2]
map f []     = []
map f (x:xs) = f x : map f xs

-- Fold a list (predefined):
foldr :: (t1->t2->t2) -> t2 -> [t1] -> t2
foldr f a []     = a
foldr f a (x:xs) = f x (foldr f a xs)

-- Filter elements in a list (predefined):
filter :: (t -> Bool) -> [t] -> [t]
filter p []     = []
filter p (x:xs) = if p x then x : filter p xs
                         else filter p xs

-- Quicksort function:
quicksort :: [Int] -> [Int]
quicksort []      = []
quicksort (x:xs)  = quicksort (filter (<= x) xs)
                    ++ [x]
                    ++ quicksort (filter (> x) xs)
```

A.3 Relational Programming

Here is a traditional example from logic programming: a simple deductive database with family relationships. We use a relational programming style, i.e., all relationships are represented as predicates.

```
-- Declaration of an enumeration type for persons:
-- (as an alternative, one could consider persons as strings)

data Person = Christine | Maria | Monica | Alice | Susan |
              Antony | Bill | John | Frank | Peter | Andrew

-- Two basic relationships:

married :: Person -> Person -> Bool
married Christine Antony  = True
married Maria     Bill    = True
married Monica    John    = True
married Alice     Frank   = True

mother :: Person -> Person -> Bool
mother Christine John   = True
mother Christine Alice  = True
mother Maria     Frank  = True
mother Monica    Susan  = True
mother Monica    Peter  = True
mother Alice     Andrew = True

-- and here are the deduced relationships:
father :: Person -> Person -> Bool
father f c | married m f & mother m c = True  where m free

grandfather :: Person -> Person -> Bool
grandfather g c | father g f && father f c  = True  where f free
grandfather g c | father g m && mother m c  = True  where m free
```

Expressions and their evaluated results:

```
father John child   ⇝   {child=Susan} | {child=Peter}

grandfather g c   ⇝
{g=Antony,c=Susan} | {g=Antony,c=Peter} |
{g=Bill,c=Andrew}  | {g=Antony,c=Andrew}
```

A.4 Functional Logic Programming

This is the same example as in the previous section. However, we use here a functional logic programming style which is more readable but provides the same goal solving capabilities. The basic functions are `husband` and `mother` which express the functional dependencies between the different persons. Note that the derived function `grandfather` is a non-deterministic function which yields all grandfathers for a given person.

```
data Person = Christine | Maria | Monica | Alice | Susan |
              Antony | Bill | John | Frank | Peter | Andrew

-- Two basic functional dependencies:

husband :: Person -> Person
husband Christine  = Antony
husband Maria      = Bill
husband Monica     = John
husband Alice      = Frank

mother :: Person -> Person
mother John    = Christine
mother Alice   = Christine
mother Frank   = Maria
mother Susan   = Monica
mother Peter   = Monica
mother Andrew  = Alice

-- and here are the deduced functions and relationships:
father :: Person -> Person
father c  = husband (mother c)

grandfather :: Person -> Person
grandfather c = father (father c)
grandfather c = father (mother c)
```

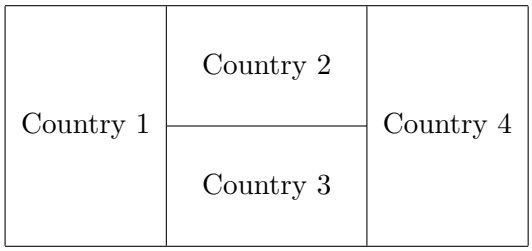
Expressions and their evaluated results:

```
solve $ father child == John   ⇝   {child=Susan} | {child=Peter}

grandfather c   ⇝
{c=Susan} Antony | {c=Peter} Antony | {c=Andrew} Bill | {c=Andrew} Antony
```

A.5 Constraint Solving and Concurrent Programming

In this example we demonstrate how Curry's concurrent features can be used to solve constraint problems with finite domains more efficiently than a simple generate-and-test solver. We want to solve the classical map coloring problem. Consider the following simple map:



The problem is to assign to each of the four countries a color (red, green, yellow, or blue) such that countries with a common border have different colors.

There is a straightforward solution to the map coloring problem. We define a constraint `coloring` specifying the valid colors for each country and a constraint `correct` specifying which countries must have different colors:

```
data Color = Red | Green | Yellow | Blue

isColor :: Color -> Bool
isColor Red    = True
isColor Yellow = True
isColor Green  = True
isColor Blue   = True

coloring :: Color -> Color -> Color -> Color -> Bool
coloring l1 l2 l3 l4 = isColor l1 & isColor l2 & isColor l3 & isColor l4

correct :: Color -> Color -> Color -> Color -> Bool
correct l1 l2 l3 l4 = l1 /= l2 & l1 /= l3 & l2 /= l3 & l2 /= l4 & l3 /= l4
```

As in classical logic programming, we can compute the solutions to the map coloring problem by enumerating all potential solutions followed by a check whether a potential solution is a correct one ("generate and test"). This can be expressed by solving the following goal:

```
coloring l1 l2 l3 l4 && correct l1 l2 l3 l4
```

A.6 Concurrent Object-Oriented Programming

The following example shows a simple method to program in a concurrent object-oriented style in Curry. For this purpose, an object is a process waiting for incoming messages. The local state is a parameter of this process. Thus, a process is a function of type

```
State -> [Message] -> Bool
```

In the subsequent example, we implement a bank account as an object waiting for messages of the form `Deposit i`, `Withdraw i`, or `Balance i`. Thus, the bank account can be defined as follows:

```
data Message = Deposit Int | Withdraw Int | Balance Int

account :: Int -> [Message] -> Bool
account _ []                 =  True
account n (Deposit  a : ms)  =  account (n+a) ms
account n (Withdraw a : ms)  =  account (n-a) ms
account n (Balance  b : ms)  =  b=:=n & account n ms

-- Install an initial account with message stream s:
makeAccount s = account 0 (ensureSpine s)
```

A new account object is created by the constraint "`makeAccount s`" where `s` is a free variable. The function `ensureSpine` (see Section 5.6) used in the definition of `makeAccount` ensures that the evaluation of the function call to `account` suspends on uninstantiated parts of the message list

When `s` is instantiated with messages, the account object starts processing these messages. The following concurrent conjunction of constraints creates an account and sends messages:

```
makeAccount s & s=:=[Deposit 200, Deposit 50, Balance b]
```

After this goal is solved, the free variable `b` has been bound to `250` representing the balance after the two deposits.

To show a client-server interaction, we define a client of a bank account who is finished if his account is `50`, buys things for `30` if his account is greater than `50`, and works for an income of `70` if his account is less than `50`. In order to get the client program independent of the account processing, the client sends messages to his account. Therefore, the client is implemented as follows:

```
-- Send a message to an object identified by its message stream obj:
sendMsg :: msg -> [msg] -> [msg]
sendMsg msg obj | obj =:= msg:obj1  = obj1  where obj1 free

-- Client process of an bank account
client s | s1 =:= sendMsg (Balance b) s  =
  if b==50 then s1=:=[]                                          -- stop
           else if b>50 then client (sendMsg (Withdraw 30) s1)   -- buy
                        else client (sendMsg (Deposit  70) s1)   -- work
  where s1,b free
```

We start the account and client process with an initial amount of `100`:

```
makeAccount s & client (sendMsg (Deposit 100) s)
```

A Curry system evaluates this goal by a concurrent evaluation of both processes and computes the final answer

```
{s=[Deposit 100, Balance 100, Withdraw 30, Balance 70, Withdraw 30,
    Balance 40, Deposit 70, Balance 110, Withdraw 30, Balance 80,
    Withdraw 30, Balance 50]}
```

which shows the different messages sent to the account process.

B Standard Prelude

This section contains the standard prelude for Curry programs. The module `Prelude` will be imported into every Curry module unless the module has an explicit import declaration for the module `Prelude` (for instance, in order to hide some names from the standard prelude). Note that some of the definitions shown below are syntactically not allowed (e.g., datatypes for tuples). They are only provided for documentation purposes.

```
module Prelude where

-- Infix operator declarations

infixl 9 !!
infixr 9 .
infixl 7 *, /, `div`, `mod`
infixl 6 +, -
infixr 5 ++, :
infix  4 =:=, ==, /=, <, >, <=, >=
infix  4  `elem`, `notElem`
infixr 3 &&
infixr 2 ||
infixl 1 >>, >>=
infixr 0 $, $!, $!!, $#, $##, `seq`, &, &>, ?


-- Some standard combinators:

-- Function composition
(.)             :: (b -> c) -> (a -> b) -> (a -> c)
f . g           = \x -> f (g x)

-- Identity
id              :: a -> a
id x            = x

-- Constant function
const           :: a -> b -> a
const x _       = x

-- Convert an uncurried function to a curried function
curry           :: ((a,b) -> c) -> a -> b -> c
curry f a b     =  f (a,b)

-- Convert an curried function to a function on pairs
uncurry         :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

-- (flip f) is identical to f but with the order of arguments reversed
flip            :: (a -> b -> c) -> b -> a -> c
```

```
flip  f x y     = f y x

-- Repeat application of a function until a predicate holds
until          :: (a -> Bool) -> (a -> a) -> a -> a
until p f x     = if p x then x else until p f (f x)

-- Evaluate the first argument to head normal form and return the
-- second argument.
seq     :: a -> b -> b

-- Evaluate the argument to head normal form and returns it.
-- Suspend until the result is bound to a non-variable term.
ensureNotFree :: a -> a

--- Evaluate the argument to spine form and returns it.
--- Suspend until the result is bound to a non-variable spine.
ensureSpine :: [a] -> [a]
ensureSpine l = ensureList (ensureNotFree l)
 where ensureList []     = []
       ensureList (x:xs) = x : ensureSpine xs

-- Right-associative application
($)     :: (a -> b) -> a -> b
f $ x   = f x

-- Right-associative application with strict evaluation of its argument.
($!)    :: (a -> b) -> a -> b
f $! x  = x `seq` f x

-- Right-associative application with strict evaluation of its argument
-- to normal form.
($!!)   :: (a -> b) -> a -> b
f $!! x | x=:=y = f y  where y free

-- Right-associative application with strict evaluation of its argument
-- to a non-variable term.
($#)    :: (a -> b) -> a -> b
f $# x  = f $! (ensureNotFree x)

-- Right-associative application with strict evaluation of its argument
-- to ground normal form.
($##)   :: (a -> b) -> a -> b

-- Abort the execution with an error message
error :: String -> _

-- failed is a non-reducible polymorphic function.
-- It is useful to express a failure in a search branch of the execution.
-- It could be defined by: failed = head []
```

```
failed :: _


-- Boolean values

data Bool = False | True

-- Sequential conjunction
(&&)                    :: Bool -> Bool -> Bool
True  && x              = x
False && _              = False

-- Sequential disjunction
(||)                    :: Bool -> Bool -> Bool
True  || _              = True
False || x              = x

-- Negation
not                     :: Bool -> Bool
not True                = False
not False               = True

-- Conditional
if_then_else            :: Bool -> a -> a -> a
if_then_else b t f      = case b of True  -> t
                                    False -> f

--- Useful name for the last condition in a sequence of conditional equations.
otherwise               :: Bool
otherwise               = True

--- Enforce a Boolean condition to be true.
--- The computation fails if the argument evaluates to False.
solve :: Bool -> Bool
solve True = True

-- Equality on finite ground data terms
(==)                :: a -> a -> Bool

-- Disequality
(/=)                :: a -> a -> Bool
x /= y              = not (x==y)

-- Equational constraint
(=:=) :: a -> a -> Bool

-- Concurrent conjunction
(&)  :: Bool -> Bool -> Bool
True & True = True
```

```
-- Constrained expression. (c &> e) is evaluated by first solving c
-- and then evaluating e.
(&>) :: Bool -> a -> a
True &> x = x


-- Ordering
data Ordering = LT | EQ | GT

-- Comparison of arbitrary ground data terms.
-- Data constructors are compared in the order of their definition
-- in the datatype declarations and recursively in the arguments.
compare :: a -> a -> Ordering

-- Less-than on ground data terms
(<)   :: a -> a -> Bool
x < y = case compare x y of LT -> True
                            _  -> False

-- Greater-than on ground data terms
(>)   :: a -> a -> Bool
x > y = case compare x y of GT -> True
                            _  -> False

-- Less-or-equal on ground data terms
(<=)  :: a -> a -> Bool
x <= y = not (x > y)

-- Greater-or-equal on ground data terms
(>=)  :: a -> a -> Bool
x >= y = not (x < y)

-- Maximum of ground data terms
max :: a -> a -> a
max x y = if x >= y then x else y

-- Minimum of ground data terms
min :: a -> a -> a
min x y = if x <= y then x else y


-- Pairs

data (a,b) = (a,b)

fst               :: (a,b) -> a
fst (x,_)         = x
```

```
snd               :: (a,b) -> b
snd (_,y)         = y


-- Unit type

data () = ()


-- Lists

data [a] = [] | a : [a]

type String = [Char]

-- First element of a list
head              :: [a] -> a
head (x:_)        = x

-- Remaining elements of a list
tail              :: [a] -> [a]
tail (_:xs)       = xs

-- Is a list empty?
null            :: [_] -> Bool
null []         = True
null (_:_)      = False

-- Concatenation
(++)              :: [a] -> [a] -> [a]
[]     ++ ys      = ys
(x:xs) ++ ys      = x : xs++ys

-- List length
length            :: [a] -> Int
length []         = 0
length (_:xs)     = 1 + length xs

-- List index (subscript) operator, head has index 0
(!!)              :: [a] -> Int -> a
(x:xs) !! n | n==0      = x
            | n>0       = xs !! (n-1)

-- Map a function on a list
map               :: (a->b) -> [a] -> [b]
map _ []          = []
map f (x:xs)      = f x : map f xs

-- Accumulate all list elements by applying a binary operator from
```

```
-- left to right, i.e.,
-- foldl f z [x1,x2,...,xn] = (...((z `f` x1) `f` x2) ...) `f` xn :
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

-- Accumulate a non-empty list from left to right:
foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  = foldl f x xs

-- Accumulate all list elements by applying a binary operator from
-- right to left, i.e.,
-- foldr f z [x1,x2,...,xn] = (x1 `f` (x2 `f` ... (xn `f` z)...)) :
foldr             :: (a->b->b) -> b -> [a] -> b
foldr _ z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)

-- Accumulate a non-empty list from right to left:
foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 _ [x]        = x
foldr1 f (x1:x2:xs) = f x1 (foldr1 f (x2:xs))

-- Filter elements in a list
filter            :: (a -> Bool) -> [a] -> [a]
filter _ []       = []
filter p (x:xs)   = if p x then x : filter p xs
                           else filter p xs

-- Join two lists into one list of pairs. If one input list is shorter than
-- the other, the additional elements of the longer list are discarded.
zip               :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip (_:_)  []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

-- Join three lists into one list of triples. If one input list is shorter than
-- the other, the additional elements of the longer lists are discarded.
zip3                      :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3 []     _      _      = []
zip3 (_:_)  []     _      = []
zip3 (_:_)  (_:_)  []     = []
zip3 (x:xs) (y:ys) (z:zs) = (x,y,z) : zip3 xs ys zs

-- Join two lists into one list by applying a combination function to
-- corresponding pairs of elements (i.e., zip = zipWith (,)):
zipWith                  :: (a->b->c) -> [a] -> [b] -> [c]
zipWith _ []     _       = []
zipWith _ (_:_)  []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

```
-- Join three lists into one list by applying a combination function to
-- corresponding triples of elements (i.e., zip3 = zipWith3 (,,)):
zipWith3                        :: (a->b->c->d) -> [a] -> [b] -> [c] -> [d]
zipWith3 _ []     _      _      = []
zipWith3 _ (_:_)  []     _      = []
zipWith3 _ (_:_)  (_:_)  []     = []
zipWith3 f (x:xs) (y:ys) (z:zs) = f x y z : zipWith3 f xs ys zs

-- Transform a list of pairs into a pair of lists
unzip               :: [(a,b)] -> ([a],[b])
unzip []            = ([],[])
unzip ((x,y):ps)    = (x:xs,y:ys) where (xs,ys) = unzip ps

-- Transform a list of triples into a triple of lists
unzip3              :: [(a,b,c)] -> ([a],[b],[c])
unzip3 []           = ([],[],[])
unzip3 ((x,y,z):ts) = (x:xs,y:ys,z:zs) where (xs,ys,zs) = unzip3 ts

-- Concatenate a list of lists into one list
concat            :: [[a]] -> [a]
concat l          = foldr (++) [] l

-- Map a function from elements to lists and merge the result into one list
concatMap         :: (a -> [b]) -> [a] -> [b]
concatMap f       = concat . map f

-- Infinite list of repeated applications of a function f to an element x:
-- iterate f x = [x, f x, f (f x),...]
iterate           :: (a -> a) -> a -> [a]
iterate f x       = x : iterate f (f x)

-- Infinite list where all elements have the same value x:
repeat            :: a -> [a]
repeat x          = x : repeat x

-- List of length n where all elements have the same value x:
replicate         :: Int -> a -> [a]
replicate n x     = take n (repeat x)

-- Return prefix of length n
take              :: Int -> [a] -> [a]
take n l          = if n<=0 then [] else takep n l
   where takep _ []     = []
         takep n (x:xs) = x : take (n-1) xs

-- Return suffix without first n elements
drop              :: Int -> [a] -> [a]
drop n l          = if n<=0 then l else dropp n l
```

```
   where dropp _ []     = []
         dropp n (_:xs) = drop (n-1) xs

-- (splitAt n xs) is equivalent to (take n xs, drop n xs)
splitAt           :: Int -> [a] -> ([a],[a])
splitAt n l       = if n<=0 then ([],l) else splitAtp n l
   where splitAtp _ []     = ([],[])
         splitAtp n (x:xs) = let (ys,zs) = splitAt (n-1) xs in (x:ys,zs)

-- Return longest prefix with elements satisfying a predicate
takeWhile          :: (a -> Bool) -> [a] -> [a]
takeWhile _ []     = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []

-- Return suffix without takeWhile prefix
dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile _ []     = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs

-- (span p xs) is equivalent to (takeWhile p xs, dropWhile p xs)
span               :: (a -> Bool) -> [a] -> ([a],[a])
span _ []          = ([],[])
span p (x:xs)
       | p x       = let (ys,zs) = span p xs in (x:ys, zs)
       | otherwise = ([],x:xs)

-- (break p xs) is equivalent to (takeWhile (not.p) xs, dropWhile (not.p) xs)
-- i.e., it breaks a list at the first occurrence of an element satisfying p
break              :: (a -> Bool) -> [a] -> ([a],[a])
break p            = span (not . p)

-- Break a string into list of lines where a line is terminated at a
-- newline character. The resulting lines do not contain newline characters.
lines :: String -> [String]
lines [] = []
lines (c:cs) = let (l,restcs) = splitline (c:cs) in l : lines restcs
 where splitline []     = ([],[])
       splitline (c:cs) = if c=='\n'
                          then ([],cs)
                          else let (ds,es) = splitline cs in (c:ds,es)

-- Concatenate a list of strings with terminating newlines
unlines    :: [String] -> String
unlines ls = concatMap (++"\n") ls

-- Break a string into a list of words where the words are delimited by
-- white spaces.
words      :: String -> [String]
words s    = let s1 = dropWhile isSpace s
```

```
              in if s1=="" then []
                           else let (w,s2) = break isSpace s1
                                  in w : words s2
 where
    isSpace c = c == ' '  || c == '\t' || c == '\n' || c == '\r'

-- Concatenate a list of strings with a blank between two strings.
unwords    :: [String] -> String
unwords ws = if ws==[] then []
                       else foldr1 (\w s -> w ++ ' ':s) ws

-- Reverse the order of all elements in a list
reverse    :: [a] -> [a]
reverse    = foldl (flip (:)) []

-- Compute the conjunction of a Boolean list
and        :: [Bool] -> Bool
and        = foldr (&&) True

-- Compute the disjunction of a Boolean list
or         :: [Bool] -> Bool
or         = foldr (||) False

-- Is there an element in a list satisfying a given predicate?
any        :: (a -> Bool) -> [a] -> Bool
any p      = or . map p

-- Is a given predicate satisfied by all elements in a list?
all        :: (a -> Bool) -> [a] -> Bool
all p      = and . map p

-- Element of a list?
elem       :: a -> [a] -> Bool
elem x     = any (x==)

-- Not element of a list?
notElem    :: a -> [a] -> Bool
notElem x  = all (x/=)

--- Looks up a key in an association list.
lookup             :: a -> [(a,b)] -> Maybe b
lookup _ []        = Nothing
lookup k ((x,y):xys)
      | k==x       = Just y
      | otherwise = lookup k xys


-- Generating arithmetic sequences:
enumFrom               :: Int -> [Int]                   -- [n..]
```

```
enumFrom n              = n : enumFrom (n+1)

enumFromThen            :: Int -> Int -> [Int]            -- [n1,n2..]
enumFromThen n1 n2      = iterate ((n2-n1)+) n1

enumFromTo              :: Int -> Int -> [Int]            -- [n..m]
enumFromTo n m          = if n>m then [] else n : enumFromTo (n+1) m

enumFromThenTo          :: Int -> Int -> Int -> [Int]     -- [n1,n2..m]
enumFromThenTo n1 n2 m = takeWhile p (enumFromThen n1 n2)
                          where p x | n2 >= n1  = (x <= m)
                                    | otherwise = (x >= m)


-- Conversion functions between characters and their ASCII values

ord :: Char -> Int
chr :: Int -> Char


-- Convert a term into a printable representation

show :: a -> String


-- Types of primitive arithmetic functions

(+)     :: Int -> Int -> Int
(-)     :: Int -> Int -> Int
(*)     :: Int -> Int -> Int
div     :: Int -> Int -> Int
mod     :: Int -> Int -> Int

-- Unary minus (usually written as "- e")
negate :: Int -> Int
negate x = 0 - x


-- Maybe type

data Maybe a = Nothing | Just a

maybe               :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x


-- Either type
```

```
data Either a b = Left a | Right b

either               :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)  = f x
either _ g (Right x) = g x


-- Monadic IO

data IO a  -- conceptually: \mbox{\it{}World} -> (a,\mbox{\it{}World})

(>>=)             :: IO a -> (a -> IO b) -> IO b
return            :: a -> IO a

(>>)              :: IO a -> IO b -> IO b
a >> b            = a >>= (\_ -> b)

done              :: IO ()
done              = return ()

putChar           :: Char -> IO ()
getChar           :: IO Char

readFile          :: String -> IO String
writeFile         :: String -> String -> IO ()
appendFile        :: String -> String -> IO ()

putStr            :: String -> IO ()
putStr []         = done
putStr (c:cs)     = putChar c >> putStr cs

putStrLn          :: String -> IO ()
putStrLn cs       = putStr cs >> putChar '\n'

getLine           :: IO String
getLine           = do c <- getChar
                       if c=='\n' then return []
                                  else do cs <- getLine
                                          return (c:cs)

-- Convert a term into a string and print it
print             :: _ -> IO ()
print t           = putStrLn (show t)

-- Solve a constraint as an I/O action:
-- Note: the constraint should be always solvable in a deterministic way
doSolve :: Bool -> IO ()
doSolve b | b = done
```

```
-- IO monad auxiliary functions:

-- Execute a sequence of I/O actions and collect all results in a list:
sequenceIO       :: [IO a] -> IO [a]
sequenceIO []     = return []
sequenceIO (c:cs) = do x  <- c
                       xs <- sequenceIO cs
                       return (x:xs)

-- Execute a sequence of I/O actions and ignore the results:
sequenceIO_        :: [IO _] -> IO ()
sequenceIO_         = foldr (>>) done

-- Map an I/O action function on a list of elements.
-- The results of all I/O actions are collected in a list.
mapIO             :: (a -> IO b) -> [a] -> IO [b]
mapIO f            = sequenceIO . map f

-- Map an I/O action function on a list of elements.
-- The results of all I/O actions are ignored.
mapIO_            :: (a -> IO _) -> [a] -> IO ()
mapIO_ f           = sequenceIO_ . map f


-- Non-determinism and free variables

-- Non-deterministic choice par excellence.
-- The value of (x ? y) is either x or y.
(?)   :: a -> a -> a
x ? _ = x
_ ? y = y

-- Evaluates to a fresh free variable.
unknown :: _
unknown = let x free in x


-- Type of constraints (included for backward compatibility)
type Success = Bool

success :: Success
success = True


-- Encapsulated search

-- primitive operator to control non-determinism
try :: (a->Bool) -> [a->Bool]
```

```
-- compute all solutions via depth-first search
solveAll      :: (a->Bool) -> [a->Bool]
solveAll g    = collect (try g)
              where collect []         = []
                    collect [g]        = [g]
                    collect (g1:g2:gs) = concat (map solveAll (g1:g2:gs))

-- compute first solution via depth-first search
once          :: (a->Bool) -> (a->Bool)
once g        = head (solveAll g)

-- compute all values of solutions via depth-first search
findall       :: (a->Bool) -> [a]
findall g     = map unpack (solveAll g)

-- compute best solution via branch-and-bound with depth-first search
best          :: (a->Bool) -> (a->a->Bool) -> [a->Bool]


-- show the solution of a solved constraint
browse        :: (a->Bool) -> IO ()
browse g      = putStr (show (unpack g))

-- show the solutions of a list of solved constraints
browseList        :: [a->Bool] -> IO ()
browseList []     = done
browseList (g:gs) = browse g >> putChar '\n' >> browseList gs


-- unpack a solution's value from a (solved) search goal
unpack          :: (a -> Bool) -> a
unpack g | g x  = x  where x free
```

C Syntax of Curry

The syntax description is not intended to be used by automatic tools (since it contains some ambiguities for the sake of simplicity).

The syntax is close to Haskell but the following differences should be noted.

- Currently, there are no infix constructors except for "`:`", the predefined list *cons*tructor. They will be added later, although they are already used in concrete example programs.

C.1 Notational Conventions

The syntax is given in extended Backus-Naur-Form (eBNF), using the following notation:

NonTerm	::= α	production
	NonTerm	nonterminal symbol
	`Term`	terminal symbol
	$[\alpha]$	optional
	$\alpha \mid \beta$	alternative
	$\alpha_{\langle\beta\rangle}$	difference – elements generated by α without those generated by β

C.2 Lexicon

The case of identifiers matters, i.e., "abc" differs from "Abc". There are four *case modes* selectable at compilation time:

- Prolog mode: variables start with an upper case latter, all other identifier symbols start with a lower case letter.
- Gödel mode: like Prolog mode with the cases swapped.
- Haskell mode: see section 1.3 of the Haskell report.
- free: no constraints on the case of identifiers.

The default case mode is *free*. If a case mode is selected but not obeyed, the compiler issues a warning.

The syntax does not define the following non-terminal symbols defining classes of identifiers: *ModuleID*, *TypeConstrID*, *DataConstrID*, *TypeVarID*, *InfixOpID*, *FunctionID*, *VariableID*, *LabelID*. All, except *InfixOpID*, consist of an initial letter, whose upper or lower case depend on the case mode, followed by any number of letters, digits, underscores, and single quotes. Additionally, *ModuleID* can contain dots at inner positions. *InfixOpID* is any string of characters from the string "`~!@#$%^&*+-=<>?./|\:`" or another identifier (like *VariableID*) enclosed in ‘`...`‘ like ‘`mod`‘.

The following symbols are *keywords* and cannot be used as an identifier:

```
case      data      do        else      external  fcase     free
if        infix     infixl    infixr    import    in        let
module    of        then      type      where
```

Note that the symbols `as`, `hiding`, and `qualified` are not keywords. They have only a special meaning in module headers and can be used as ordinary identifiers.

The syntax leaves undefined *Literal* of primitive types. These are literal constants, such as "`1`", "`3.14`", or the character "`'a'`". They are as in Java which adopts the Unicode standard to represent characters and character strings.

Comments begins either with "`--`" and terminate at the end of the line or with "`{-`" and terminate with a matching "`-}`", i.e., the delimiters "`{-`" and "`-}`" act as parentheses and can be nested.

C.3 Layout

Similarly to Haskell, a Curry programmer can use layout information to define the structure of blocks. For this purpose, we define the indentation of a symbol as the column number indicating the start of this symbol. The indentation of a line is the indentation of its first symbol.[13]

The layout (or "off-side") rule applies to lists of syntactic entities after the keywords `let`, `where`, `do`, or `of`. In the subsequent context-free syntax (Section C.4), these lists are enclosed with curly brackets ({ }) and the single entities are separated by semicolons (`;`). Instead of using the curly brackets and semicolons of the context-free syntax, a Curry programmer must specify these lists by indentation: the indentation of a list of syntactic entities after `let`, `where`, `do`, or `of` is the indentation of the next symbol following the `let`, `where`, `do`, `of`. Any item of this list start with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in its previous line. Lines with an indentation less than the indentation of the list terminates the entire list. Moreover, a list started by `let` is terminated by the keyword `in`. Thus, the sentence

```
f x = h x where { g y = y+1 ; h z = (g z) * 2 }
```

which is valid w.r.t. the context-free syntax, is written with the layout rules as

```
f x = h x
 where g y = y+1
       h z = (g z) * 2
```

or also as

```
f x = h x  where
  g y = y+1
  h z = (g z)
          * 2
```

To avoid an indentation of top-level declarations, the keyword `module` and the end-of-file token are assumed to start in column 0.

C.4 Context Free Syntax

Module ::= `module` *ModuleID* [*Exports*] `where` *Block*
| *Block*

[13] In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.

$$
\begin{array}{rcll}
\mathit{ModuleID} & ::= & \text{see lexicon} & \\
\\
\mathit{Exports} & ::= & \texttt{(}\ \mathit{Export}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Export}_n\ \texttt{)} & \\
\\
\mathit{Export} & ::= & \mathit{QFunctionName} & \\
& | & \mathit{QTypeConstrID} & \\
& | & \mathit{QTypeConstrID}\ \texttt{(..)} & \\
& | & \texttt{module}\ \mathit{ModuleID} & \\
\\
\mathit{Block} & ::= & \{\ [\mathit{ImportDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{ImportDecl}_k\ \texttt{;}] & \\
& & \quad [\mathit{FixityDeclaration}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{FixityDeclaration}_m\ \texttt{;}] & \\
& & \quad \mathit{BlockDeclaration}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{BlockDeclaration}_n\ \} & (k, m, n \geqslant 0) \\
\\
\mathit{ImportDecl} & ::= & \texttt{import}\ [\texttt{qualified}]\ \mathit{ModuleID}\ [\texttt{as}\ \mathit{ModuleID}]\ [\mathit{ImportRestr}] & \\
\\
\mathit{ImportRestr} & ::= & \texttt{(}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} & \\
& | & \texttt{hiding (}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} & \\
\\
\mathit{Import} & ::= & \mathit{FunctionName} & \\
& | & \mathit{TypeConstrID} & \\
& | & \mathit{TypeConstrID}\ \texttt{(..)} & \\
\\
\mathit{BlockDeclaration} & ::= & \mathit{TypeSynonymDecl} & \\
& | & \mathit{DataDeclaration} & \\
& | & \mathit{FunctionDeclaration} & \\
\\
\mathit{TypeSynonymDecl} & ::= & \texttt{type}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{TypeExpr} & \\
\\
\mathit{SimpleType} & ::= & \mathit{TypeConstrID}\ \mathit{TypeVarID}_1\ \ \ldots\ \ \mathit{TypeVarID}_n & (n \geqslant 0) \\
\\
\mathit{TypeConstrID} & ::= & \text{see lexicon} & \\
\\
\mathit{DataDeclaration} & ::= & \texttt{data}\ \mathit{SimpleType} & (\textit{external data type}) \\
& | & \texttt{data}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{ConstrDecl}_n & (n > 0) \\
\\
\mathit{ConstrDecl} & ::= & \mathit{DataConstrID}\ \mathit{SimpleTypeExpr}_1\ \ \ldots\ \ \mathit{SimpleTypeExpr}_n & (n \geq 0) \\
& | & \mathit{DataConstrID}\ \{\ \mathit{FieldDeclaration}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FieldDeclaration}_n\ \} & (n \geq 0) \\
\\
\mathit{DataConstrID} & ::= & \text{see lexicon} & \\
\\
\mathit{FieldDeclaration} & ::= & \mathit{LabelID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{LabelID}_n\ \texttt{::}\ \mathit{TypeExpr} & (n > 0) \\
\\
\mathit{LabelID} & ::= & \text{see lexicon} &
\end{array}
$$

TypeExpr ::= *TypeConsExpr* [`->` *TypeExpr*]

TypeConsExpr ::= *QTypeConstrID SimpleTypeExpr*$_1$ ... *SimpleTypeExpr*$_n$ ($n > 0$)
| *SimpleTypeExpr*

SimpleTypeExpr ::= *TypeVarID* | `_`
| *QTypeConstrID*
| `()` (*unit type*)
| `(` *TypeExpr*$_1$ `,` ... `,` *TypeExpr*$_n$ `)` (*tuple type*, $n > 1$)
| `[` *TypeExpr* `]` (*list type*)
| `(` *TypeExpr* `)` (*parenthesized type*)

TypeVarID ::= see lexicon

FixityDeclaration ::= *FixityKeyword Natural InfixOp*$_1$ `,` ... `,` *InfixOp*$_n$ ($n > 0$)

FixityKeyword ::= `infixl` | `infixr` | `infix`

Natural ::= *Digit* | *Digit Natural*

Digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

FunctionDeclaration ::= *Signature* | *Equat*

Signature ::= *FunctionNames* `::` *TypeExpr*

FunctionNames ::= *FunctionName*$_1$ `,` ... `,` *FunctionName*$_n$ ($n > 0$)

FunctionName ::= *FunctionID* | `(`*InfixOpID*`)` (*function*)

QFunctionName ::= *QFunctionID* | `(`*QInfixOpID*`)` (*qualified function*)

FunctionID ::= see lexicon

Equat ::= *FunLHS* `=` *Expr* [`where` *LocalDecls*]
| *FunLHS CondExprs* [`where` *LocalDecls*]

FunLHS ::= *FunctionName SimplePat*$_1$ ... *SimplePat*$_n$ ($n \geqslant 0$)
| *SimplePat InfixOp SimplePat*

Pattern ::= *QDataConstrID SimplePat*$_1$ ... *SimplePat*$_n$ [`:` *Pattern*] ($n > 0$)
| *SimplePat* [`:` *Pattern*]

$$
\begin{array}{rcll}
\mathit{SimplePat} & ::= & \mathit{VariableID} \mid \texttt{\_} & \\
& \mid & \mathit{QDataConstrID} & \\
& \mid & \mathit{Literal} & \\
& \mid & \texttt{()} & \\
& \mid & \texttt{(}\ \mathit{Pattern}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Pattern}_n\ \texttt{)} & (n > 1) \\
& \mid & \texttt{(}\ \mathit{Pattern}\ \texttt{)} & \\
& \mid & \texttt{[}\ \mathit{Pattern}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Pattern}_n\ \texttt{]} & (n \geqslant 0) \\
& \mid & \mathit{VariableID}\ \texttt{@}\ \mathit{SimplePat} & (\textit{as-pattern}) \\
& \mid & \mathit{QDataConstrID}\ \{\ \mathit{FieldPat}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FieldPat}_n\ \}\ (\textit{labeled pattern},\ n \geq 0) & \\
\\
\mathit{FieldPat} & ::= & \mathit{QLabelID}\ \texttt{=}\ \mathit{Pattern} & \\
\\
\mathit{VariableID} & ::= & \text{see lexicon} & \\
\\
\mathit{LocalDecls} & ::= & \{\mathit{LocalDeclaration}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{LocalDeclaration}_n\ \} & (n > 0) \\
\\
\mathit{LocalDeclaration} & ::= & \mathit{FunctionDeclaration} & \\
& \mid & \mathit{PatternDeclaration} & \\
& \mid & \mathit{VariableID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{VariableID}_n\ \texttt{free} & (n > 0) \\
\\
\mathit{PatternDeclaration} & ::= & \mathit{Pattern}\ \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\\
\mathit{CondExprs} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExprs}] & \\
\\
\mathit{Expr} & ::= & \mathit{InfixExpr}\ \texttt{::}\ \mathit{TypeExpr} & (\textit{expression type signature}) \\
& \mid & \mathit{InfixExpr} & \\
\\
\mathit{InfixExpr} & ::= & \mathit{NoOpExpr}\ \mathit{QInfixOp}\ \mathit{InfixExpr} & (\textit{infix operator application}) \\
& \mid & \texttt{-}\ \mathit{InfixExpr} & (\textit{unary minus}) \\
& \mid & \mathit{NoOpExpr} & \\
\\
\mathit{NoOpExpr} & ::= & \texttt{\textbackslash}\ \mathit{SimplePat}_1\ \ \ldots\ \ \mathit{SimplePat}_n\ \texttt{->}\ \mathit{Expr} & (n > 0) \\
& \mid & \texttt{let}\ \mathit{LocalDecls}\ \texttt{in}\ \mathit{Expr} & (\textit{let expression}) \\
& \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} & (\textit{conditional}) \\
& \mid & \texttt{case}\ \mathit{Expr}\ \texttt{of}\ \{\mathit{Alt}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{Alt}_n\ \} & (\textit{case expression},\ n \geqslant 1) \\
& \mid & \texttt{fcase}\ \mathit{Expr}\ \texttt{of}\ \{\mathit{Alt}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{Alt}_n\ \} & (\textit{fcase expression},\ n \geqslant 1) \\
& \mid & \texttt{do}\ \{\mathit{Stmt}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{Stmt}_n\ \texttt{;}\ \mathit{Expr}\ \} & (\textit{do expression},\ n \geqslant 0) \\
& \mid & \mathit{FunctExpr} & \\
\\
\mathit{FunctExpr} & ::= & [\mathit{FunctExpr}]\ \mathit{BasicExpr} & (\textit{function application}) \\
\\
\mathit{BasicExpr} & ::= & \mathit{VariableID} & (\textit{variable}) \\
& \mid & \texttt{\_} & (\textit{anonymous free variable}) \\
& \mid & \mathit{QDataConstrID} & (\textit{data constructor})
\end{array}
$$

	QFunctionName	(*defined function*)
|	*Literal*	
|	`()`	(*empty tuple*)
|	`(` *Expr* `)`	(*parenthesized expression*)
|	`(` $Expr_1$ `,` ... `,` $Expr_n$ `)`	(*tuple*, $n > 1$)
|	`[` $Expr_1$ `,` ... `,` $Expr_n$ `]`	(*finite list*, $n \geqslant 0$)
|	`[` *Expr* [`,` *Expr*] `..` [*Expr*] `]`	(*arithmetic sequence*)
|	`[` *Expr* `|` $Qual_1$ `,` ... `,` $Qual_n$ `]`	(*list comprehension*, $n \geqslant 1$)
|	`(` *Expr QInfixOp* `)`	(*left section*)
|	`(` $QInfixOp_{\langle - \rangle}$ *Expr* `)`	(*right section*)
|	*QDataConstrID* `{` $FBind_1$ `,` ... `,` $FBind_n$ `}`	(*labeled construction*, $n \geqslant 0$)
|	$BasicExpr_{\langle QDataConstrID \rangle}$ `{` $FBind_1$ `,` ... `,` $FBind_n$ `}`	(*labeled update*, $n > 0$)

FBind ::= *QLabelID* `=` *Expr*

Alt ::= *Pattern* `->` *Expr* [`where` *LocalDecls*]
| *Pattern GdAlts* [`where` *LocalDecls*]

GdAlts ::= `|` *InfixExpr* `->` *Expr* [*GdAlts*]

Qual ::= *Expr*
| `let` *LocalDecls*
| *Pattern* `<-` *Expr*

Stmt ::= *Expr*
| `let` *LocalDecls*
| *Pattern* `<-` *Expr*

QInfixOp ::= *QInfixOpID*
| ‘ *QFunctionID* ‘

InfixOp ::= *InfixOpID*
| ‘ *FunctionID* ‘

QTypeConstrID ::= [*ModuleID* `.`] *TypeConstrID*
QDataConstrID ::= [*ModuleID* `.`] *DataConstrID*
QInfixOpID ::= [*ModuleID* `.`] *InfixOpID*
QFunctionID ::= [*ModuleID* `.`] *FunctionID*
QLabelID ::= [*ModuleID* `.`] *LabelID*

Literal ::= *Int* | *Char* | *String* | *Float*

InfixOpID ::= see lexicon

Int	::=	see lexicon
Char	::=	see lexicon
String	::=	see lexicon
Float	::=	see lexicon

If the alternative *FunctionDeclaration* is used in a *LocalDeclaration*, then the left-hand side (*FunLHS*) must have at least one pattern after the *FunctionName* (instead of zero patterns which is possible in top-level function declarations).

In *CondExprs*, the first expression must be of type `Bool`.

In qualified names (e.g., *QFunctionID*), no characters (e.g., spaces) are allowed between the dot and the module and entity names. On the other hand, an infix expression (*Expr QInfixOpID Expr*) must contain at least one space or similar character after the left expression if the infix operator starts with a dot.

C.5 Infix Operators

In the grammar above, the use of infix operators in the rule for *Expr* is ambiguous. These ambiguities are resolved by assigning an associativity and precedence to each operator (*InfixOpID*) by a *fixity declaration*. There are three kinds of associativities, non-, left- and right-associativity (`infix`, `infixl`, `infixr`) and ten levels of precedence, 0 to 9, where level 0 binds least tightly and level 9 binds most tightly. All fixity declarations must appear at the beginning of a module. Any operator without an explicit fixity declaration is assumed to have the declaration `infixl 9`. For instance, the expression "`1+2*3+4==x && b`" is equivalent to "`((((1+(2*3))+4)==x) && b)`" w.r.t. the fixity declarations provided in the prelude.

Note that the correct use of precedences and associativities excludes some expressions which are valid w.r.t. the context-free grammar. In general, the arguments of an infix operator must have an infix operator at the top with a higher precedence than the current infix operator (or no infix operator at the top). Additionaly, the left or right argument of a left- or right-associative operator can have a top infix operator with the same precedence. The unary minus operator is interpreted with precedence 6 which means that its argument must have a precedence of at least 7. The expression "`(- t)`" is not considered as a right section but as the negation of t. As a consequence, the expressions "`1<2==True`" and "`1 + - 3`" are not allowed and must be bracketed as "`(1<2)==True`" and "`1 + (- 3)`".

D Operational Semantics of Curry

The precise specification of the operational semantics of Curry is based on the patterns on the rules' left-hand sides for each function. Therefore, we describe the computation model by providing a precise definition of pattern matching (Section D.1) which is the basis to define the computation steps on expressions (Section D.2). The extension of this basic computation model to solving equational constraints and higher-order functions is described in Sections D.3 and D.4, respectively. Section D.5 describes the automatic generation of the definitional trees to guide the pattern matching strategy. Finally, Section D.6 specifies the operational semantics of the primitive operator `try` for encapsulating search.

D.1 Definitional Trees

We start by considering only the unconditional first-order part of Curry, i.e., rules do not contain conditions and λ-abstractions and partial function applications are not allowed. We assume some familiarity with basic notions of term rewriting [15] and functional logic programming [22].

We denote by $\mathcal{C}$ the set of *constructors* (with elements c, d), by $\mathcal{F}$ the set of *defined functions* or *operations* (with elements f, g), and by $\mathcal{X}$ the set of *variables* (with elements x, y), where $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{X}$ are disjoint. An *expression* (*data term*) is a variable $x \in \mathcal{X}$ or an application $\varphi(e_1, \ldots, e_n)$ where $\varphi \in \mathcal{C} \cup \mathcal{F}$ ($\varphi \in \mathcal{C}$) has arity n and $e_1, \ldots, e_n$ are expressions (data terms).[15] The set of all expressions and data terms are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. A *call pattern* is an expression of the form $f(t_1, \ldots, t_n)$ where each variable occurs only once, $f \in \mathcal{F}$ is an n-ary function, and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. $root(e)$ denotes the symbol at the *root* of the expression e. A *position* p in an expression e is represented by a sequence of natural numbers, $e|_p$ denotes the *subterm* or *subexpression* of e at position p, and $e[e']_p$ denotes the result of *replacing the subterm* $e|_p$ by the expression e' (see [15] for details).

A *substitution* is a mapping $\sigma\colon \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\sigma(x) \neq x$ only for finitely many variables x. Thus, we denote a substitution by $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $\sigma(x_i) \neq x_i$ for $i = 1, \ldots, n$, and id denotes the identity substitution ($n = 0$). $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ is the *domain* of σ and

$$\mathcal{VR}an(\sigma) = \{x \mid x \text{ is a variable occurring in some } t_i, i \in \{1, \ldots, n\}\}$$

is its *variable range*. Substitutions are extended to morphisms on expressions by $\sigma(f(e_1, \ldots, e_n)) = f(\sigma(e_1), \ldots, \sigma(e_n))$ for every expression $f(e_1, \ldots, e_n)$.

A Curry program is a set of rules $l = r$ satisfying some restrictions (see Section 2.3). In particular, the left-hand side l must be a call pattern. A rewrite rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables.

An *answer expression* is a pair $\sigma \,[]\, e$ consisting of a substitution σ (the answer computed so far) and an expression e. An answer expression $\sigma \,[]\, e$ is *solved* if e is a data term. We sometimes omit the identity substitution in answer expressions, i.e., we write e instead of $id \,[]\, e$ if it is clear from the context. A *disjunctive expression* is a (multi-)set of answer expressions $\{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$. The set of all disjunctive expressions is denoted by $\mathcal{D}$.

[15] Since we do not consider partial applications in this part, we write the full application as $\varphi(e_1, \ldots, e_n)$ which is equivalent to Curry's notation $\varphi\ e_1 \ldots e_n$.

A single *computation step* performs a reduction in exactly one unsolved expression of a disjunction. To provide a precise definition of the operational semantics, we use definitional trees.[16] A definitional tree is a hierarchical structure containing all rules of a defined function. $\mathcal{T}$ is a *definitional tree with call pattern* π iff the depth of $\mathcal{T}$ is finite and one of the following cases holds:

$\mathcal{T} = rule(l = r)$, where $l = r$ is a variant of a rule such that $l = \pi$.

$\mathcal{T} = branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where o is an occurrence of a variable in π, $r \in \{rigid, flex\}$, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi|_o$, for some $k > 0$, and, for all $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with call pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new variables.

$\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are definitional trees with call pattern π.[17]

The $rigid/flex$ tag in a branch will be used to specify the suspension behavior w.r.t. free variables as actual arguments. User-defined functions are always translated into definitional trees with $flex$ tags, but case expressions are translated into definitional trees with *rigid* tags.

A *definitional tree of an n-ary function* f is a definitional tree $\mathcal{T}$ with call pattern $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables, such that for each rule $l = r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l' = r')$ in $\mathcal{T}$ with l variant of l'. In the following, we write $pat(\mathcal{T})$ for the call pattern of a definitional tree $\mathcal{T}$.

It is always possible to construct a definitional tree for each function (concrete algorithms are described in [4, 34] and in Section D.5). For instance, consider the following definition of the less-or-equal predicate on natural numbers represented by data terms built from `z` (zero) and `s` (successor):

```
data Nat = z | s Nat

leq :: Nat -> Nat -> Bool
leq z     n     = True
leq (s m) z     = False
leq (s m) (s n) = leq m n
```

Consider a function call like `(leq` e_1 e_2`)`. In order to apply some reduction rule, the first argument e_1 must always be evaluated to *head normal form* (i.e., to an expression without a defined function symbol at the top). However, the second argument must be evaluated only if the first argument has the form `(s` e`)`.[18] This dependency between the first and the second argument is expressed by the definitional tree

$$branch(\texttt{leq}(x_1, x_2), 1, flex, rule(\texttt{leq}(\texttt{z}, x_2) = \texttt{True}),$$
$$branch(\texttt{leq}(\texttt{s}(x), x_2), 2, flex, rule(\texttt{leq}(\texttt{s}(x), \texttt{z}) = \texttt{False}),$$
$$rule(\texttt{leq}(\texttt{s}(x), \texttt{s}(y)) = \texttt{leq}(x, y))))$$

[16] Our notion is influenced by Antoy's work [4], but here we use an extended form of definitional trees.

[17] For the sake of simplicity, we consider only binary *or* nodes. The extension to such nodes with more than two subtrees is straightforward.

[18] Naive lazy narrowing strategies may also evaluate the second argument in any case. However, as shown in [7], the consideration of dependencies between arguments is essential to obtain optimal evaluation strategies.

Computation step for a single expression:

$$\frac{\mathcal{E}val[\![e_i]\!] \Rightarrow D}{\mathcal{E}val[\![e_1 \,\&\, e_2]\!] \Rightarrow replace(e_1 \,\&\, e_2, i, D)} \quad i \in \{1,2\}$$

$$\frac{\mathcal{E}val[\![e_i]\!] \Rightarrow D}{\mathcal{E}val[\![c(e_1,\ldots,e_n)]\!] \Rightarrow replace(c(e_1,\ldots,e_n), i, D)} \quad i \in \{1,\ldots,n\}$$

$$\frac{\mathcal{E}val[\![f(e_1,\ldots,e_n);\mathcal{T}]\!] \Rightarrow D}{\mathcal{E}val[\![f(e_1,\ldots,e_n)]\!] \Rightarrow D} \quad \text{if } \mathcal{T} \text{ is a definitional tree for } f \text{ with fresh variables}$$

Computation step for an operation-rooted expression e:

$$\frac{}{\mathcal{E}val[\![e; rule(l = r)]\!] \Rightarrow \{id \,[\!]\, \sigma(r)\}} \quad \text{if } \sigma \text{ is a substitution with } \sigma(l) = e$$

$$\frac{\mathcal{E}val[\![e;\mathcal{T}_1]\!] \Rightarrow D_1 \quad \mathcal{E}val[\![e;\mathcal{T}_2]\!] \Rightarrow D_2}{\mathcal{E}val[\![e; or(\mathcal{T}_1,\mathcal{T}_2)]\!] \Rightarrow D_1 \cup D_2}$$

$$\mathcal{E}val[\![e; branch(\pi,p,r,\mathcal{T}_1,\ldots,\mathcal{T}_k)]\!] \Rightarrow$$
$$\begin{cases} D & \text{if } e|_p = c(e_1,\ldots,e_n),\ pat(\mathcal{T}_i)|_p = c(x_1,\ldots,x_n), \text{ and } \mathcal{E}val[\![e;\mathcal{T}_i]\!] \Rightarrow D \\ \varnothing & \text{if } e|_p = c(\cdots) \text{ and } pat(\mathcal{T}_i)|_p \neq c(\cdots), i = 1,\ldots,k \\ \bigcup_{i=1}^{k} \{\sigma_i \,[\!]\, \sigma_i(e)\} & \text{if } e|_p = x,\ r = flex, \text{ and } \sigma_i = \{x \mapsto pat(\mathcal{T}_i)|_p\} \\ replace(e,p,D) & \text{if } e|_p = f(e_1,\ldots,e_n) \text{ and } \mathcal{E}val[\![e|_p]\!] \Rightarrow D \end{cases}$$

Derivation step for a disjunctive expression:

$$\frac{\mathcal{E}val[\![e]\!] \Rightarrow \{\sigma_1 \,[\!]\, e_1,\ldots,\sigma_n \,[\!]\, e_n\}}{\{\sigma \,[\!]\, e\} \cup D \;\Rightarrow\; \{\sigma_1 \circ \sigma \,[\!]\, e_1,\ldots,\sigma_n \circ \sigma \,[\!]\, e_n\} \cup D}$$

Figure 2: Operational semantics of Curry

This definitional tree specifies that the first argument is initially evaluated and the second argument is only evaluated if the first argument has the constructor `s` at the top. The precise operational meaning induced by definitional trees is described in the following section.

D.2 Computation Steps

The operational semantics is defined by the derivability relation $D_1 \Rightarrow D_2$ on disjunctive expressions specified in Figure 2. An inference rule $\frac{\alpha}{\beta}$ should be read as "if α is derivable, then β is also derivable". We say that the computation of an expression e *suspends* if there is no D with $\mathcal{E}val[\![e]\!] \Rightarrow D$. A constraint is solvable if it can be reduced to `True`. The exact method to solve constraints depends on the constraint solver. A method to solve equational constraints on data terms is specified in Section D.3. For the purpose of this definition, we consider `True` as the neutral element of the operation `&`, i.e., `True &` c and c `& True` are considered as equivalent to c.

As can be seen by the rules for evaluating constraints in Figure 2, the concurrent conjunction of two constraints is evaluated by applying computation steps to one of the two constraints. Since both constraints must be finally solved (i.e., reduced to `True`), one can consider the evaluation

of a concurrent conjunction as two computation threads evaluating these constraints. They are synchronized by accessing common variables (which may suspend a thread, see below). In a simple sequential implementation, the evaluation of $e_1 \,\&\, e_2$ could be started by an attempt to solve e_1. If the evaluation of e_1 suspends, an evaluation step is applied to e_2. If a variable responsible to the suspension of e_1 is instantiated, the left expression e_1 will be evaluated in the subsequent step. Thus, we obtain a concurrent behavior with an interleaving semantics. However, a sophisticated implementation should provide a fair selection of threads, e.g., as done in a Java-based implementation of Curry [27].

A computation step for an expression e attempts to apply a reduction step to an outermost operation-rooted subterm in e by evaluating this subterm with the definitional tree for the function symbol at the root of this subterm. Although it is unspecified which outermost subterm is evaluated (compare second inference rule in Figure 2 for constructor-rooted expressions), we assume that a single (e.g., leftmost) outermost subterm is always selected. The evaluation of an operation-rooted term is defined by a case distinction on the definitional tree. If it is a *rule* node, this rule is applied. An *or* node produces a disjunction where the different alternatives are combined. Note that this definition requires that the entire disjunction suspends if one disjunct suspends. This can be relaxed in concrete implementations by continuing the evaluation in one *or* branch even if the other branch is suspended. However, to ensure completeness, it is not allowed to omit a suspended *or* branch and continue with the other non-suspended *or* branch [24]. For a similar reason, we cannot commit to a disjunct which does not bind variables but we have to consider both alternatives (see [6] for a counter-example).

The most interesting case is a *branch* node. Here we have to branch on the value of the top-level symbol at the selected position. If the symbol is a constructor, we proceed with the appropriate definitional subtree, if possible. If it is a function symbol, we proceed by evaluating this subexpression. If it is a variable and the branch is flexible, we instantiate the variable to the different constructors, otherwise (if the branch is *rigid*) we cannot proceed and suspend. The auxiliary function *replace* puts a possibly disjunctive expression into a subterm:

$$replace(e, p, \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}) = \{\sigma_1 \,[]\, \sigma_1(e)[e_1]_p, \ldots, \sigma_n \,[]\, \sigma_n(e)[e_n]_p\} \qquad (n \geq 0)$$

The overall computation strategy on disjunctive expressions takes a disjunct $\sigma \,[]\, e$ not in solved form and evaluates e. If the evaluation does not suspend and yields a disjunctive expression, we substitute it for $\sigma \,[]\, e$ composed with the old answer substitution.

Soundness and completeness results for the operational semantics can be found in [24]. Note that the evaluation of expressions is completely deterministic if the concurrent conjunction of constraints does not occur.

Conditional rules. Note that the operational semantics specified in Figure 2 handles only unconditional rules. Conditional rules are not explicitly treated but we assume that each conditional rule of the form

```
l | c = r
```

is transformed into

```
l = (c Prelude.&> r)
```

$$\frac{\mathcal{E}val[\![c(e_1,\ldots,e_n)]\!] \Rightarrow D}{\mathcal{E}val[\![\texttt{ensureNotFree}(c(e_1,\ldots,e_n))]\!] \Rightarrow D}$$

$$\frac{\mathcal{E}val[\![f(e_1,\ldots,e_n)]\!] \Rightarrow D}{\mathcal{E}val[\![\texttt{ensureNotFree}(f(e_1,\ldots,e_n))]\!] \Rightarrow \mathit{replace}(\texttt{ensureNotFree}(f(e_1,\ldots,e_n)),1,D)}$$

Figure 3: Evaluation of `ensureNotFree`

(see also Section D.7.3).

Ensuring instantiation The semantics of the primitive `ensureNotFree` to ensure computations with instantiated expressions, as informally described in Section 5.6, can be easily described by the rules in Figure 3. If `ensureNotFree` occurs in a computation with a constructor term as argument, it has no effect on the computation (first rule). It behaves like the identity as long as the argument is a function call. Otherwise (i.e., if the argument is a free variable), no evaluation step is possible.

Sharing and graph rewriting. For the sake of simplicity, this description of the operational semantics is based on term rewriting and does not take into account that common subterms are shared (see Section 2.3.1). We only note here that *several occurrences of the same variable are always shared*, i.e., if an argument of a function is instantiated during a call to this function to an expression and this expression is evaluated to some value (head normal form), then all other expressions resulting from instantiating occurrences of the same argument are replaced by the same value (head normal form).[19] This is necessary not only for efficiency reasons but also for the soundness of the operational semantics in the presence of non-deterministic functions, as discussed in Section 2.3.1. The sharing of variables can be described with the more complicated framework of graph rewriting. Formal descriptions of graph rewriting and narrowing can be found in [16, 19]. A formal description of Curry's operational semantics based on heap structures to model sharing can be found in [3].

D.3 Equational Constraints

An equational constraint e_1`=:=`e_2 is solvable if both sides are reducible to a unifiable data term (*strict equality*). An equational constraint can be solved in an incremental way by an interleaved lazy evaluation of the expressions and binding of variables to constructor terms. The evaluation steps implementing this method are shown in Figure 4, where we consider the symbol `=:=` is a binary infix function symbol. The last two rules implements an occur check where the *critical variables* (cv) are only those variables occurring outside a function call:

$$cv(x) \quad = \quad \{x\}$$

[19] It should be noted that values are constructor terms like `23`, `True`, or `[2,4,5]`. This means that the evaluation of constraints and I/O actions are not shared since they are not replaced by a value after evaluation but constraints are solved in order to apply a conditional rule (in case of constraints) and I/O actions are applied to the outside world when they appear at the top-level in a program. This is the intended behavior since the expressions "`putChar 'a' >> putChar 'a'`" and "`let ca = putChar 'a' in ca >> ca`" should have an identical behvarior, namely printing the character `'a'` twice.

$$\frac{\mathcal{E}val[\![e_i]\!] \Rightarrow D}{\mathcal{E}val[\![e_1 \texttt{=:=} e_2]\!] \Rightarrow replace(e_1 \texttt{=:=} e_2, i, D)} \text{ if } e_i = f(t_1, \ldots, t_n),\ i \in \{1, 2\}$$

$$\overline{\mathcal{E}val[\![c(e_1, \ldots, e_n) \texttt{=:=} c(e'_1, \ldots, e'_n)]\!] \Rightarrow \{id \,[\!]\, e_1 \texttt{=:=} e'_1 \,\&\, \ldots \,\&\, e_n \texttt{=:=} e'_n\}}$$

$$\overline{\mathcal{E}val[\![c(e_1, \ldots, e_n) \texttt{=:=} d(e'_1, \ldots, e'_m)]\!] \Rightarrow \varnothing} \text{ if } c \neq d \text{ or } n \neq m$$

$$\frac{\mathcal{E}val[\![x \texttt{=:=} e]\!] \Rightarrow D}{\mathcal{E}val[\![e \texttt{=:=} x]\!] \Rightarrow D} \text{ if } e \text{ is not a variable}$$

$$\overline{\mathcal{E}val[\![x \texttt{=:=} y]\!] \Rightarrow \{\{x \mapsto y\} \,[\!]\, \texttt{True}\}}$$

$$\overline{\mathcal{E}val[\![x \texttt{=:=} c(e_1, \ldots, e_n)]\!] \Rightarrow \{\sigma \,[\!]\, y_1 \texttt{=:=} \sigma(e_1) \,\&\, \ldots \,\&\, y_n \texttt{=:=} \sigma(e_n)\}} \begin{array}{l}\text{if } x \notin cv(c(e_1, \ldots, e_n)),\\ \sigma = \{x \mapsto c(y_1, \ldots, y_n)\},\\ y_1, \ldots, y_n \text{ fresh variables}\end{array}$$

$$\overline{\mathcal{E}val[\![x \texttt{=:=} c(e_1, \ldots, e_n)]\!] \Rightarrow \varnothing} \text{ if } x \in cv(c(e_1, \ldots, e_n))$$

Figure 4: Solving equational constraints

$$\begin{array}{lcl} cv(c(e_1, \ldots, e_n)) & = & cv(e_1) \cup \ldots \cup cv(e_n) \\ cv(f(e_1, \ldots, e_n)) & = & \varnothing \end{array}$$

However, Curry implementations can also provide other methods to solve equational constraints or other types of constraints with appropriate constraint solvers. The only property, which is important for the operational semantics, is the fact that a solved constraint has the form `True`, i.e., a solvable constraint c is reducible to the answer expression $\sigma \,[\!]\, \texttt{True}$ where σ is a solution of c.

D.4 Higher-Order Features

Warren [48] has shown for the case of logic programming that the higher-order features of functional programming can be implemented by providing a (first-order) definition of the application function. Since Curry supports the higher-order features of current functional languages but excludes the guessing of higher-order objects by higher-order unification (as, e.g., in [26, 39, 43]), the operational semantics specified in Figure 2 can be simply extended to cover Curry's higher-order features by adding the following axiom (here we denote by the infix operator "`@`" the application of a function to an expression):

$$\mathcal{E}val[\![\varphi(e_1, \ldots, e_m) \texttt{@} e]\!] \Rightarrow \varphi(e_1, \ldots, e_m, e) \quad \text{if } \varphi \text{ has arity } n \text{ and } m < n$$

Thus, we evaluate an application by simply adding the argument to a partial application. If a function has the right number of arguments, it is evaluated by the rules in Figure 2. Note that the function application suspends if the applied function is unknown (instead of a possible but expensive non-deterministic search for the appropriate function).

λ-abstractions occurring in expressions are anonymous functions and can be implemented by assigning a new name to each λ-abstraction, where the free variables become parameters of the function. For instance, the λ-abstraction (`f` is a free variable)

```
\x y -> f x y + 2
```

can be replaced by the (partial) application `(lambda f)`, where `lambda` is a new name, and adding the rule

```
lambda f x y = f x y + 2
```

Note that this transformation (as well as the extension of $\mathcal{E}val[\![\cdot]\!]$ above) is only used to explain the operational meaning of Curry's higher-order features. A concrete Curry system is free to choose other implementation techniques.

D.5 Generating Definitional Trees

Curry's computation model is based on the specification of a definitional tree for each operation. Although definitional trees are a high-level formalism to specify evaluation strategies, it is tedious to annotate each operation which its definitional tree. Therefore, the user need not write them explicitly in the program since they are automatically inserted by the Curry system. In the following, we present Curry's algorithm to generate definitional trees from the left-hand sides of the functions' rules.

The generation of definitional trees for each function is not straightforward, since there may exist many non-isomorphic definitional trees for a single function representing different evaluation strategies. This demands for a default strategy to generate definitional trees. Curry uses the following default strategy:

1. The leftmost argument, where a constructor occurs at the corresponding position in all left-hand sides defining this function, is evaluated to head normal form.

2. *or* nodes (i.e., disjunctions) are generated in case of a conflict between constructors and variables in the left-hand sides, i.e., if two rules have a variable and a constructor at the same position on the left-hand side.

In the following, we assume that all rules are unconditional (it is obvious how to extend it to conditional rules since only the left-hand sides of the rules are relevant for the definitional trees). To specify the construction algorithm, we define by

$$DP(\pi, R) = \{o \text{ position of a variable in } \pi \mid root(l|_o) \in \mathcal{C} \text{ for some } l = r \in R\}$$

the set of *demanded positions* of a call pattern π w.r.t. a set of rules R. For instance, the demanded positions of the call pattern `leq(x,y)` w.r.t. the rules for the predicate `leq` (see Section D.1, page 75) are $\{1, 2\}$ referring to the pattern variables `x` and `y`. Furthermore, we define by

$$IP(\pi, R) = \{o \in DP(\pi, R) \mid root(l|_o) \in \mathcal{C} \text{ for all } l = r \in R\}$$

the set of *inductive positions* of a call pattern π w.r.t. a set of rules R. Thus, the inductive positions are those demanded positions where a constructor occurs in all left-hand sides defining

this function. For instance, the set of inductive positions of the call pattern `leq(x,y)` w.r.t. the rules for the predicate `leq` is $\{1\}$.

The *g*eneration of a definitional *t*ree for a call pattern π and a set of rules R (where l is an instance of π for each $l = r \in R$) is described by the function $gt(\pi, R)$. We distinguish the following cases for gt:

1. If the position o is leftmost in $IP(\pi, R)$, $\{root(l|_o) \mid l = r \in R\} = \{c_1, \ldots, c_k\}$ where $c_1, \ldots, c_k$ are different constructors with arities $n_1, \ldots, n_k$, and $R_i = \{l = r \in R \mid root(l|_o) = c_i\}$, then

$$gt(\pi, R) = branch(\pi, o, flex, gt(\pi[c_1(x_{11}, \ldots, x_{1n_1})]_o, R_1), \ldots, gt(\pi[c_k(x_{k1}, \ldots, x_{kn_k})]_o, R_k))$$

   where x_{ij} are fresh variables. I.e., we generate a *branch* node for the leftmost inductive position (provided that there exists such a position).

2. If $IP(\pi, R) = \varnothing$, let o be the leftmost position in $DP(\pi, R)$ and $R' = \{l = r \in R \mid root(l|_o) \in \mathcal{C}\}$. Then

$$gt(\pi, R) = or(gt(\pi, R'), gt(\pi, R - R'))$$

   I.e., we generate an *or* node if the leftmost demanded position of the call pattern is not demanded by the left-hand sides of all rules.

3. If $DP(\pi, R) = \varnothing$ and $l = r$ variant of some rule in R with $l = \pi$, then

$$gt(\pi, R) = rule(l = r)$$

   Note that all rules in R are variants of each other if there is no demanded position (this follows from the weak orthogonality of the rewrite system). For non-weakly orthogonal rewrite systems, which may occur in the presence of non-deterministic functions [19] or conditional rules, the rules in R may not be variants. In this case we simply connect the different rules by *or* nodes.

If R is the set of all rules defining the n-ary function f, then a definitional tree for f is generated by computing $gt(f(x_1, \ldots, x_n), R)$.

It is easy to see that this algorithm computes a definitional tree for each function since the number of rules is reduced in each recursive call and it keeps the invariant that the left-hand sides of the current set of rules are always instances of the current call pattern.

Note that the algorithm gt is not strictly conform with the pattern matching strategy of functional languages like Haskell. For instance, it generates for the rules

```
f 0 0 = 0
f x 1 = 0
```

the definitional tree

$$branch(\texttt{f(x1,x2)}, 2, flex, branch(\texttt{f(x1,0)}, 1, flex, rule(\texttt{f(0,0) = 0})), rule(\texttt{f(x1,1) = 0}))$$

$\mathcal{E}val[\![\texttt{try(}g\texttt{)}]\!] \Rightarrow$

$$\begin{cases} \texttt{[]} & \text{if } \mathcal{E}val[\![c]\!] \Rightarrow \varnothing \\ \texttt{[}g'\texttt{]} & \text{if } \mathcal{E}val[\![c]\!] \Rightarrow \{\sigma \,[\!]\, \texttt{True}\} \text{ (i.e., } \sigma \text{ is a mgu for all equations in } c\text{) with} \\ & \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\} \text{ (or } c = \texttt{True} \text{ and } \sigma = id\text{) and} \\ & g' = \texttt{\textbackslash}x\texttt{->let } x_1, \ldots, x_n \texttt{ free in } \sigma \circ \varphi \,[\!]\, \texttt{True} \\ D & \text{if } \mathcal{E}val[\![c]\!] \Rightarrow \{\sigma \,[\!]\, c'\} \text{ with } \mathcal{D}om(\sigma) \subseteq \{x, x_1, \ldots, x_n\}, \\ & g' = \texttt{\textbackslash}x\texttt{->let } x_1, \ldots, x_n, y_1, \ldots, y_m \texttt{ free in } \sigma \circ \varphi \,[\!]\, c' \\ & \text{where } \{y_1, \ldots, y_m\} = \mathcal{VR}an(\sigma) \setminus (\{x, x_1, \ldots, x_n\} \cup free(g)), \\ & \text{and } \mathcal{E}val[\![\texttt{try(}g'\texttt{)}]\!] \Rightarrow D \\ \texttt{[}g_1\texttt{,}\ldots\texttt{,}g_k\texttt{]} & \text{if } \mathcal{E}val[\![c]\!] \Rightarrow \{\sigma_1 \,[\!]\, c_1, \ldots, \sigma_k \,[\!]\, c_k\},\ k > 1, \text{ and, for } i = 1, \ldots, k, \\ & \mathcal{D}om(\sigma_i) \subseteq \{x, x_1, \ldots, x_n\} \text{ and} \\ & g_i = \texttt{\textbackslash}x\texttt{->let } x_1, \ldots, x_n, y_1, \ldots, y_{m_i} \texttt{ free in } \sigma_i \circ \varphi \,[\!]\, c_i \\ & \text{where } \{y_1, \ldots, y_{m_i}\} = \mathcal{VR}an(\sigma_i) \setminus (\{x, x_1, \ldots, x_n\} \cup free(g)) \end{cases}$$

Figure 5: Operational semantics of the `try` operator for $g = \texttt{\textbackslash}x\texttt{->let } x_1, \ldots, x_n \texttt{ free in } \varphi \,[\!]\, c$

(although both arguments are demanded, only the second argument is at an inductive position) whereas Haskell has a strict left-to-right pattern matching strategy which could be expressed by the definitionl tree

$$or(branch(\texttt{f(x1,x2)}, 1, flex, branch(\texttt{f(0,x2)}, 2, flex, rule(\texttt{f(0,0) = 0}))),\\ branch(\texttt{f(x1,x2)}, 2, flex, rule(\texttt{f(x1,1) = 0})))$$

The latter tree is not optimal since it has a non-deterministic *or* node and always requires the evaluation of the first argument (in the first alternative). If the function definitions are *uniform* in the sense of [47], the strategy described by *gt* is identical to traditional functional languages.

D.6 Encapsulated Search

The exact behavior of the `try` operator is specified in Figure 5. Note that a substitution φ of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, computed by evaluating the body of a search goal, must be encoded as a constraint in the new search goal. Therefore, φ is transformed into its *equational representation* $\overline{\varphi}$, where $\overline{\varphi}$ is a solved constraint of the form $x_1\texttt{=:=}t_1\,\texttt{\&}\ldots\texttt{\&}\,x_n\texttt{=:=}t_n$. Hence, a search goal can always be written as `try (\`x`->`$\overline{\varphi}$`&`c`)`. Since the solved constraint $\overline{\varphi}$ does not have to be evaluated again, we use the sugared form `try (\`x`->`$\varphi \,[\!]\, c$`)` where only c is further evaluated. Thus, an initial search goal `try (\`x`->`c`)` is equivalent to `try (\`x`->`$id \,[\!]\, c$`)`.

A search goal is solved (second case) if the constraint is solvable without bindings of global variables. In a deterministic step (third case), we apply the `try` operator again after adding the newly introduced variables to the list of local variables. Note that the *free variables* occurring in g, denoted by $free(g)$, must not be locally declared because they can appear also outside of g, and therefore they have to be removed from $\mathcal{VR}an(\sigma)$. In a non-deterministic step (fourth case), we return the different alternatives as the result. Note that the evaluation of the `try` operator suspends if a computation step on the constraint attempts to bind a free variable. In order to ensure that an encapsulated search will not be suspended due to necessary bindings of free variables, the search goal should be a closed expression when a search operator is applied to it, i.e., the search variable

is bound by the lambda abstraction and all other free variables are existentially quantified by local declarations.

If one wants to use free variables in a search goal, e.g., for synchronization in systems where concurrency is intensively used, the goal should perform only deterministic steps until all free variables have been bound by another part of the computation. Otherwise, a non-deterministic step could unnecessarily split the search goal. For instance, consider the expression

```
try (\x -> y=:=[x] & append x [0,1] =:= [0,1])
```

The computation of `y=:=[x]` suspends because `y` is free. Therefore, a concurrent computation of the conjunction will split the goal by reducing the `append` expression. However, if `y` is first bound to `[[]]` by another thread of the computation, the search goal could proceed deterministically.

To avoid such unnecessary splittings, we can restrict the applicability of the split case and adapt a solution known as *stability* from AKL [29] and Oz [44]. For this purpose, we can slightly change the definition of `try` such that non-deterministic steps lead to a suspension as long as a deterministic step might be enabled by another part of the computation, i.e., the search goal contains free global variables. To do so, we have to replace only the fourth rule of the `try` operator:

$$\begin{array}{ll} \mathcal{E}val[\![\texttt{try(}g\texttt{)}]\!] \Rightarrow \texttt{[}g_1\texttt{,}\ldots\texttt{,}g_k\texttt{]} & \text{if } \mathcal{E}val[\![c]\!] \Rightarrow \{\sigma_1 \,[\!]\, c_1, \ldots, \sigma_k \,[\!]\, c_k\},\ k > 1, \\ & \mathcal{E}val[\![c]\!] \not\Rightarrow \{\sigma \,[\!]\, c'\}, free(c) \subseteq \{x, x_1, \ldots, x_n\} \\ & \text{and, for } i = 1, \ldots, k, \\ & g_i = \texttt{\textbackslash}x\texttt{->let } x_1, \ldots, x_n, y_1, \ldots, y_{m_i} \texttt{ free in } \sigma_i \circ \varphi \,[\!]\, c_i \\ & \text{where } y_1, \ldots, y_{m_i} = \mathcal{VR}an(\sigma_i) \setminus (\{x, x_1, \ldots, x_n\} \cup free(g)) \end{array}$$

Thus, a non-deteterministic step is only performed if no deterministic step is possible and the unsolved constraint c contains no free variables except those locally declared in the search goal. Global variables appearing in the data terms t_i of the substitution $\varphi = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ but not in c are not considered by these conditions because they cannot influence the further evaluation of c. Note that the former condition $\mathcal{D}om(\sigma_i) \subseteq \{x, x_1, \ldots, x_n\}$ is covered by the new condition $free(c) \subseteq \{x, x_1, \ldots, x_n\}$ because $\mathcal{D}om(\sigma_i)$ is always a subset of $free(c)$.

Note that the first definition of `try` in Figure 5 computes non-deterministic splittings where the latter definition suspends. On the other hand, the latter definition can avoid some unnecessary splittings. Thus, different implementations of Curry can support one or both of these definitions.

D.7 Eliminating Local Declarations

In the description of the operational semantics above, we assumed that a Curry program is a set of rules l`=`r where the left-hand side l is a call pattern and the right-hand side r is an expression containing variables, constructors, and defined functions. However, a Curry program can also contain local declarations (cf. Section 2.4) whose operational semantics is not defined so far. To simplify the operational semantics, we do not extend it to local declarations but provide in the following a transformation of Curry programs containing local declarations into Curry programs without local declarations (except for free variables). The main purpose of this transformation is to provide a precise definition of the operational semantics of the full language. This transformation can also be performed by implementations of Curry but it is also possible that some implementations provide explicit support for local declarations, provided that they satisfy the operational meaning described in the following.

The elimination of local function and pattern declarations is done by the following steps which are subsequently described in more detail:

1. Eliminate sections and lambda abstractions
2. Eliminate Boolean guards in rules
3. Transform `where` declarations into `let` declarations
4. Eliminate `fcase`
5. Eliminate local patterns and functions

For the following, we assume that all name conflicts have been resolved, i.e., the names of functions and argument variables defined in the different declarations are pairwise different.

D.7.1 Eliminate Sections and Lambda Abstractions

Left and right sections are transformed into partial applications, i.e.,

`(`*expr op*`)` is transformed into `(\f x y -> f x y) (`*op*`)` *expr*

`(`*op expr*`)` is transformed into `(\f x y -> f y x) (`*op*`)` *expr*

Note that a transformation of a right section `(`*op expr*`)` into the lambda abstraction `(\`*x* `->` `(`*op*`)` *x expr*`)` (where *x* is a new variable) would have a different behavior w.r.t. sharing of subterms. A compiler could avoid the introduction of lambda abstractions by transforming a left section `(`*expr op*`)` into the partial application `((`*op*`)` *expr*`)` and a *right section* `(`*op expr*`)` into `(Prelude.flip (`*op*`)` *expr*`)`.

All λ-abstractions are eliminated by providing a new name (cf. Section D.4), i.e., each λ-abstraction `(\`$p_1 \ldots p_n$`->`e`)` is replaced by the expression `(let` f $p_1 \ldots p_n$ `=` e `in` f`)` where f is a new function name. This transformation must be recursively applied to all λ-abstractions occurring in e.

D.7.2 Eliminate Boolean Guards

This is done according the meaning described in Section 2.3.2. A rule of the form

$f\ t_1 \ldots t_n$ `|` b_1 `=` e_1
⋮
`|` b_k `=` e_k
`where` *decls*

(the `where` part can also be missing), where all guards $b_1, \ldots, b_k$ ($k > 0$) are expressions of type `Bool`, is transformed into the single rule

$f\ t_1 \ldots t_n$ `=` `if` b_1 `then` e_1 `else`
⋮
`if` b_k `then` e_k `else failed`
`where` *decls*

D.7.3 Transform `where` into `let`

Each unconditional rule of the form

l `=` r `where` $decls$

is transformed into

l `=` `let` $decls$ `in` r `.`

Each conditional rule of the form

l `|` c `=` r [`where` $decls$]

is transformed into

l `=` [`let` $decls$ `in`] (`c` `Prelude.&>` r)

Thus, we assume in the subsequent transformations that all program rules are of the form "l `=` r" (where r might be a let-expression).

Note that this transformation is not really necessary but provides for a unified treatment of the elimination of local pattern and function declarations in `let` and `where`.

D.7.4 Eliminate `fcase`

As discussed in Section 5.4, `fcase` expressions are a short notation for flexible pattern matching as used in defined functions. Thus, each `fcase` expression can be transformed into a let-expression defining a local function that is subsequently turned into a global function (see below). Hence, every expression of the form

`fcase` e `of`
  p_1 `|` g_1 `->` $e1$
  ⋮
  p_n `|` g_n `->` en

(where some of the guards "`|` g_j" can be omitted) is translated into the expression

`let` f p_1 `|` g_1 `=` $e1$
    ⋮
    f p_n `|` g_n `=` en
 `in` f e

where f is a fresh function symbol not occurring in the initial `fcase` expression.

D.7.5 Eliminate Local Patterns and Functions

All local pattern and function declarations in a rule "l `=` r" are eliminated by iterating the elimination of outermost local declarations as follows. For this purpose, we denote by $r = C[\texttt{let}\ decls\ \texttt{in}\ e]_p$ an outermost `let` declaration in r, i.e., there is no prefix position $p' < p$ in r with $r|_{p'} = \texttt{let}\ldots$

We apply the following transformations as long as possible to eliminate all local pattern and function declarations in a rule of the form "l `=` $C[\texttt{let}\ decls\ \texttt{in}\ e]_p$":

Eliminate Patterns Select a local pattern declaration which contains only argument variables from the main function's left-hand side in the expression, i.e., the rule has the form

$$
\begin{array}{l}
l \texttt{ = } C[\texttt{let } decls_1 \\
\qquad\qquad p \texttt{ = } e' \\
\qquad\qquad decls_2 \\
\qquad \texttt{in } e]_p
\end{array}
$$

with $free(e') \subseteq free(l)$ (it is a programming error if no pattern declaration has this property, i.e., cyclic pattern definitions or pattern definitions depending on locally free variables are not allowed). Then transform this rule into the rules

$$
\begin{array}{l}
l \texttt{ = } C[f'\ x_1 \ldots x_k\ e']_p \\
f'\ x_1 \ldots x_k\ z \texttt{ = } f''\ x_1 \ldots x_k\ (f_1 z) \ldots (f_m z) \\
f''\ x_1 \ldots x_k\ y_1 \ldots y_m \texttt{ = let } decls_1 \\
\qquad\qquad\qquad\qquad\qquad\quad decls_2 \\
\qquad\qquad\qquad\qquad\quad \texttt{in } e \\
f_1 p \texttt{ = } y_1 \\
\ldots \\
f_m p \texttt{ = } y_m
\end{array}
$$

where $x_1 \ldots x_k$ are all the variables occurring in l, $y_1 \ldots y_m$ are all the variables occurring in p, z is a new variable symbol, and $f', f'', f_1, \ldots, f_m$ are new function symbols. Repeat this step for f'' until all local pattern declarations are eliminated.

This translation can be optimized in some specific cases. If p is just a variable, the function f' is not needed and the definition of l can be simplified into

$$l \texttt{ = } C[f''\ x_1 \ldots x_k\ e']_p$$

Similarly, if e' is a variable, the function f' is also not needed and the definition of l can be replaced by

$$l \texttt{ = } C[f''\ x_1 \ldots x_k\ (f_1 e') \ldots (f_m e')]_p$$

Complete Local Function Definitions If a locally declared function f refers in its definition to a variable v not contained in its argument patterns, then add v as an additional first[20] argument to all occurrences of f (i.e., left-hand sides and calls). Repeat this step until all locally defined functions are completed. Note that this completion step must also be applied to free variables introduced in the same `let` expression, i.e., the rule

```
f x = let y free
          g z = c y z
      in g 0
```

is completed to

```
f x = let y free
          g y z = c y z
      in g y 0
```

[20]The new arguments must come first because of possible partial applications of this function.

Globalize Local Function Definitions If the definitions of all locally declared functions are completed, i.e., the definitions only refer to variables in the argument patterns, delete the locally declared functions and define them at the top level (and rename them if there are already top-level functions with the same name). For instance, the previous rule is transformed into the definitions

```
g y z = c y z
f x = let y free in g y 0
```

Note that the entire transformation process must be applied again to the new top-level declarations since they may also contain local declarations.

D.7.6 Resulting Program

After applying these transformation steps to all rules in the program, we obtain a program without sections and λ-abstractions where all local declarations contain only free variables. Note that partial applications are not eliminated since they can be treated as shown in Section D.4.

References

[1] H. Aït-Kaci. An overview of life. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.

[2] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.

[3] 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.

[4] S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[5] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[6] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press, 1997.

[7] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[8] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[9] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. European Symposium on Programming*, pages 119–132. Springer LNCS 213, 1986.

[10] S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.

[11] J. Boye. S-sld-resolution – an operational semantics for logic programs with external procedures. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 383–393. Springer LNCS 528, 1991.

[12] S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Concurrency in functional and logic programming. In *Fuji International Workshop on Functional and Logic Programming*. World Scientific Publ., 1995.

[13] M. Chakravarty, Y. Guo, M. Köhler, and H. Lock. Goffin - higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.

[14] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[16] R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340, 1998.

[17] L. Fribourg. Slog: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 172–184, Boston, 1985.

[18] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.

[19] J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[20] J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[21] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

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

[23] M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[24] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[25] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[26] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[27] M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.

[28] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[29] S. Janson and S. Haridi. Programming paradigms of the andorra kernel language. In *Proc. 1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[30] T. Johnsson. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer LNCS 201, 1985.

[31] H. Kuchen, R. Loogen, J. Moreno-Navarro, and M. Rodríguez-Artalejo. The functional logic language babel and its implementation on a graph machine. *New Generation Computing*, 14:391–427, 1996.

[32] J. Lloyd. Combining functional and logic programming languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.

[33] J. Lloyd. Declarative programming in escher. Technical report cstr-95-013, University of Bristol, 1995.

[34] R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.

[35] R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. *Theoretical Computer Science 142*, pages 59–87, 1995.

[36] F. López Fraguas. A general scheme for constraint functional logic programming. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 213–227. Springer LNCS 632, 1992.

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

[38] J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language babel. *Journal of Logic Programming*, 12:191–223, 1992.

[39] G. Nadathur and D. Miller. An overview of λprolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 810–827. MIT Press, 1988.

[40] L. Naish. Adding equations to nu-prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.

[41] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[42] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[43] C. Prehofer. *Solving Higher-order Equations: From Logic to Programming*. PhD thesis, TU München, 1995. Also appeared as Technical Report I9508.

[44] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[45] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.

[46] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Rewriting Techniques and Applications (RTA'95)*, pages 179–193. Springer LNCS 914, 1995.

[47] P. Wadler. Efficient compilation of pattern-matching. In S. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

[48] D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

Index