Functional Logic Programming Language Curry

Xiang Yin

Department of Computer Science
McMaster University

November 9, 2010

Outline

1. Functional Logic Programming Language
2. Overview of Curry
3. Main Features of Curry
4. Curry vs Other Languages

Functional Programming Language

Functional Programming

- Treats computation as the evaluation of mathematical functions
- Avoids state and mutable data
- The basis is Lambda Calculus

Features

- Nested Expressions
- Lazy Evaluation
- High-order Functions

Logic Programming Language

Logic Programming

- Defines what rules of the solution should satisfy rather than how many procedures should do for obtaining the solution
- Use of mathematical logic for computer programming
- Use of logic as both a declarative and procedural representation language

Features

- Logical Variables
- Partial Data Structures
- Built-in Search
- Residuation and Narrowing

Functional Logic Programming Language

Functional Logic Programming Language - Curry

Curry is a general-purpose declarative programming language that integrates functional with logic programming.

Benefits

- vs pure functional languages: more expressive power due to the availability of features like function inversion, partial data structures, existential variables, and non-deterministic search.
- vs pure logic languages: more efficient operational behavior since functions provide for more efficient evaluation strategies (lazy evaluation, deterministic reductions) than predicates.

Implementation of Curry

- **PAKCS** (Portland Aachen Kiel Curry System): a major Curry implementation with a WWW interface
- **MCC** (Münster Curry Complier): is a mature native code compiler for Curry which conforms to the Curry report except for committed choice which is not supported
- **KiCS** (Kiel Curry System): a compiler that translates Curry into Haskell
- **Sloth**: a compiler which translates Curry programs into Prolog programs

Predefined Types

Type	Declaration	Examples
Integer	Int	$\ldots, -2, -1, 0, 1, 2, \ldots$
Boolean	Bool	False, True
Character	Char	'a', 'b', 'c', ...
String	String	"hello", "world"
List of τ	$[\tau]$	$[], [0, 1, 2], 0 : 1 : 2 : []$
Success	Success	success
Unit	()	()

Predefined Types

Success Type

The type Success has no visible literal values and is intended to denote the result of successfully solved constraints.

Unit Type

- Type Theory: a type that allows only one value (and thus can hold no information)
- Curry: useful in situations where the return value of a function is not important
- Haskell: the Nothing in polymorphic type Maybe is isomorphic to the unit type
- C, C++, C#, Java : Void Type

Predefined Types

Success Type

The type Success has no visible literal values and is intended to denote the result of successfully solved constraints.

Unit Type

- Type Theory: a type that allows only one value (and thus can hold no information)
- Curry: useful in situations where the return value of a function is not important
- Haskell: the Nothing in polymorphic type Maybe is isomorphic to the unit type
- C, C++, C#, Java : Void Type

Predefined Types

Success Type

The type Success has no visible literal values and is intended to denote the result of successfully solved constraints.

Unit Type

- Type Theory: a type that allows only one value (and thus can hold no information)
- Curry: useful in situations where the return value of a function is not important
- Haskell: the Nothing in polymorphic type Maybe is isomorphic to the unit type
- C, C++, C#, Java : Void Type

Predefined Types

Success Type

The type Success has no visible literal values and is intended to denote the result of successfully solved constraints.

Unit Type

- Type Theory: a type that allows only one value (and thus can hold no information)
- Curry: useful in situations where the return value of a function is not important
- Haskell: the Nothing in polymorphic type Maybe is isomorphic to the unit type
- C, C++, C#, Java : Void Type

Predefined Operations

Description	Ident.	Type
Boolean Equality	==	$a \to a \to Bool$
Constrained Equality	=:=	$a \to a \to Success$
Boolean Conjunction	&&	$Bool \to Bool \to Bool$
Boolean Disjunction	||	$Bool \to Bool \to Bool$
Parallel Conjunction	&	$Success \to Success \to Success$
Constrained Expression	&>	$Success \to a \to a$

Predefined Operations

Parallel Conjunction(&)

The parallel conjunction applied to expressions *u* and *v*, i.e., *u* & *v*, evaluates *u* and *v* concurrently. If both succeeds, the evaluation succeeds; otherwise it fails.

Constrained Expression(&>)

The constrained expression applied to a constraint *c* and an expression *e*, i.e., *c* &> *e*, evaluates first *c* and, if this evaluation succeeds, then *e*, otherwise it fails.

Expressions

Definition

An expression is either a symbol or literal value or is the application of an expression to another expression.

- Prelude>$3 + 5 * 4$
  Result: 23 ?
- Prelude> $3 + 5 * 4 >= 3 * (4 + 2)$
  Result: True ?
- Prelude> $4 + 3 == 8$
  Result: False ?

Expressions

Definition

An expression is either a symbol or literal value or is the application of an expression to another expression.

- Prelude>$3 + 5 * 4$
  Result: 23 ?
- Prelude> $3 + 5 * 4 >= 3 * (4 + 2)$
  Result: True ?
- Prelude> $4 + 3 == 8$
  Result: False ?

Expressions

Definition

An expression is either a symbol or literal value or is the application of an expression to another expression.

- Prelude>$3 + 5 * 4$
  Result: 23 ?
- Prelude> $3 + 5 * 4 >= 3 * (4 + 2)$
  Result: True ?
- Prelude> $4 + 3 == 8$
  Result: False ?

Functions

Definition

A function is a device that takes arguments and returns a result. The result is obtained by evaluating an expression which generally involves the function's arguments.

- constant function: $nine = 3 * 3$
- function with argument: square $x = x * x$
- **Remarks:** The syntax of Curry is actually very similar to Haskell.

Functions

Definition

A function is a device that takes arguments and returns a result. The result is obtained by evaluating an expression which generally involves the function's arguments.

- constant function: $nine = 3 * 3$
- function with argument: square $x = x * x$
- **Remarks:** The syntax of Curry is actually very similar to Haskell.

Functions

Definition

A function is a device that takes arguments and returns a result. The result is obtained by evaluating an expression which generally involves the function's arguments.

- constant function: $nine = 3 * 3$
- function with argument: square $x = x * x$
- **Remarks:** The syntax of Curry is actually very similar to Haskell.

Functions

Load and Reload

If we write the definitions of nine and square with a standard text editor into a file named “firstprog.curry”:

```
Prelude> :l firstprog
firstprog>
firstprog> square nine
Result: 81 ?
```

If we add the definition “two = 2” to the file and we want to reload it:

```
firstprog> :r
firstprog> square (square two)
Result: 16 ?
```

Pattern Matching

Pattern Matching

The definition of a function can be broken into several rules. This feature allows a function to dispatch the expression to be returned depending on the values of its arguments.

- not x = if x == True then False
  else True
- not :: *Bool* → *Bool*
  not False = True
  not True = False
- **Remarks:** It is quite useful to simplify the complex data structures.

Pattern Matching

Pattern Matching

The definition of a function can be broken into several rules. This feature allows a function to dispatch the expression to be returned depending on the values of its arguments.

- not x = if x == True then False
  else True
- not :: *Bool* $\rightarrow$ *Bool*
  not False = True
  not True = False
- **Remarks:** It is quite useful to simplify the complex data structures.

Pattern Matching

Pattern Matching

The definition of a function can be broken into several rules. This feature allows a function to dispatch the expression to be returned depending on the values of its arguments.

- not x = if x == True then False
  else True
- not :: $Bool \rightarrow Bool$
  not False = True
  not True = False
- **Remarks:** It is quite useful to simplify the complex data structures.

Higher-Order Computation

Definition

A function that takes an argument of function type is referred to as a higher-order function. Loosely speaking, a higher-order function is computation parameterized by another computation.

Example

```
sort _ [] = []
sort f (x:xs) = insert f x (sort f xs)
insert _ x [] = [x]
insert f x (y:ys) | f x y = x : y : ys
                  | otherwise = y : insert f x ys
```

sort (<=) [3,5,1,2,6,8,9,7]

Result: [1,2,3,5,6,7,8,9] ?

Lazy Evaluation

Definition

Lazy Evaluation is the technique of delaying a computation until the result is required. More details:

- The evaluation of an expression t is the process of obtaining a value v from t.
- The value v is obtained from t by replacing an instance of the left-hand side of a rule with the corresponding instance of the right-hand side.
- The evaluation of an expression t proceeds replacement after replacement until an expression v in which no more replacements are possible is obtained.

Lazy Evaluation

Example1

square $x = x * x$
an instance of square x is replaced with the corresponding instance of $x * x$. For example, $4 + square(2 + 3)$ is replaced by $4 + (2 + 3) * (2 + 3)$.

Example2

If v is not a value: fails; otherwise v is the result.
For example, the following function head computes the first element of a (non-null) list: head (x:_) = x
An attempt to evaluate “head []” fails, since no replacement is possible and the expression is not a value since it contains a function.

Lazy Evaluation

Benefits

- Performance increases due to avoiding unnecessary calculations
- Avoiding error conditions in the evaluation of compound expressions
- The capability of constructing potentially infinite data structures
- The capability of defining control structures as abstractions instead of as primitives

Logic Variables

Definition

A logic variable is either a variable occurring in an expression typed by the user at the interpreter prompt or it is a variable in the condition and/or right-hand side of a rewrite rule which does not occur in the left-hand side.

Case One

Prelude> z == 2 + 2 where z free

Case Two

path a z = edge a b && path b z where b free

Curry vs Haskell

- It is nearly a superset of Haskell, lacking support mostly for overloading using type classes, which some implementations provide anyway as a language extension, such as the MCC.
- It contains some special features, such as Search For Solutions, and Residuation and Narrowing.

Curry vs Haskell

Search For Solutions

One of the distinguishing features of Curry in comparison to Haskell is its ability to search for solutions, i.e., to compute values for the arguments of functions so that the functions can be evaluated.

Curry vs Haskell

Prelude> x &&(y || (not x)) where x,y

- free Free variables in goal: x, y
  Result: True
  Bindings:
  x=True
  y=True ? ;
- Result: False
  Bindings: x=True
  y=False ? ;
- Result: False
  Bindings: x=False
  y=y ? ;
- No more solutions.

Curry vs Haskell

Prelude> x &&(y || (not x)) where x,y

- free Free variables in goal: x, y
  Result: True
  Bindings:
  x=True
  y=True ? ;
- Result: False
  Bindings: x=True
  y=False ? ;
- Result: False
  Bindings: x=False
  y=y ? ;
- No more solutions.

Curry vs Haskell

Prelude> x &&(y || (not x)) where x,y

- free Free variables in goal: x, y
  Result: True
  Bindings:
  x=True
  y=True ? ;
- Result: False
  Bindings: x=True
  y=False ? ;
- Result: False
  Bindings: x=False
  y=y ? ;
- No more solutions.

Curry vs Haskell

Prelude> x &&(y || (not x)) where x,y

- free Free variables in goal: x, y
  Result: True
  Bindings:
  x=True
  y=True ? ;
- Result: False
  Bindings: x=True
  y=False ? ;
- Result: False
  Bindings: x=False
  y=y ? ;
- No more solutions.

Residuation and Narrowing

Residuation

Let e be an expression to evaluate and v a variable occurring in e. Suppose that e cannot be evaluated because the value of v is not known. Residuation suspends the evaluation of e. If it is possible, we will address this possibility shortly, some other expression f is evaluated in hopes that the evaluation of f will bind a value to v. If and when this happens, the evaluation of e resumes. If the expression f does not exists, e is said to *flounder* and the evaluation of e fails.

Example

```
Prelude> z == 2+2 where z free
Free variables in goal: z
*** Goal suspended!
```

Residuation and Narrowing

Narrowing

Narrowing is a mechanism whereby a variable is bound to a value selected from among alternatives imposed by constraints. Each possible value is tried in some order, with the remainder of the program invoked in each case to determine the validity of the binding.

Remarks

By contrast to residuation, if e cannot be evaluated because the value of v is not known, narrowing guesses a value for v. The guessed value is uninformed except that only values that make it possible to continue the computation are chosen.

Narrowing

Example

Prelude> z =:= 2+2 where z free
Free variables in goal: z
Result: success
Bindings:
z=4 ?

Remarks

The difference between Boolean Equality (==) and Constrained Equality (=:=) are in two aspects:

- The type returned by the operation.
- The operation =:= does narrowing instead of == does residuation

Curry vs Prolog

By the availability of several new features in comparison to pure logic programming, Curry avoids the following impure construct of Prolog:

- The call predicate is replaced by the higher-order features of Curry.
- Many applications of **assert** and **retract** can be eliminated using implications in conditions
- The I/O operations of Prolog are replaced by the declarative monadic I/O concept of functional programming.

Summary

1. Features of Functional Logic Programming Language:
   - Functional Programming Language
   - Logic Programming Language
2. Features of Curry:
   - Predefined Types and Operations
   - Expressions and Functions
   - Features From Functional Language: Pattern Matching, Higher-Order Computation and Lazy Evaluation
   - Features From Logic Language: Logic Variables, Search For Solutions, and Residuation and Narrowing
3. Curry vs Haskell && Curry vs Prolog

References I

Sergio Antoy, Michael Hanus.
Curry A Tutorial Introduction.
Draft of December 5, 2007.

Michael Hanus, Herbert Kuchen, Juan José Moreno-Navarro.
Curry: A Truly Functional Logic Language.
1995.

Wiki
Curry (programming language).
Available:`http://en.wikipedia.org/wiki/Curry_(programming_language)`

References II

Curry (Home Page)
A Truly Integrated Functional Logic Language.
Available:`http://www-ps.informatik.uni-kiel.de/currywiki/`

Thank You !
Question ?