Adding Linear Constrains over Real Numbers to Curry

Wolfgang Lux

Universität Münster
wlux@uni-muenster.de

Abstract. Curry is a multi-paradigm declarative languages that integrates features from functional languages, logic languages, and concurrent programming. In this paper we present an extension of Curry which adds linear constraints over real numbers to the language. The basic kernel of Curry already provides equational constraints and adding linear inequality constraints is a natural task. However, the integration of functions which compute minimal or maximal solutions with respect to a set of constraints must be handled differently as they involve the manipulation of a hidden state. To give a sound semantics to these functions a constraint solving monad `C` is introduced.

1 Introduction

Curry [Han00] is a multi-paradigm language that integrates features from functional languages, logic languages, and concurrent programming. The operational model of Curry is based on an optimal reduction strategy [AEH97] which integrates needed narrowing and residuation. Narrowing [Red85] combines unification and reduction allowing the non-deterministic instantiation of unbound logic variables. The residuation strategy [ALN87], on the other hand, delays the evaluation of expressions containing unbound logic variables until these are sufficiently instantiated by other parts of the program.

Constraint logic programming languages replace unification by the concept of constraint resolution. In a constraint logic calculus, only clauses whose constraints are consistent will be selected in a resolution step. In general, less clauses will be selected, reducing the amount of search in the program. This results in a more efficient execution of constraint logic programs compared to pure logic programs.

Constraint solving techniques have proven useful in various domains, including equality constraints for infinite terms, linear constraints over finite domains, rational and real numbers, and boolean constraints.

In this paper we will explore the integration of linear constraints over real numbers into Curry. We have chosen this domain because there is a wide variation of application domains, e.g. financial applications, numerical simulations, or task scheduling. In addition there are well known, efficient algorithms for solving equalities and inequalities between linear expressions, namely Gaussian elimination and the simplex algorithm.

This paper is organized as follows. The next section briefly summarizes the computation model of Curry. The third section introduces linear constraints over real numbers. Section 4 outlines the integration of linear constraints in Curry. The fifth section presents related work and the final section concludes.

2 The computation model of Curry

Curry uses a syntax similar to Haskell [HPW92], but with a few additions.

The basic computational domain of Curry is a set of data terms. A data term t is either a variable x or an application of an n-ary data constructor to n argument terms.

$$t ::= x \mid c\, t_1 \cdots t_n$$

A data term is *ground* if it contains no variable.

New data constructors can be introduced through data type declarations. E.g., the type `Nat` of natural numbers can be defined by

```
data Nat = Zero | Succ Nat
```

The nullary data constructor `Zero` represents the natural number 0 and the unary data constructor `Succ` applied to some other natural number denotes the successor of that number.

An expression e is either a variable x, a data constructor c, a defined function f, or a (partial) application of an expression e_1 to an argument expression e_2.

$$e ::= x \mid c \mid f \mid e_1\, e_2$$

Curry employs a Hindley-Milner type discipline, therefore e_1 will always be an expression that accepts an argument of the type of e_2.

Functions are defined by a set of conditional equation of the form

$$f\, t_1 \cdots t_n \mid g = e$$

where the argument patterns t_i are terms and the so-called guard g and the body e are expressions.

A conditional equation for the function f is applicable in an expression $f\, e_1 \cdots e_n$ if each argument expression e_i matches the corresponding pattern t_i and the guard g is satisfied (see below). In that case the application is replaced by the body of the function with all variables occurring in the patterns t_i replaced by the expressions in the corresponding positions of the arguments e_i. If an argument expression contains an unbound logic variable at a position where a constructor term occurs in pattern t_i, the variable is either instantiated with the corresponding pattern (narrowing) or the evaluation of the expression is suspended until this variable is instantiated by other parts of the program (residuation).

The evaluation strategy can be specified with an evaluation annotation

f `eval flex` or f `eval rigid`

In the former case f uses narrowing, in the latter case f uses residuation. Curry provides reasonable defaults depending on the result type of the function, so that the evaluation strategy rarely needs to be specified explicitly.

A where clause may added to the right hand side of an equation to provide local definitions whose scope is the guard g and the body e of the equation. Unbound logic variables may be introduced by the special syntax `where` $x_1, \ldots, x_n$ `free`.[1]

The addition function for natural numbers can be defined as follows.

```
add Zero n = n
add (Succ m) n = Succ (add m n)
```

[1] This syntax can be used in `let`-expressions as well.

Curry distinguishes between boolean expressions, which are of type `Bool`, and expressions of type `Success`. While the former may reduce to the predefined constant terms `True` or `False`, the latter are checked for satisfiability. Usually, predicates (in the sense of Prolog) are implemented as functions with result type `Success` in Curry. These function use narrowing by default.

A guard expression in a conditional equation must be either of type `Success` or of type `Bool`. In the former case, the guard is satisfied if the guard expression g is satisfied. In the latter case, the guard is satisfied if the expression g reduces to `True`.

The predefined nullary function `success` of type `Success` is an expression that is always satisfied. The binary operator `&` denotes the concurrent conjunction of two expressions of type `Success`. The expression e_1 `&` e_2 is satisfied if both arguments are satisfied. The order of evaluation of the two arguments e_1 and e_2 is unspecified. An implementation of Curry may choose to evaluate e_1 before e_2 or vice versa or may even interleave the evaluation of both expressions.

A predicate that is satisfied for all even natural numbers can be defined as follows in Curry.

```
even Zero = success
even (Succ (Succ n)) = even n
```

Curry implements strict equality with the binary function

```
(==) :: a -> a -> Bool
```

An expression e_1 `==` e_2 reduces to `True` if both arguments can be reduced to the same finite data term and `False` if they reduce to different data terms. If the arguments contain unbound variables, the evaluation is delayed until these variables are sufficiently instantiated.

The basic kernel of Curry also provides equational constraints. An equational constraint e_1 `=:=` e_2 is an expression of type `Success`. This expression is satisfied if both arguments can be reduced to the same finite data term. If e_1 or e_2 contain unbound logic variables, an attempt is made to unify both expressions by instantiating variables to terms. If the unification succeeds, the constraint is satisfied.

In terms of a constraint programming language, `=:=` corresponds to a tell constraint and `==` corresponds to an ask constraint.

Floating point numbers in Curry are of type `Float`. The usual arithmetic operations and compare functions are available as binary operators:

```
(+),(-),(*),(/) :: Float -> Float -> Float
(<),(>),(<=),(>=) :: Float -> Float -> Bool
```

3 Linear constraints over Real Numbers

A linear expression is either a number n, a variable x, the product of a number and a variable or a sum of linear expressions.

$$l ::= n \mid x \mid n \cdot x \mid l + l$$

A linear constraint is an (in-)equality between two linear expressions. It can always be transformed into a canonical form

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \odot b$$

where all a_i and b are numbers, all x_n are distinct variables and $\odot$ is a relation from the set $\{=, <, >, \leq, \geq\}$. In the case of real numbers, Gaussian elimination can be used to solve linear constraints involving only equalities and the simplex algorithm can be used if inequalities are included.

As an example, consider the simple task scheduling problem given in Fig. 1. Every rounded box denotes a task with a unique name and

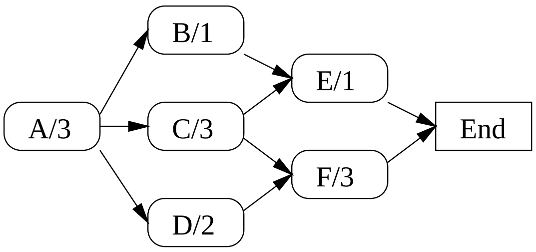

Fig. 1. A simple task graph

a duration. The arrows between the boxes indicate a dependency between two tasks, such that the task at the base of the arrow must have finished before the task at the tip of the arrow may begin. Thus, task E cannot begin before the tasks B and C are completed. This task graph leads to the following set of constraints.

$$\begin{array}{llll} a+3 \leq b & b+1 \leq e & c+3 \leq f & e+1 \leq end \\ a+3 \leq c & c+3 \leq e & d+2 \leq f & f+3 \leq end \\ a+3 \leq d & & & \end{array}$$

4 Linear constraints in Curry

In order to integrate linear constraints into Curry, we assume that the state of the program is augmented by a constraint store, which collects all (linear) constraints in the program. A constraint solver is used to simplify the constraint store and to check its consistency.

We propose to add binary constraint functions

```
(<:),(>:),(<:=),(>:=) :: Float -> Float -> Success
```

in order to add new constraints to the constraint store. If the arguments in an expression $e_1 \odot e_2$ can be reduced to a linear expression, where $\odot$ is one of the constraint functions, the corresponding inequality is added to the constraint store and the constraint solver is used to simplify the resulting store. If the constraint store remains consistent after this operation, the expression is satisfied, otherwise the computation fails. If the arguments can be reduced to numbers, the constraints are satisfied, if the corresponding inequality holds between the numbers.

Equational constraints have to be modified, such that e_1 `=:=` e_2 adds a new constraint to the store if e_1 and e_2 reduce to linear expressions instead of data terms.

With the help of these functions, the example schedule from Fig. 1 can be expressed as a function

```
schedule (a,b,c,d,e,f,end) =
  a+3 <:= b & a+3 <:= c & a+3 <:= d &
  b+1 <:= e & c+3 <:= e & c+3 <:= f & d+2 <:= f &
  e+1 <:= end & f+3 <:= end
```

A valid schedule can be computed by non-deterministically instantiating variables with floating-point numbers.

```
solution (a,b,c,d,e,f,end) =
  schedule (a,b,c,d,e,f,end) & a =:= 0 &
  enumerate 0.0 b & enumerate 0.0 c & enumerate 0.0 d &
  enumerate 0.0 e & enumerate 0.0 f & enumerate 3.0 end

enumerate l x = x =:= l
enumerate l x = enumerate (l+1) x
```

Usually the programmer is interested in finding the minimal or maximal solution admissible for a variable with respect to a given set of constraints. Curry's encapsulated search [HS98] can be used for this purpose. Using a depth-first strategy we need to compute only the first solution to determine the minimal schedule.

However, this approach does not make much use of the fact that the constraints restrict the search space for the solution. In addition, the minimum and maximum can be computed easily with the help of the simplex algorithm. Thus, we would like to add functions

```
minimum :: Float -> Float
maximum :: Float -> Float
```

returning the minimal or maximal value, resp., for their argument, which is consistent with the constraint store.[2]

Unfortunately, there is no declarative reading for such functions because the result depends on the order of evaluation. As an example, consider the constraints given by

```
x >:= 0 & y >:= 0 & x+2*y >:= 10 &
y <:= 3 & z =:= minimum x
```

When evaluated from left to right, `z` will be bound to 4. However, if `minimum` is evaluated before the constraint `y <:= 3` is added to the constraint store, the result of `minimize x` will be 0.

The problem is that `minimum` and `maximum` compute a value with reference to an implicit state. The usual solution for the implementation of computations involving state in an programming language with a lazy evaluation strategy is to use a monad which encapsulates

[2] If the argument is not a variable, a fresh unbound variable is introduced, and an equational constraint for that variable and the given expression is added to the constraint store before computing the minimum or maximum.

the state [PW93]. In Curry, the `IO`-monad is available for that purpose. However, by making `minimum` and `maximum` a function in the `IO` monad, all computations that require the minimum or maximum of a variable have to be lifted to the top-level of the program. In addition, an `IO` computation does not allow non-deterministic computations because the external world cannot be duplicated. For these reasons, we introduce a new monad `C` with the usual monadic operations and a function that lifts the result of a computation into the the `IO`-monad.

```
return_C :: a -> C a
(>>=_C) :: C a -> (a -> C b) -> C b lift_C :: C a -> IO a
```

Now, `minimum` and `maximum` are functions in this monad.

```
minimum :: Float -> C Float maximum :: Float -> C Float
```

The minimum schedule for our task scheduling example, assuming task A begin at time 0, can now be computed as follows.

```
minSchedule =
  schedule (a,b,c,d,e,f,end) & a =:= 0 >>= \_ ->
  minimum end >>=_C \minEnd -> end =:= minEnd \_ ->
  minimum f >>=_C \minF -> f =:= minF \_ ->
  minimum e >>=_C \minE -> e =:= minE \_ ->
  minimum d >>=_C \minD -> d =:= minD \_ ->
  minimum c >>=_C \minC -> c =:= minC \_ ->
  minimum b >>=_C \minB -> b =:= minB \_ ->
  return (0.0,minB,minC,minD,minE,minF,minEnd)
  where a,b,c,d,e,f,end free
```

Finally, we modify the functions `(<)`, `(>)`, `(<=)`, and `(>=)` such that they can handle linear expressions, too. Similar to `(==)`, these functions ask the constraint store whether the linear constraint imposed by the arguments or its negation is entailed by the current set of constraints and reduce to `True` or `False`, resp. If neither the constraint nor its negation is entailed by the constraint store, the evaluation is delayed until enough constraints have been added to prove the entailment relation.

5 Related work

Constraints over real numbers have been implemented in a logic programming language for the first time in Prolog III [Col87] and CLP($\Re$) [JMSY90]. Many other Prolog implementations since then include a constraint solver for linear constraints over real numbers or for other constraint domains.

Constraint solving has been integrated into functional-logic languages. For instance, Oz's evaluation model [Smo95] is based on concurrent constraint solving but does not include constraints over real numbers. The TOY system[3] includes disequality constraints between syntactic terms and linear constraints over real numbers [HLSU97] and uses a lazy narrowing strategy. It does not implement residuation, however. In addition TOY does not provide `minimize` and `maximize` functions.

6 Conclusion

In this paper we have sketched an integration of linear constraints over real numbers into the functional logic language Curry. The integration is straight forward concerning basic inequality predicates. However, the implicit state manifest in the constraint solver must be made explicit when we introduce functions which minimize or maximize variables with respect to the constraint store. In order to give a sound semantics, we have introduced a constraint solver monad `C` and define the `minimize` and `maximize` as operations in this monad.

References

[AEH97] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. ICLP '97*, pages 138–152. MIT Press, 1997.

[ALN87] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. ILPS '87*, pages 17–23, 1987.

[Col87] A. Colmerauer. Opening the Prolog III universe. *BYTE*, 12(9):177–182, August 1987.

[Han00] Curry: An integrated functional logic language, (version 0.7). `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 2000.

[3] `http://mozart.sip.ucm.es/incoming/toy.html`

[HLSU97] T. Hortalá-González, F.J. López-Fraguas, J. Sánchez-Hernández, and E. Ullán-Hernández. Declarative programming with real constraints. Technical Report SIP 5997, Univ. Complutense Madrid, 1997.

[HPW92] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell (version 1.2). *SIGPLAN Notices*, 27(5), 1992.

[HS98] M. Hanus and F. Steiner. Controlling search in functional logic programs. In *Proc. PLILP'98*, 1998.

[JMSY90] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP($\Re$) language and system. Technical report, IBM, 1990.

[PW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th POPL'93*, pages 123–137, 1993.

[Red85] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. ILPS'85*, pages 138–151, 1985.

[Smo95] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*. Springer LNCS 1000, 1995.