UPV-Curry: An Incremental Curry Interpreter with Polymorphic Types and Monadic I/O*

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

May 26, 1999

Abstract

Functional logic programming integrates the most interesting features of modern functional and logic languages. The multi-paradigm declarative language Curry is an extension of Haskell which is intended to become a standard in the area. In this paper, we present UPV-Curry, an efficient and rather complete implementation of Curry. We compare UPV-Curry with already existing implementations and provide experimental evidence of the fact that the techniques used in implementing UPV-Curry, which exploit incrementality both in the basic operational machinery and in the data structures used to guide the computations, actually speed up the Curry execution mechanism.

Keywords: Curry, Functional logic languages, Implementation, Needed narrowing.

1 Introduction

Functional logic languages combine the best features of the most important declarative programming paradigms, namely functional and logic programming (see [Han94] for a survey). Throughout this decade, many practical proposals have been made to amalgamate functional and logic programming languages. However, these languages have not succeeded in becoming widely used by the functional or logic programming communities. The multi-paradigm language Curry [HKMN95, HAK+99] is an extension of Haskell [HPW92] which is supported by an international initiative with the aim of becoming a standard in the area. Curry incorporates almost all the desirable features of an integrated language: types, modules, higher-order facilities, and declarative input/output from the functional side; and non-determinism, partly-instantiated calls, logical variables, search for solutions, and a logical view of concurrency from the logic side.

In order to ease and extend the use of Curry, it is essential to make efficient and practical implementations available. Program transformation techniques which aim to derive better semantically equivalent programs have recently been investigated for functional logic languages [AFV98a, AFV98b, AFMV97] and adapted to their use in Curry [AHLV99]. In this paper, we consider a different approach for improving the execution of Curry programs which relies on an incremental definition of the Curry operational machinery. We present UPV-Curry, a novel implementation of Curry which provides an almost complete implementation of the language. In Curry, each evaluation step for an expression e is performed on a *needed* redex of e, i.e., a subexpression of e which is an instance of a left-hand side l of a

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

(possibly conditional) program equation $l \mid c = r$ and which is really necessary for a complete evaluation of e. As is usual in functional logic languages, it is allowed to adequately instantiate variable occurrences of e in order to reduce this redex. The overall process is called *needed narrowing* [AEH94] and is implemented by using *definitional trees* [Ant92]. We develop a novel representation of definitional trees which is close to more traditional, well-known data structures such as Huet and Lévy's matching dags [HL91] or Peyton-Jones matching trees [Pey87], which have been previously used in pure functional programming. The new representation not only saves memory but is also better for defining the variable instantiations which make the needed steps possible. Moreover, it also makes the pattern matching between program equations and goal expressions faster. We also implement an *incremental* optimization of the basic evaluation strategy of Curry which is able to *locally* resume the search for a new redex after performing each single evaluation step. We show that all these modifications are actually (and independently) effective by comparing a basic non-optimized version of UPV-Curry (without the proposed refinements) and three optimized versions. Finally, we compare our implementation with TasteCurry, the most popular (and, the only public one currently available) implementation of Curry.

In Section 2, we give some preliminares. Section 3 briefly describes UPV-Curry. Section 4 describes definitional trees and our optimizations. Section 5 gives some experimental results. Section 6 concludes.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [DJ90] and functional logic programming [Han94]. We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. The set of *terms* and *constructor terms* with *variables* (e.g., x, y, z) from $\mathcal{X}$ are denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term is *linear* if it does not contain multiple occurrences of one variable. $root(t)$ denotes the symbol at the root of the term t. If $root(t) \in \mathcal{F}$ ($root(t) \in \mathcal{C}$), the term t is said *operation-rooted* (*constructor-rooted*). A *pattern* is a linear term of the form $f(\overline{d_n})$ where $f \in \mathcal{F}$ is n-ary and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $\mathcal{P}os(t)$ denotes the set of positions of t. Positions are ordered by the *prefix* ordering: $p \leq q$, if there exists p' such that $p.p' = q$. $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing* $t|_p$ by the term s (see [DJ90] for details). $\mathcal{FP}os(t)$ is the set of positions of operation-rooted subterms of t.

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

A set of rewrite rules $l \to r$ such that $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists $p \in \mathcal{P}os(t)$, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is called a *redex*. To evaluate terms containing variables, *narrowing* non-deterministically instantiates the variables such that a rewrite step is possible. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_1 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_1 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_{n-1}} t_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_1$.

3 Curry and UPV-Curry

Curry is a functional logic programming language that combines the best ideas of declarative

languages such as Haskell [HPW92] and SML [MTH90] (functional languages), Gödel [HL94] and λProlog [NM88] (logic languages), and ALF [HS91], Babel [KA96] and $\mathcal{TOY}$ [LS99] (functional logic languages). More specifically, Curry includes higher-order features, a type system, a module system, modern evaluation strategies, non-determinism, (encapsulated) search, partial data structures, existential variables, constraints, and declarative I/O.

3.1 Basic Curry Syntax

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

Example 3.1 *The following* Curry *program:*

```
data Nat = Z | S Nat

plus :: Nat -> Nat
plus Z     x = x
plus (S x) y = S (plus x y)
```

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

User defined data types are collections of *values* which are terms built from the data constructors associated to the considered data type. For instance, `Z` and `S` are the data constructors for the data type `Nat` and `Z`, `(S Z)`, `(S (S Z))` are examples of values of `Nat`.

A Curry program mainly consists of a collection of data type declarations and function definitions given by (conditional) equations of the form:

$$f\ d_1\ \cdots\ d_n\ \{\texttt{|}\ c\}\ \texttt{=}\ r$$

where f is the function symbol which the equation (partially) defines, and $d_1, \ldots, d_n$ are constructor terms with variables, c is an (optional) condition, and r is the result expression. The condition c can be either a boolean expression (in the usual sense of functional languages) or a constraint expression (i.e., a conjunction of equational constraints e_1`=:=`e_2 where both e_1 and e_2 are expressions). An expression has the form:

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

Example 3.2 *Some correct expressions and their evaluations*[1] *are:*

```
                    1 + 1 ⇝ 2
                100 * 100 ⇝ 10000
          True && False ⇝ False
              x || True ⇝ {x ↦ True}‖True, {x ↦ False}‖True
    chr ((ord 'a') + 1) ⇝ 'b'
         [1,2] ++ [3,4] ⇝ [1,2,3,4]
['a',x,'c'] =:= 'a':'b':y ⇝ {x ↦ 'b', y ↦ ['c']}
```

[1] Curry's answer expressions are represented as "$\sigma \| e$" consisting of an answer substitution σ and a result expression e. We write "e" instead of "$id \| e$", and "σ" if the expression e is empty.

3.2 UPV-Curry

UPV-Curry is a novel implementation of Curry which has been developed by the ELP group at the Technical University of Valencia (UPV), and is publicly available from the following URL: `http://www.dsic.upv.es/users/elp/soft.html`. For a complete description of UPV-Curry, see [EAL98].The system is written in SICStus Prolog v3.6 and provides an almost complete implementation of Curry according to [HAK+99]. The main features supported by the implementation are summarized below.

Polymorhic types. Curry is a strongly typed language, i.e., each expression has a unique type. It uses the Hindley-Milner polymorphic type system. UPV-Curry supports these features and it also incorporates a type inference mechanism which permits avoiding explicit function type declarations. Data type declarations define the type of constructors, but the types of functions can be omitted, being obtained by the type inference mechanism.

Standard types such as `Bool`, `Integer`, `Char`, and `String` and their associated operators are predefined in UPV-Curry. The polymorphic types for building lists and tuples, are also predefined.

Higher order. Curry treates the functions as *first class citizens*, i.e., they can be passed as arguments to other functions and returned as the value of an expression. Such higher order features are exemplified in the following.

Example 3.3 *Given the following function definition:*

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

the expression `map not [True,False]`*, specifies the application of the boolean (predefined) function* `not` *to* each *element of the list* `[True,False]` *thus yielding the list* `[False,True]`.

Example 3.4 *Partial function applications like* `(1+)` *are also allowed. The expression* `(1+)` *denotes a function which adds one to a given integer argument.*

Curry constraint solver. An *equational constraint* e_1`=:=`e_2 is satisfied if both e_1 and e_2 are reducible to unifiable data terms. Curry includes an equational constraint solver which is extended with the operator `&`, and which defines the concurrent conjunction of equational constraints.

Example 3.5 *(Continuing Example 3.1) The constraint:*

```
plus x y =:= S Z & x =:= Z
```

is satisfied by the substitution $\{$`x` $\mapsto$ `Z`, `y` $\mapsto$ `S Z`$\}$.

Monadic I/O. Curry provides a declarative I/O model based on *monads* which is also supported by UPV-Curry. The main idea is to represent actions as interactions with the 'outside world' by using a polymorphic type `IO t`. A value of type `IO t` is intended to represent an action which modifies the outside world with the possibility of taking a value of type `t` from outside. There are special operators to represent the sequential behaviour of the outside world. In this way, the monadic approach permits enclosing a typical imperative feature as I/O transactions into the declarative type system of Curry.

Example 3.6 *The following function symbols define some actions on the outside world:*

```
getChar   :: IO Char
getLine   :: IO String
putStrLn  :: String -> IO ()
>>        :: IO a -> IO b -> IO b
```

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

A simple menu can be defined by the following equation:

```
menu = putStr "Options [a,b,c]: " >> getLine >>=
       \ x -> if x=="a" then putStrLn "Option a" else
              if x=="b" then putStrLn "Option b" else
              if x=="c" then putStrLn "Option c" else
              putStrLn "Wrong option" >> menu
```

User interface. The working environment of UPV-Curry provides the following facilities:

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

4 Definitional trees

The definition of computations with Curry programs uses definitional trees. Given a program $\mathcal{R}$, a *definitional tree* $\mathcal{T}$ with pattern π is an expression of the form[2] [Ant92]:

$\mathcal{T}$ = $rule(\pi = r')$ where $\pi = r'$ is a variant of a program equation $l = r \in \mathcal{R}$.

$\mathcal{T}$ = $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ where o is a variable position of π (called the inductive position), $c_1, \ldots, c_n$ are different constructors for $n > 0$, and each $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of c_i and $x_1, \ldots, x_k$ are new variables.

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

Example 4.1 *Given the following function definition:*

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

The associated definitional tree is:

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

Figure 1 shows the graphical representation of this definitional tree. Each branch node shows the associated pattern and its inductive position, which is enclosed by a box.

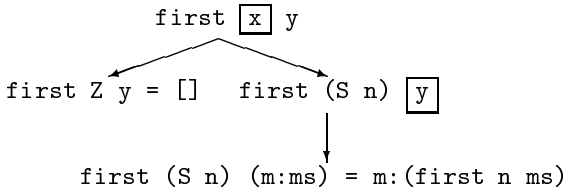

Figure 1: Definitional tree for the function `first`

Definitional trees are used for describing Curry computations as follows. A function λ from terms and definitional trees to sets of triples (p, R, σ) determines each evaluation step. Here, p is a position in t, R is the rule to be applied, and σ is a substitution which should be applied to t before reducing it. Given an operation-rooted term t and a definitional tree $\mathcal{T}$ with $pattern(\mathcal{T}) = \pi$ such that $\pi \leq t$, then $\lambda(t, \mathcal{T})$ is defined as follows [Ant92]:

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

This definition ensures that, whenever $(p, l \to r, \sigma) \in \lambda(t, \mathcal{T})$, $t \leadsto_{(p, l \to r, \sigma)} s$ is a valid narrowing step which is called a *needed* narrowing step. Needed narrowing only evaluates subterms which are needed to compute a value and it is the best narrowing strategy for inductively sequential programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions [AEH94].

4.1 Incremental definitional trees

By means of the function λ, definitional trees guide the search for a redex (with possible variable instantiations). By looking at the definition of λ, it is immediate to see that the pattern π of a branch node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and the patterns of its sons are almost the same except for a flat term $c_i(x_1, \ldots, x_k)$ which is placed on the inductive position o of π at the root node of each $\mathcal{T}_i$ for $1 \leq i \leq n$. On the other hand, the decision about the next immediate definitional tree which is going to be visited, only depends on the constructor symbol c_i rather than on the entire pattern of the son.

We have developed an optimized representation of the definitional trees which moves the pattern matching information to the exact place where it is needed. Given a program $\mathcal{R}$, an *incremental definitional tree* $\mathcal{T}'$ with pattern π is an expression of the form:

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

The pattern of an incremental definitional tree, which is not explicitly included in its syntactic structure, can be easily obtained as follows:

$$pattern'(\mathcal{T}') = \begin{cases} \pi & \text{if } \mathcal{T}' = rule'(\pi = r); \\ pattern'(\mathcal{T}_1')[x]_o & \text{if } \mathcal{T}' = branch'(o, (c_1, \mathcal{T}_1'), \ldots, (c_n, \mathcal{T}_n')) \text{ and} \\ & x \notin \mathcal{V}ar(pattern'(\mathcal{T}_1')). \end{cases}$$

[2] We ignore the *exempt* nodes which are in the original definition of [Ant92]. For simplicity, we also ignore the *or* nodes used in Curry in the development of our technical results, though they have been considered for implementing the interpreter.

Standard and incremental definitional trees are related by means of a function ρ which maps standard definitional trees to incremental definitional trees:

$$\rho(\mathcal{T}) = \begin{cases} rule'(l = r) & \text{if } \mathcal{T} = rule(l = r); \\ branch'(o, (c_1, \rho(\mathcal{T}_1)), \ldots, (c_n, \rho(\mathcal{T}_n))) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n), \text{ and} \\ & pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_k})]_o, \text{ for } 1 \leq i \leq n \end{cases}$$

Example 4.2 *The definitional tree of Example 4.1 is represented as follows:*

$$branch'(1,\ (\texttt{Z}, rule'(\texttt{first Z y = []})),\ (\texttt{S}, branch'(2,\ (\texttt{:}, rule'(\texttt{first (S n) (m:ms) = m:(first n ms)})))))$$

Figure 2 shows the corresponding graphical representation. As differences w.r.t. Figure 1, note that: a) patterns have been removed from branch nodes, b) a dashed box encloses the inductive position which labels branch nodes, and c) we label arcs with the corresponding constructor.

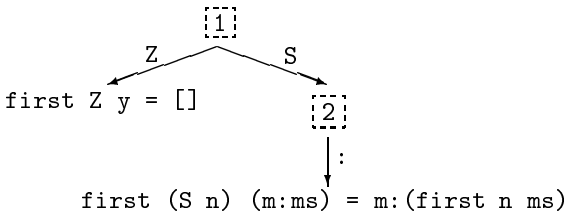

Figure 2: Incremental definitional tree for the function `first`

It has recently been proved that inductively sequential programs and Huet and Lévy's strongly sequential CB-programs coincide [HLM98]. Our incremental definitional trees are comparable to the original implementation of the data structures developed for strongly sequential TRSs, namely *matching dags* [HL91]. However, in contrast to Huet and Lévy's work, which only covers rewriting computations, we use such a structure not only for easing pattern matching, but also for supporting the instantiation of variables in the goal term as required for the narrowing steps.

4.2 Incremental evaluation

Since each step $e_i \leadsto_{(p_i, R_i, \sigma_i)} e_{i+1}$ in a needed narrowing sequence $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ is scheduled by $\lambda(e_i, \mathcal{T}_i)$ (where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$), and it usually changes only small parts of e_i, it is obvious that, if $p_i \in \mathcal{P}os(e_i) - \{\Lambda\}$, then the needed narrowing step for e_{i+1} (i.e., the computation of $\lambda(e_{i+1}, \mathcal{T}_{i+1})$) might *wastefully* examine some of the same symbols than in the preceding step and take almost the same decisions. Hence, we look for an *incremental* definition of λ which is able to locally resume the search from $e_{i+1}|_{q_i}$ for some $q_i \leq p_i$. Intuitively, whenever $(p, l \to r, \sigma) \in \lambda(t, \mathcal{T})$ and $t \leadsto_{(p, l \to r, \sigma)} s$, we can ensure that the symbols occurring at the position above p have not been modified, and the term $s|_p$ may or may not be operation-rooted. If $root(s|_p) \in \mathcal{F}$, then $s|_p$ should be further narrowed until a constructor-rooted term or a variable is obtained. If $root(s|_p) \in \mathcal{C} \cup \mathcal{X}$, then the next needed narrowing step is determined by the definitional tree $\mathcal{T}$ of $root(s|_q)$, where $q < p$ is the position of the defined symbol occurring at the position immediately above p.

Proposition 4.3 *Let t be an operation-rooted term, such that $t \leadsto_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$ and $q = max(\{q' \in \mathcal{FP}os(s) \mid q' \leq p\})$. If $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq q$.*

Proof. Let us first consider the case when $q \neq p$. Then, the computation of $\lambda(t, \mathcal{T})$ has reached a branch node $branch(o, (c_1, \mathcal{T}_1), \ldots, (c_n, \mathcal{T}_n))$ and there is an operation-rooted subterm $t|_{q.o}$ in the inductive position o. Thus the computation of $\lambda(s, \mathcal{T}')$ proceeds by

checking whether $s|_{q.o}$ is a constructor-rooted term or a variable, because if it were an operation-rooted term then $q = p$, which contradicts the original assumption.

If $s|_{q.o}$ is a constructor-rooted term or a variable, then the computation of $\lambda(s, \mathcal{T}')$ proceeds by selecting the corresponding subtree. Thus, the computed occurrence will be below q if a branch subtree with an operation-rooted term in the inductive occurrence is reached, or q if a rule subtree is reached.

Now consider the case when $q = p$. Then $s|_p$ is operation-rooted and this position needs to be further narrowed to compute a constructor-rooted term. Then, the computed occurrence is trivially q or below q. □

Corollary 4.4 *Let t be an operation-rooted term, such that $t \leadsto_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$. If $s|_p$ is operation-rooted and $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq p$.*

Since each step $e_i \leadsto_{(p_i,R_i,\sigma_i)} e_{i+1}$ in a needed narrowing sequence $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ is performed using $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$, where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$, Corollary 4.4 ensures that, whenever $e_{i+1}|_{p_i}$ is operation-rooted, then $p_{i+1} \geq p_i$. Thus, we take $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (p_i.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_{p_i}, \mathcal{T}')$ and $\mathcal{T}'$ is a definitional tree for $root(e_{i+1}|_{p_i})$. If $e_{i+1}|_{p_i}$ is constructor-rooted or a variable, then it is necessary to resume the evaluation somewhere above p_i. Proposition 4.3 allows us to come back to $e_{i+1}|_q$ by taking the ('most defined') definitional subtree $\mathcal{T}_i'$ of the definitional tree of $root(e_i|_q)$ which has been used in the computation of $\lambda(e_i, \mathcal{T}_i)$ immediately before obtaining $\mathcal{T}_i$. Thus, we let $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (q.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_q, \mathcal{T}_i')$.

Thus, we define a data structure $\mathcal{L}$ which contains the information needed to perform the needed narrowing steps incrementally. Given a term t, we let $\mathcal{L}$ denote a list of pairs $(p, \mathcal{T}')$ where $p \in \mathcal{FP}os(t)$ and $\mathcal{T}'$ (such that $pattern'(\mathcal{T}') \leq t|_p$) is the incremental definitional subtree needed to evaluate $t|_p$ in a forthcoming computation. The *incremental needed narrowing strategy* is denoted by $\lambda'(t, \mathcal{L})$. It takes as arguments a term t and a suitable auxiliary list $\mathcal{L}$ which plays the role of the definitional tree $\mathcal{T}$ in the original strategy $\lambda(t, \mathcal{T})$. Given an operation-rooted term t and a list $\mathcal{L} = [(q, \mathcal{T}'), \ldots]$ with $pattern'(\mathcal{T}') \leq t|_q$, we compute $(p, R, \sigma, \mathcal{L}') \in \lambda'(t, \mathcal{L})$, where p is the position pointing to the subterm of t to be narrowed, R is the rule to be applied, σ is the computed substitution, and $\mathcal{L}'$ is a list containing the information which allows us to proceed with the subsequent evaluation steps incrementally. Formally,

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

The following proposition states that λ' is able to compute the same needed narrowing step than λ starting from a given term and an initial list of incremental information.

Proposition 4.5 *Let t be an operation-rooted term and $\mathcal{T}$ be a definitional tree such that $pattern(\mathcal{T}) \leq t$. Then, $(p, l{\to}r, \sigma) \in \lambda(t, \mathcal{T})$ if and only if $(p, l{\to}r, \sigma, \mathcal{L}) \in \lambda'(t, [(\Lambda, \rho(\mathcal{T}))])$.*

Proof. Immediately from the definition of λ and λ'. □

Now, we have to ensure that the defined function symbols occurring at positions which are above (or at) a previously narrowed position are obtained by the strategy λ' and thus considered for the subsequent narrowing steps.

Proposition 4.6 (Completeness) *Let t be an operation-rooted term and $\mathcal{T}$ be a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $t \leadsto_{(p,l\to r,\sigma)} s$ using $(p, l\to r, \sigma) \in \lambda(t, \mathcal{T})$. Let $(p, l\to r, \sigma, \mathcal{L}) \in \lambda'(t, [(\Lambda, \rho(\mathcal{T}))])$, $F = \{q' \in \mathcal{FP}os(s) \mid q' \leq p\}$, and $q = max(F)$. Then, $q' \in F$ if and only if $(q', \mathcal{T}') \in \mathcal{L}$.*

Proof. We prove it by induction on q.

- If $q = \Lambda$, we consider two cases:
  - If $q = p$, then the rule case of λ' checks whether r is operation-rooted or not. Thus, if $F = \{\Lambda\}$, then $\mathcal{L} = [(\Lambda, \mathcal{T}')]$.
  - If $q < p$, then an operation-rooted term has been found at the inductive position of a branch subtree $\mathcal{T}'$ and r is not operation-rooted because if it were operation-rooted, then $q = p$, thus contradicting the original assumption. Therefore, if $F = \{\Lambda\}$, then $\mathcal{L} = [(\Lambda, \mathcal{T}')]$.
- If $q > \Lambda$, let q' be the defined symbol at the position immediately above q, F' be the set of positions which have an operation-rooted term occurring at position above q' (or equal to q' if $s|_p$ is operation-rooted), and $\mathcal{L}'$ be the list corresponding to the set F'. We consider two cases:
  - $q < p$. The computation of $\lambda'(t, [(\Lambda, \rho(\mathcal{T}))])$ recursively calls to $\lambda'(t, [(q, \mathcal{T}')])$ where the branch subtree $\mathcal{T}'$ corresponds to the operation-rooted term which appears at the inductive occurrence. From the definition of λ', an element $(q, \mathcal{T}')$ is added to $\mathcal{L}'$ to prepare the incremental evaluation of $t|_q$ in a forthcoming narrowing step.
  - $q = p$. Then the term r is operation-rooted and the rule case of λ' is entered, which includes an element $(p, \mathcal{T}')$ in $\mathcal{L}'$, where $\mathcal{T}'$ is a definitional tree for $root(r)$.

□

The following theorem shows that the needed narrowing steps performed by the incremental strategy λ' and the computation steps carried out by the original needed narrowing strategy λ are equivalent.

Theorem 4.7 (Correctness) *Let $\mathcal{R}$ be an inductively sequential program, $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$ be a needed narrowing derivation where $\mathcal{T}_i$ is a definitional tree such that $pattern(\mathcal{T}_i) \leq e_i$ for all $1 \leq i \leq n$, and $\mathcal{L}_0 = [(\Lambda, \rho(\mathcal{T}_1))]$. Then, for all $1 \leq i \leq n$, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$ if and only if $(p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda'(e_i, \mathcal{L}_{i-1})$.*

Proof. We prove it by induction on i.

- $i = 1$. This case is trivial, since by Proposition 4.5 it amounts to a call to $\lambda'(e, \mathcal{L}_0)$.
- $i > 1$. Consider the previously selected occurrence p_{i-1}, the corresponding expression e_{i-1}, and the list $\mathcal{L}_{i-1}$ of selected defined function symbols. Let $\mathcal{L}_{i-1} = [(q, \mathcal{T}) | \mathcal{L}']$.

  By Proposition 4.6, we know that q points to a defined symbol of e_i which is above or at the position p_{i-1}. By the ordering between the occurrences recorded in $\mathcal{L}_{i-1}$, q is the position of the defined symbol immediately above or at the position p_{i-1}.

  By Proposition 4.3, we know that $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}) \Leftrightarrow (q.p', R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$. Then by Proposition 4.5, $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}') \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}_i') \in \lambda'(e_i|_q, [(\Lambda, \mathcal{T})])$.

Benchmarks	Original	Incremental trees	Incremental evaluation	Both optimizations	Speed-up
`ackermann`	1856	1073	826	720	2.57
`fibonacci`	795	697	652	608	1.30
`append`	856	825	851	839	1.02
`last`	3348	3333	3352	3335	1.00
`horseman`	3071	3023	2937	2794	1.09
`assembler`	967	953	933	928	1.04
`append2`	2060	2028	2042	2033	1.01
`quicksort`	3272	2509	2236	2157	1.51
`mergesort`	492	421	359	347	1.41

Table 1: Comparison between UPV-Curry implementations (in ms.)

Now, note that it is immediate that $(q.p', R_i, \sigma_i, \mathcal{L}_i) \in \lambda'(e_i, [(q, \mathcal{T})|\mathcal{L}']) \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}'_i) \in \lambda'(e_i|_q, [(\wedge, \mathcal{T})])$ if $(q', \mathcal{T}'') \in \mathcal{L}'_i \Rightarrow (q.q', \mathcal{T}'') \in \mathcal{L}_i$.

Thus, we conclude that $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i) \Leftrightarrow (p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda'(e_i, [(q, \mathcal{T})|\mathcal{L}'])$.

□

5 Experimental results

In the experiments reported in this section, we use five Curry implementations. The first four are (slightly different) UPV-Curry versions, while the fifth one is the public TasteCurry environment. We first consider a basic UPV-Curry implementation which mechanizes the standard λ function, and the original definitional trees. The second UPV-Curry implementation includes the refinement due to the incremental definitional trees. The third UPV-Curry implementation includes the incremental strategy λ', but it considers the original definitional trees. We finally consider a completely refined implementation, including all optimizations.

We have measured the improvements of our optimizations by some benchmarks whose results are presented in Table 1 and Table 2. Table 1 summarizes the runtime differences for some examples using the four UPV-Curry implementations. Table 2 compares the runtimes of the benchmarks using the original UPV-Curry implementation, the complete refined UPV-Curry implementation, and the TasteCurry environment. Times were measured on a SUN SparcStation, running under UNIX System V Release 4.0. They are expressed in milliseconds and are the average of 10 executions. UPV-Curry implementations and TasteCurry are SICStus Prolog compiled programs running onto the SICStus Prolog interpreter. The benchmarks used for the analysis are given in Appendix. The functions `ackermann`, `fibonacci`, `append`, `last`, `quicksort` and `mergesort` are known. `horseman` is a Curry test example that computes, by means of constraints, the number of men and horses that have a certain number of heads and feet; `assembler` is another standard Curry test example that translates arbitrary jump-instructions into a certain machine code using logical variables to resolve forward jump addresses. Runtime input goals were chosen to give a reasonably long overall time, and are included in Table 2. Natural numbers are implemented using Z/S-terms, and lists are shown in goals using a subindex which represents its size. In the `assembler` benchmark goal, L represents the list: `[Label L0,Jump L1,Jump L0, Label L1,Label L2, Label L3,Jump L3, Label L5, Label L4, Jump L5, Jump L4, Jump L2]`. Similarly, L' in the goal for `quicksort` denotes the list `[10,9,8,7,6,5,4,3,2,1,0]` and L'' in the goal for `mergesort` denotes the list `[3,1,2,5,4,8,6,7,2,9,1,4,3,7]`.

Table 1 reveals that goals which require much computation, such as `ackermann`, `fibonacci`, `quicksort`, and `mergesort` get a substained speed-up when incrementality is used (e.g. get a 2.57 speed-up for the `ackermann` example). On the other hand, even in the cases when no significant speed-up is obtained (`append`, `last`, and `append2`), execution is not punished

Benchmarks	Goals	λ	λ'	TasteCurry
ackermann	ackermann 20	1856	720	2505
fibonacci	fibonacci 10	795	608	2262
append	append $[1]_{100}$ [1]	856	839	2956
last	last $[1]_{100}$	3348	3335	58736
horseman	horseman x y 8 20	3071	2794	3542
assembler	assembler L	967	928	748
append2	append x [] =:= $[1]_{100}$	2060	2033	57062
quicksort	qsort L'	3272	2157	1829
mergesort	sort (intMerge) L'' xs	492	347	870

Table 2: Runtimes of goals of the examples (in ms.)

Benchmarks	Incr./Std.
ackermann	57%
fibonacci	75%
append	66%
last	66%
horseman	66%
assembler	40%
append2	66%
quicksort	100%
mergesort	40%

Table 3: Definitional tree sizes

with any apreciable overhead. In other words, the use of incremental definitional trees causes no overload, which is justified by the fact that it is a simplification of the standard definitional trees. On the other hand, the use of incremental evaluation can lead to considerable speed-up in program execution while guaranteeing no appreciable slowdown.

A comparison between the UPV-Curry and the TasteCurry environment is more difficult since they use a different internal representation for Curry expressions. Nevertheless, the figures in Table 2 show that, in general, our Curry implementation has a better overall behaviour.

Another interesting aspect for the comparison concerns the spatial savings due to the use of incremental definitional trees. Table 3 shows the ratio between the number of symbols of inner nodes of standard definitional trees and incremental definitional trees. We only consider the information which is necessary for defining the needed narrowing strategy. The results of Table 3 clearly evidence that incremental definitional trees can also bring substantial savings in memory space (up to 60% in these tests).

6 Conclusions

We have presented UPV-Curry, a novel and effective implementation of Curry which provides an almost complete implementation of the language. In order to improve performance, we have introduced *incremental* definitional trees, a data structure which is closer to the traditional Huet and Lévy matching dags [HL91] than the original definitional trees and which also permits a more compact representation. Then we have adapted the Curry evaluation strategy in order to take the most of such an incremental representation of the definitional trees and we have shown that this representation can substantially improve the evaluation of goal expressions.

We have also implemented and proved correct an *incremental* (re)definition of the basic evaluation strategy of Curry which is able to *locally* resume the search for a new redex

after performing a previous incremental narrowing step, and which reduces the number of redexes to be considered in a needed narrowing derivation. Finally we have shown that these refinements can significantly speed-up the execution of Curry programs.

References

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[AFMV97] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Berlin, 1997.

[AFV98a] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[AFV98b] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of 1999 International Conference on Functional Programming, ICFP'99*, 1999. To appear.

[Ant92] S. Antoy. Definitional trees. In *editors Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[EAL98] S. Escobar, M. Alpuente, and S. Lucas. UPV-Curry User's Manual. Technical Report DSIC-II/38/98, DSIC, Universidad Politécnica de Valencia, 1998.

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

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

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

[HL91] G. Huet and J.-J. Lévy. Computations in Orthogonal Term Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honour of Alan Robinson*. The MIT Press, Cambridge, MA, 1991.

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

[HLM98] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

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

[HS91] M. Hanus and A. Schwab. The Implementation of the Functional-Logic Language ALF. FB Informatik, Univ. Dortmund, 1991.

[KA96] H. Kuchen and J. Anastasiadis. Higher Order Babel: Language and implementation. In *editors Proc. 5th International Workshop on Extensions of Logic Programming ELP'96*, pages 193–207. Springer LNAI 1050, 1996.

[LS99] F.J. López Fraguas and J. Sánchez Hernández. Toy: A multiparadigm Declarative System. In *10th International Conference on Rewriting Techniques and Applications, RTA'99*. Springer Verlag, Berlin, 1999. To appear.

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

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming ICLP'88*, pages 810–827. The MIT Press, Cambridge, MA, 1988.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

A Benchmarks code

ackermann

```
data Nat = Z | S Nat

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

fibonacci

```
data Nat = Z | S Nat

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

append

```
append [] y = y
append (x:r) y = (x:append r y)
```

last

```
append eval flex
append [] y = y
append (x:r) y = (x:append r y)
last l | append _ [_x] =:= l = _x
```

horseman

```
data Nat = Z | S Nat

int2nat   ::  Int -> Nat
int2nat n = if n<=Z then Z else S (int2nat (n-1))
add Z     n = n
add (S m) n = S (add m n)
horseman m h heads feet = heads =:= add m h & feet =:= add (add m m) (add (add h h) (add h h))
```

assembler

```
data LabelId = L0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9
data Instruction = Jump LabelId | Label LabelId

assembler ::  [Instruction] -> [(LabelId,Int)] -> Int -> [Int]
assembler [] _ _ = []
assembler ((Jump l):ins) st a | lookup l st _label _st1 = 9:_label:assembler ins _st1 (a+2)
assembler ((Label l):ins) st a | _st1 =:= insert l a st = assembler ins _st1 a
insert l a [] = [(l,a)]
insert l a ((l1,a1):st) | (l==l1)=:=True & a=:=a1 = (l1,a1):st
                        | (l==l1)=:=False = (l1,a1):(insert l a st)
lookup l [] a st1 = st1=:=[(l,a)]
lookup l ((l1,a1):st) a st1 = if l==l1 then a=:=a1 & st1=:=(l1,a1):st
        else let st2 free in lookup l st a st2 & st1=:=(l1,a1):st2
```

quicksort

```
-- quicksort using higher-order functions:
qsort ::  [Int] -> [Int]
qsort [] = []
qsort (x:l) = qsort (filter (<x) l) ++ x :  qsort (filter (>=x) l)
```

mergesort

```
-- merge sort:  sorting two list by merging the sorted first
-- and second half if the list
firsthalf xs = take (length xs `div` 2) xs
secondhalf xs = drop (length xs `div` 2) xs

sort ::  ([a] -> [a] -> [a] -> Constraint) -> [a] -> [a] -> Constraint

sort merge xs ys =
   if length xs < 2 then ys =:= xs
   else sort merge (firsthalf xs) _us
        & sort merge (secondhalf xs) _vs
        & (merge _us _vs ys)

intMerge ::  [Int] -> [Int] -> [Int] -> Constraint

intMerge [] ys zs = zs =:= ys
intMerge (x:xs) [] zs = zs =:= x:xs
intMerge (x:xs) (y:ys) zs =
   if (x > y) then intMerge (x:xs) ys _us & zs =:= y:_us
              else intMerge xs (y:ys) _vs & zs =:= x:_vs
```