The Pull-Tab Transformation

Abdulla Alqaddoumi[1] Sergio Antoy[2] Sebastian Fischer[3] Fabian Reck[3]

[1] Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, U.S.A.

[2] Computer Science Department
Portland State University
Portland, OR 97207, U.S.A.

[3] Institut für Informatik
Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany

Abstract. We present a new approach to the execution of functional logic programs. Our approach relies on definitional trees for the deterministic portions of a computation and on a graph transformation, called *pull-tab*, for the non-deterministic portions. This transformation moves, one level at a time, non-deterministic choices towards the root of the graph representing the state of a computation. With respect to need-based strategies for functional logic computations, our approach executes only localized graph replacements, a property that characterizes it as "pay as you go" and makes it suitable for parallel execution.

1 Introduction & Motivation

Non-deterministic programs are simpler to design and easier to reason about than their deterministic counterparts [4]. These advantages do not come for free. The burden unloaded from the programmer is placed on the execution mechanism. Loosely speaking, all the alternatives of a non-deterministic choice must be explored to some degree to ensure that no result of a computation is lost. Doing this efficiently is a long-standing problem.

There are three main approaches to the execution of non-deterministic steps in functional logic programs. This paper proposes a fourth approach with some interesting characteristics missing from the other approaches. We begin by proposing a simple example to present the existing approaches, to understand their limitations, and to compare their differences. Below, is a short program that we use as a running example. The syntax is Curry [10].

```
flip 0 = 1
flip 1 = 0                                    (1)
coin = 0 ? 1
```

We want to evaluate the expression

```
(flip x, flip x) where x = coin                         (2)
```

We recall that '`?`' is a library function, called *choice*, that returns either of its arguments, i.e., it is defined by the rules:

```
x ? _ = x
_ ? y = y                                               (3)
```

and that the `where` clause introduces a shared expression. Every occurrence of `x` in (2) has the same value throughout the entire computation according to the *call-time choice* semantics [13]. By contrast in `(flip coin, flip coin)` each occurrence of `coin` is evaluated independently of the other. Fig. 1 highlights the difference between these two expressions when they are represented as graphs.

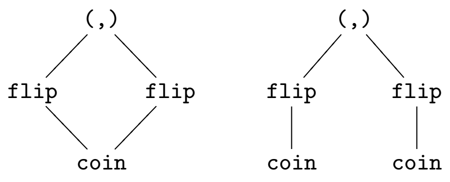

Fig. 1. Graph representations of (2) and `(flip coin, flip coin)`.

We recall that a *context* is an expression with a distinguished symbol called *hole* denoted '[ ]'. If C is a context, $C[x]$ is the expression obtained by replacing the hole in C with x. E.g., the expression in (2) can be written as t[`coin`], where t is the context of `coin`. An expression rooted by a node labeled by the choice symbol is referred to as *a choice*.

1.1 Previous approaches

Backtracking is the most traditional approach to non-deterministic computations in functional logic programming. Evaluating a choice in some context, say $C[u?v]$, consists in selecting either argument of the choice, e.g., u (the criterion for selecting the argument is not relevant to our discussion), replacing the choice with the selected argument, which gives $C[u]$, and continuing the computation. In typical interpreters, if and when the computation of $C[u]$ completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or compute $C[v]$. Referring to our running example, t[`0?1`] results in the evaluation of t[`0`] followed by the evaluation of t[`1`]. Backtracking is well-understood and relatively simple to implement. It is employed in successful languages such as Prolog [14] and in language implementations such as PAKCS [11] and $\mathcal{TOY}$ [8]. The major objection to backtracking is its incompleteness. If the computation of $C[u]$ does not terminate, no result of $C[v]$ is ever obtained.

Copying (or *cloning*) fixes the inherent incompleteness of backtracking. Evaluating a choice in some context, say $C[u?v]$, consists in evaluating simultaneously (e.g., by interleaving steps) and independently $C[u]$ and $C[v]$. In typical interpreters, if and when the computation of either completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or continue with the computation of the other. Referring to our running example, $t[\texttt{0?1}]$ results in the simultaneous and independent evaluations of $t[\texttt{0}]$ and $t[\texttt{1}]$. Copying is simpler than backtracking and it is used in some experimental implementations of functional logic languages [5, 18]. A significant optimization of copying consists in sharing (and thus computing only once) subexpressions of the context that are not on the spine of the choice (the path from the root to the choice). The major objection to copying is the significant investment of time and memory made when a non-deterministic step is executed. In well-designed programs, most alternatives of a choice fail to produce any result, hence portions of the copied context may never be used. For a contrived example, notice that in `1+(2+(...+(`n` 'div' coin)...))` an arbitrarily large context is copied when the choice is evaluated, but this context is almost immediately discarded.

Bubbling is an approach proposed to avoid the drawbacks of backtracking and copying [2, 15]. Bubbling is similar to copying, in that it copies a portion of the context of a choice to concurrently compute all its alternatives, but this portion of copied context is typically smaller than the entire context. We recall that in a rooted graph g, a node d is a *dominator* of a node n, proper when $d \neq n$, iff every path from the root of g to n contains d. An expression $C[u?v]$ can always be seen as $C_1[C_2[u?v]]$ in which the root of $C_2[\,]$ is a dominator of the choice. A trivial case arises when $C_1[\,] = [\,]$ and $C_2[\,] = C[\,]$. Evaluating a choice in some context, say $C[u?v]$, distinguishes whether or not C is empty. If C is the empty context, u and v are evaluated simultaneously and independently, as in copying, but there is no context to copy. Otherwise, the evaluation consists in finding C_1 and C_2 such that $C[u?v] = C_1[C_2[u?v]]$ and the root of C_2 is a proper dominator of the choice, and evaluating $C_1[C_2[u]?C_2[v]]$. If C_1 is the empty context, then bubbling is exactly as copying. Otherwise a smaller context, i.e., C_2 instead of C, is copied. Bubbling intends to reduce copying in the hope that some alternative of a choice will quickly fail. Referring to our running example, $t[\texttt{0?1}]$ bubbles to the expression represented in the left-hand side of Fig. 2. Observe that the node labeled `(,)` is the immediate proper dominator of the choice.

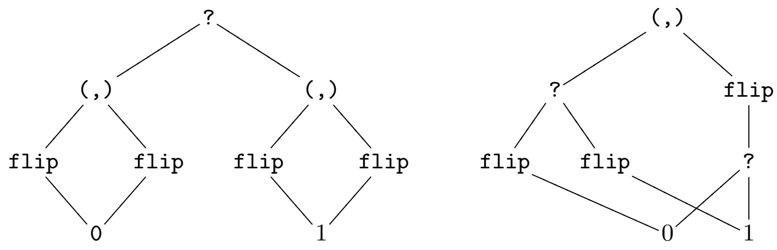

Fig. 2. Graph representation of the state of the computation of (2) after a bubbling (left side) and a pull-tab (right side) step.

Bubbling is more recent than the other approaches, it is not yet as well-understood, and it still is the subject of active investigation [7]. An objection to bubbling is the cost of finding a choice's immediate dominator and the risk of paying this cost repeatedly if no alternative of the choice fails. This cost entails traversing a possibly-large portion of the choice's context. Traversing the context is more efficient than copying it, since copying requires node construction in addition to the traversal, but it is still unappealing, since the cost of a non-deterministic step is not predictable and it may grow with the size of an expression.

2 Pulling the Tab

A *program* is a graph rewriting system [9, 16]. An *expression* is a rooted graph over the signature of the program. A *computation* is the repeated transformation of an expression by either a *rewrite* or a *pull-tab* step defined below. Rewrite steps are computed with standard techniques [1]. Informally, a pull-tab step moves a choice toward the root of an expression one level at a time. As in a rewrite, a (sub)expression of an expression is replaced. Textually, a (sub)expression of the form $f(t_1, \ldots, a_1?a_2, \ldots, t_k)$, where f is not a choice, is replaced by $f(t_1, \ldots, a_1, \ldots, t_k)?f(t_1, \ldots, a_2, \ldots, t_k)$. For example, $((0+2) ? (1+2)) * 3$ is the pull-tab of $(0 ? 1) + 2 * 3$. If and when a choice reaches the root of an expression, its alternatives have no context and are evaluated independently of each other. The metaphor behind the name is to look at a path from the root of an expression down to a choice as a zipper in which the choice is a pull tab. As a choice is pulled up, the path opens into two strands, like a zipper, below the pull tab. Pulling a choice above a predecessor copies the smallest amount of context, i.e., the predecessor node only.

Unfortunately, the pull-tab transformation as sketched above may be unsound. Fig. 3 shows a state of the computation of (2) after some rewrite and pull-tab steps. The superscript of some symbols may be ignored for the time being. Without a corrective action, four results would be produced. In particular, the right argument of the left choice, i.e., `(1,0)`, is not intended by the semantics of current functional logic languages such as Curry [10] and $\mathcal{TOY}$ [8].

Unsoundness occurs when some choice has two predecessors, as in our running example. The choice will be pulled up along two paths creating four strands that eventually must be combined together. Some combinations will contain mutually exclusive alternatives, or in other words subexpressions impossible to obtain with the call-time choice semantics. In our running example, one such combination mixes a 1 originating from the left alternative of the initial choice with a 0 from the right alternative of *the same choice*. Avoiding expressions with mutually exclusive alternatives suffices to recover the soundness of the pull-tab strategy.

To avoid impossible combinations of subexpressions, we track the history of the non-deterministic steps of each expression. This history has been used in other aspects of functional logic computations [3, 6] under the name of "fingerprint." A node in a graph is decorated with information such as labeling and

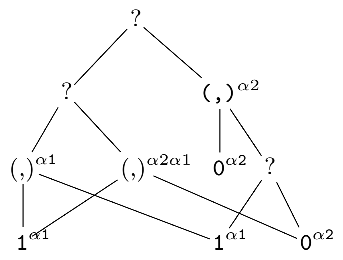

Fig. 3. States of the computation of (2) after both rewrite and pull-tab steps. Superscripts are fingerprints. α is the choice identifier of every node labeled by `?`. A node labeled by (,), the pair constructor, has fingerprint $\alpha 1 \alpha 2$. The subgraph at this node mixes the left and right arguments of a choice and consequently does not produce a result.

successor functions. For defining the pull-tab strategy, we extend the decorations of nodes. Let Ω be a denumerable set whose elements we call *choice identifiers* and denote by Greek letters. A *fingerprint* is a finite subset of $\Omega \times \{1, 2\}$ whose pairs we denote by juxtaposition. A node of each graph of a computation is decorated by a fingerprint and, if the node is labeled by the choice symbol, it is also decorated by a choice identifier.

A rewrite step preserves these decorations and assigns an empty fingerprint to any node introduced by the replacement and a fresh choice identifier if the node is a choice. A pull-tab step involves two nodes, a choice c and one of its predecessors p not labeled by a choice. Let α be the choice identifier of c and f the fingerprint of p. Informally, the step "moves up" c creating a new node c' and "splits" the predecessor p creating two new nodes, say p_1 and p_2. In the resulting expression, the choice identifier of c' is again α and the fingerprints of p_1 and p_2 are $f \cup \{\alpha 1\}$ and $f \cup \{\alpha 2\}$, respectively.

If the fingerprint of a node n contains $\alpha 1$ and $\alpha 2$, for some choice identifier α, the graph rooted by n is semantically impossible and should be eliminated. Fig. 3 shows an example of such a node, where superscripts denote fingerprints.

3 Current Work

We are developing a virtual machine based on the pull-tab strategy. The machine, about 1000 lines of commented Ruby [17] code, includes a rudimentary parser for the command line interpreter and a sophisticated printer for development purposes and the presentation of results. The machine executes multisteps [12] that, depending on the functional logic program being executed, may contain dozens or hundreds of elementary steps. Since both rewrite and pull-tab steps are localized graph replacements, we expect to be able to execute the elementary steps of a multistep in parallel with only a modest synchronization overhead.

References

1. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
2. S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006.
3. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 73–82, Lisbon, Portugal, September 2009.
4. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
5. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125, Lubeck, Germany, September 2005. Springer LNCS 3474.
6. Bernd Brassel and Frank Huch. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*, pages 122–138, Berlin, Heidelberg, 2007. Springer-Verlag.
7. D. Brown. Ph.D. dissertation, 2010. In progress.
8. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.
9. R. Echahed. Inductively sequential term-graph rewrite systems. In *Graph Transformations, 4th International Conference (ICGT 2008)*, pages 84–98, Leicester, UK, 2008. Springer, LNCS 5214.
10. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at `http://www.informatik.uni-kiel.de/~curry`.
11. M. Hanus, editor. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*, 2008. Available at `http://www.informatik.uni-kiel.de/~pakcs`.
12. 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*. MIT Press, Cambridge, MA, 1991.
13. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
14. ISO. Information technology - Programming languages - Prolog - Part 1, 1995. General Core. ISO/IEC 13211-1, 1995.
15. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: The HO case. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 147–162. Springer LNCS 4989, 2008.
16. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
17. D. Thomas and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., 2001.
18. A. Tolmach, S. Antoy, and M. Nita. Implementing functional logic languages using multiple threads and stores. In *Proc. of the 2004 International Conference on Functional Programming (ICFP)*, pages 90–102, Snowbird, Utah, USA, September 2004. ACM.