# A Slicing Tool for Lazy Functional Logic Programs*

Claudio Ochoa[1], Josep Silva[2], and Germán Vidal[2]

[1] DIA, Tech. University of Madrid, 28660 Boadilla del Monte, Spain
cochoa@fi.upm.es
[2] Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain
{jsilva,gvidal}@dsic.upv.es

**Abstract.** Program slicing is a well-known technique that has been widely used for debugging in the context of imperative programming. Debugging is a particularly difficult task within lazy declarative programming. In particular, there exist very few approaches to program slicing in this context. In this paper, we describe a slicing tool for first-order lazy functional logic languages. We also illustrate its usefulness by means of an example.

## 1 Introduction

Program slicing is a well-known technique to extract a program fragment w.r.t. some criterion. It was first proposed as a debugging tool [3] to allow a better understanding of the portion of code which revealed an error; nowadays, it has been successfully applied to a wide variety of software engineering tasks, such as program understanding, debugging, testing, specialization, etc. Unfortunately, there are very few approaches to program slicing in the context of *declarative* languages. Basically, a *program slice* consists of those program statements which are (potentially) related to the values computed at some program point and/or variable, referred to as a *slicing criterion*.

In this work, we describe a slicing tool for first-order lazy functional logic languages. Our tool is built on top of a tracer based on *redex trails* [1], which allows the presentation of computation traces in a way easier to understand for the programmer. A clear advantage of our approach [2] is that existing tracers can be extended with slicing capabilities with a modest implementation effort, since the same data structure—the redex trail—is used for both tracing and slicing. Furthermore, it can easily be extended to cope with other language features like built-in functions, higher-order combinators, etc., since all these features are already covered by state-of-the-art debuggers based on redex trails.

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

## 2 The Slicing Tool

In this section, we describe the structure of our slicing tool. It can be used both for debugging—by automatically extracting the program fragment which contains an error—and for program specialization—by generating executable slices w.r.t. a given slicing criterion. The technical details of this slicing technique can be found in [2].

**Tracer.** It is introduced in [1]. The tracer executes a program using an *instrumented* interpreter. As a side effect of the execution, the *redex trail* of the computation is stored in a file. The tracer is implemented in Haskell and accepts first-order lazy functional logic programs that can be traced either backwards or forwards. In our slicer, we only slightly extended the original tracer in order to also store in the redex trail the location (the so called *program position*), in the source program, of every reduced expression.

**Viewer.** Once the computation terminates (or it is aborted by the user in case of a looping computation), the viewer reads the file with the redex trail and allows the user to navigate through the entire computation. The viewer, also introduced in [1], is implemented in Curry, a conservative extension of Haskell with features from logic programming including logical variables and non-determinism. The viewer is useful in our approach to help the user to identify the slicing criterion.

**Slicer.** Given a redex trail and a slicing criterion, the slicer outputs a set of program positions that uniquely identify the associated program slice. The slicing tool is implemented in Curry too, and includes an editor that shows the original program and, when a slice is computed, it also highlights the expressions that belong to the computed slice.

**Specializer.** Similarly to the slicer, the specializer also computes a set of program positions—a slice—w.r.t. a slicing criterion. However, rather than using this information to highlight a fragment of the original program, it is used to extract an executable slice (possibly simplified) that can be seen as a specialization of the original program for the given slicing criterion.

More information on the slicing tool (including examples, benchmarks, source code) is publicly available at: `http://www.dsic.upv.es/~jsilva/slicer/`.

## 3 The Slicer in Practice

In order to show the usefulness of our slicer, this section presents a debugging session that combines tracing and slicing.

We consider the program shown in Fig. 1 (for the time being, the reader can safely ignore the distinction between gray and black text). In this program,

```
data T = Hits Int Int

main = printMax (minMaxHits webSiteHits)

webSiteHits = [0, 21, 23, 45, 16, 65, 17]

printMin t = case t of (Hits x _) -> show x
printMax t = case t of (Hits _ y) -> show y

fst t = case t of (Hits x _) -> x
snd t = case t of (Hits _ y) -> y

minMaxHits xs = case xs of
                    (y:ys) -> case ys of
                                  []     -> (Hits y y);
                                  (z:zs) -> let m = minMaxHits (z:zs)
                                            in (Hits (min y (fst m))
                                                     (max y (snd m)))

min x y = if (leq x y) then x else y
max x y = if (leq x y) then y else x

leq x y = if x==0 then False
                  else if y==0 then False else leq (x-1) (y-1)
```

**Fig. 1.** Example program `minMaxHits`

the function `main` returns the maximum number of hits of a given web page in a span of time. Function `main` simply calls `minMaxHits` which traverses a list containing the daily hits of a given page and returns a data structure with the minimum and maximum of such a list.

The execution of the program above should return 65, since this is the maximum number of hits in the given span of time. However, the code is faulty and prints 0 instead. We can trace this computation in order to find the source of the error. The tracer initially shows the following top-level trace:

```
0 = main
0 = printMax     (Hits _ 0)
0 = prettyPrint  0
0 = if_then_else True 0 0
0 = 0
```

Each row in the trace has the form $val = exp$, where $exp$ is an expression and $val$ is the computed value for this expression.

By inspecting this trace, it should be clear that the argument of `printMax` is erroneous, since it contains 0 as the maximum number of hits. Note that the minimum (represented by "`_`" in the trace) has not been computed due to the laziness of the considered language. Now, if the user selects the argument of `printMax`, the following subtrace is shown:

```
0          = printMax   (Hits _ 0)
(Hits _ 0) = minMaxHits (0:_)
(Hits _ 0) = Hits       _ 0
```

**Table 1.** Benchmark results

| benchmark | time | orig size | slice size | reduction (%) |
|---|---|---|---|---|
| minmax | 11 ms. | 1.035 bytes | 724 bytes | 69.95 |
| horseman | 19 ms. | 625 bytes | 246 bytes | 39.36 |
| lcc | 33 ms. | 784 bytes | 613 bytes | 78.19 |
| colormap | 3 ms. | 587 bytes | 219 bytes | 37.31 |
| family_con | 4 ms. | 1453 bytes | 262 bytes | 18.03 |
| family_nd | 2 ms. | 731 bytes | 289 bytes | 29.53 |
| **Average** | **12 ms.** | | | **43.73** |

From these subtraces, the user can easily conclude that the evaluation of function `minMaxHits` (rather than its definition) contains a bug since it returns `0` as the maximum of a list without evaluating the rest of the list.

At this point, the tracer cannot provide any further information about the location of the bug. This is where slicing comes into play: the programmer can use the slicer in order to isolate the slice which is responsible of the wrong result; in general, it would be much easier to locate the bug in the slice than in the complete source program. For instance, in this example, the slice would contain the black text in Fig. 1. Indeed, this slice contains a bug: the first occurrence of `False` in function `leq` (less than or equal to) should be `True`. Note that, even though the evaluation of `minMaxHits` was erroneous, its definition was correct.

## 4 Benchmarking the slicer

In order to measure the specialization capabilities of our tool, we conducted some experiments over a subset of the examples listed in

`http://www.informatik.uni-kiel.de/~curry/examples`.

Some of the benchmarks are purely functional programs (`horseman`, `family_nd`), some of them are purely logic (`colormap`, `family_con`), and the rest are functional logic programs.

Results are summarized in Table 1. For each benchmark, we show the time spent to slice it, the sizes of both the benchmark and its slice, and the percentage of source code reduction after slicing. As shown in the table, an average code reduction of more than 40% is reached.

## References


1. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of PPDP'04*, pages 179–190. ACM Press, 2004.
2. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of PEPM'04*, pages 123–134. ACM Press, 2004.
3. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, U. of Michigan, 1979.