Analyzing Path Profiles with the Hot Path Browser

Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

James R. Larus
Microsoft Research
Microsoft Corp.
larus@microsoft.com

Genevieve Rosay
Dept. of Computer Science
University of Wisconsin–Madison
rosay@cs.wisc.edu

1 Introduction

Measuring program behavior is easy; understanding program behavior is hard. It is not difficult to construct program profilers using instrumentation libraries such as ATOM [SE94], and EEL [LS95]. The resultant profiling tools can accurately and efficiently record many aspects of programs' execution. Such tools generate reams of data but offer little support to the end user in analyzing and understanding this data. Producing useful information that provides insight into a program's behavior remains a difficult task. Without this understanding, consumers of measurement data–programmers, computer architects, compiler writers, etc.–can be distracted by minor moguls and miss important mountains.

We have built a tool, called the Hot Path Browser (HPB) for graphically displaying path profiles [BL96, ABL97]. Paths provide a concise record of a program's dynamic control flow. Even large complex programs, such as gcc and Microsoft Word, only execute a few tens of thousands of paths in an execution. Moreover, the vast majority of these paths contribute little to the overall execution, which is dominated by a small subset of hot paths. HPB helps the user analyze the path profile data from one or more runs of a program. Its browser-like windowing interface provides mechanisms for isolating and displaying hot paths at the source level.

With a tool such as HPB, a programmer can examine the small number of heavily executed paths, to better understand his or her program's behavior and to find redundant computation. Similarly, a computer architect or compiler writer can use the tool to relate dynamic metrics, such as cache misses or instructions stalls along a path to higher-level constructs in a program.

1.1 Workshop Goals

The goals of our workshop presentation will be to demonstrate the HPB tool, describe our experience in using it for performance tuning, find potential users, and get feedback and ideas about how to improve the tool. Based on our experience with applying path profiles to optimization, we believe there is a valuable opportunity to integrate a tool such as HPB into a profile-directed compiler, so as to aid compiler writers in understanding and improving optimization. Longer term, the cost/benefit tradeoff of using profile-directed optimization is a difficult problem, which we hope that HPB can be used to address.

1.2 Software Distribution

We plan to release a beta version of HPB to friendly testers in mid-December. This release will support path profiles created by the PP tool, which will also be made available. Please visit the Hot Path Browser home page at

`http://www.bell-labs.com/projects/HPB/`

to find out more information about the tool's availability in the coming months.

2 A Quick Tour of the Hot Path Browser

HPB is written in Java and functions much like a web browser. HPB imports path profiles described in an extensible ASCII data format so that additional attributes may easily be associated with paths, procedures and files. Currently, profiles are created using the PP profiling tool of Ammons, Ball and Larus [BL96, ABL97]. PP instruments Solaris executable files to record intraprocedural path profiles. The paths PP records begin at

procedure entry or a loop backedge and end at procedure exit or a loop backedge.

Profiles may be directly imported into HPB or created within the browser by applying operations (such as union, difference, intersection, etc.) to existing profiles. HPB also supports "roll-up" operations over profiles, so that aggregate profiles at the procedure or file level may be used to filter the profile.

Projects are used to organize a set of profiles and store historical information about how the user has filtered the profile, as well as other information about the state of the data exploration task the user is undertaking.

Figure 1 shows a screen snapshot of main window of HPB, which is divided into four panes. The path profile shown was generated by applying the PP tool to a C++ program implementing red-black trees (procedure names are mangled because PP only has access to the executable's symbol table).

The upper left pane displays path profile data in a table, with one row per path. Paths have been sorted in descending order of frequency. Other path attributes include the path's ID (computed by the Ball/Larus algorithm), the procedure the path occurs in, the length of the path (number of basic blocks in the path) and the number of dynamic machine instructions executed along that path (which is simply the path's frequency multiplied by its length). The user may sort paths in ascending/descending/original order by any attribute by clicking on the appropriate column heading.

The other two tables (lower left and upper right) show aggregate information about the path profile at the procedure and file level (respectively). In this example, both tables show the procedure/file name, the total number of static paths (as reported by PP) in each procedure/file, the number of executed paths, as well as the number of executed instructions.

Finally, a source code view is provided in the lower right. This source view may be annotated in many ways, as we will see later. Numerical metrics (such as the number of cycles per line) may easily be displayed in the left margin of this view.

The four views comprising this page are linked together by the currently selected set of paths. In each view, the selected entities are shown in bold italic font. In this example, the most frequently executed path (of frequency 30,008) has been selected in the path table. The procedure and file views show the procedure and file containing that path, while the source view shows the selected path. When a selection is made, all views automatically scroll to the first selected entity. For example, when the above path is selected, the source code view scrolls to the first line contained in the path. This capability is very useful for quickly browsing the paths. The user may select sets of entities in any pane: selecting a procedure/file will select all the executed paths in that procedure/file; selecting a line in the source code view selects all paths that executed that line. We will discussion other ramifications of selection later on.

A standard feature of interactive data analysis tools is to support a "drill down" mechanism to focus attention on a subset of data and explore this in greater detail. HPB supports drill down using the "Select" button, which, when pressed, causes a new page to be created in which only the currently selected paths are displayed. This provides a useful mechanism for focusing attention on paths of interest. For example, a user might sort paths in descending order of frequency, select the top ten paths and press the "Select" button. The browser history maintains the previous pages so that one can return to a previous selection easily (using the "Back" button). If the user finds a set of paths of interest, a bookmark can be created, to make it easy to return to this state later. Each project has its own set of bookmarks.

2.1 Visualizing Path Overlap with the Annotated Source View

An important aspect of paths is that they can overlap one another in non-trivial ways (whereas basic blocks do not overlap). Two hot paths may overlap in such a way as to create an even hotter subpath. To address this issue, HPB provides an annotated source code view, as shown in Figure 2, that depicts path overlap in the left margin of the view.

In this example, only the paths from the procedure `RBDelete` are shown. In this page, there are three panes: the upper left is the list of paths that executed (10 of them); the lower left is the list of procedures (just one); the right pane is the annotated source view. In this view, each executed path is allocated a column in the left margin. For each executed line contained in a path, there is a large square in that path's column. For each line inbetween two lines in a path, a small dot is displayed, so the user can see whether there is more code left in a path without having to scroll unnecessarily.

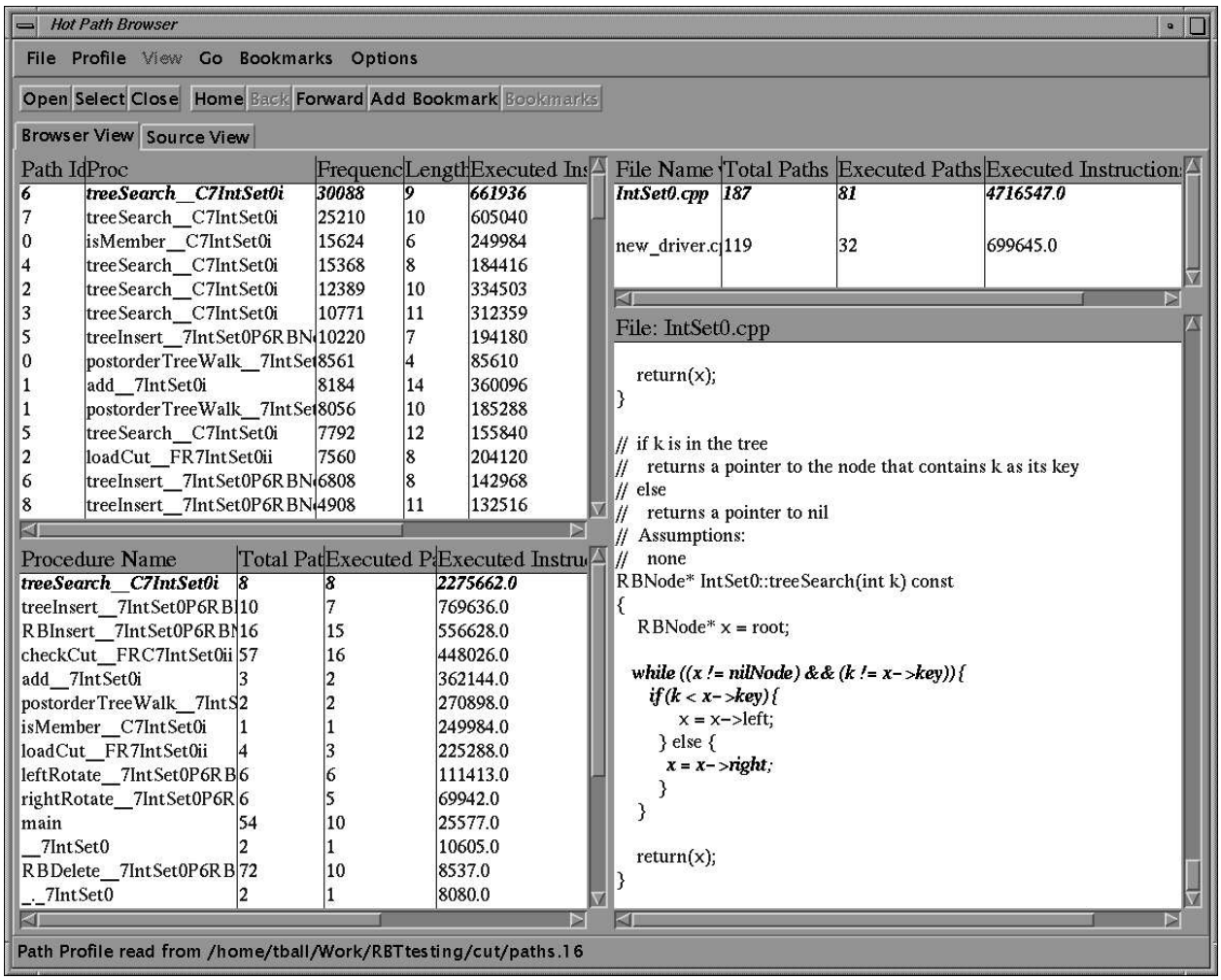

Figure 1: Screen snapshot of the Hot Path Browser. See text for explanation.

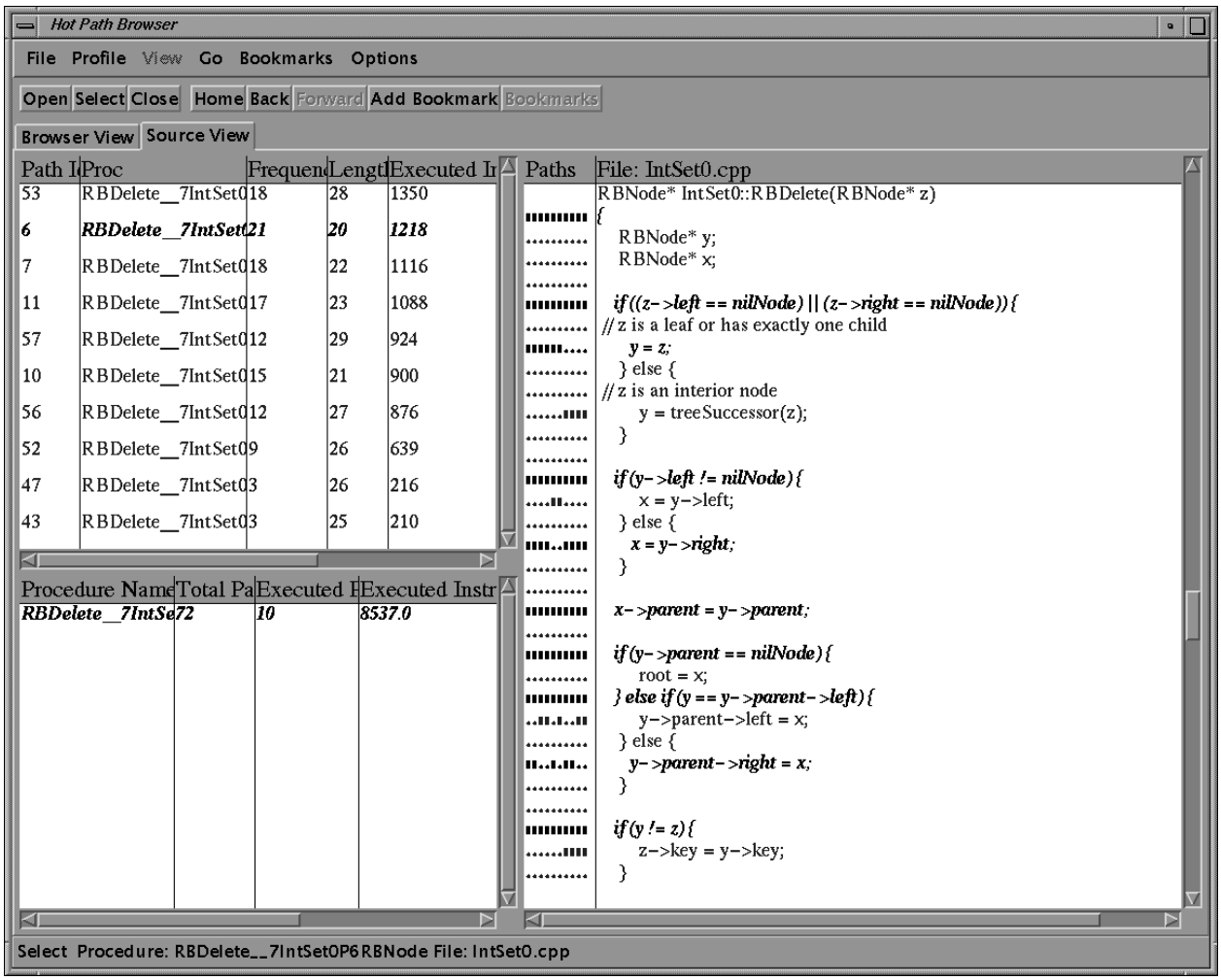

Figure 2: Screen snapshot of the annotated source view of HPB. See text for explanation.

2.2 Support for Analyzing Correlated Code

Correlated branches are branches whose behavior depends on one another. For example, a program may branch on the condition `(x < 5)` and then later branch on the condition `(x < 10)`. These branches are logically dependent. Correlated branches when a branch controls the setting of a variable's value that is later branched on. Correlated branches lead to infeasible paths: paths that can never be executed. They can also be an indication of redundant checks [BGS97].

Of course, determining whether or not branches are correlated is an undecidable problem. However, path profiles give an approximation to correlated branches, and can be used to point to potentially correlated branches and correlated code. Given a path profile P, statements t is correlated with statement s if for all paths p in P, the presence/absence of s in path p implies the presense/absence of t in p.

HPB currently supports investigation of correlated code in several ways. First, the annotated source view can be used to examine correlations between lines of code. For example, in Figure 2, we can see that there is no (dynamic) path which executes both the statements "`y = z`" and "`z->key = y->key`". Second, a user may select two more lines in the source view and then ask HPB to show only the paths containing all the selected lines.

2.3 HPB Performance and Scalability

HPB has been run on programs totally more than 100,000 lines of code. We expect HPB to scale very well to large program since the size of path profiles is small with respect to program size. Although there are potentially exponentially many paths that could execute, only a small number of paths evecute in any given run. For example, a program such as `gcc` executes on the order of tens of thousands of paths. We expect the data structures needed to support HPB to be memory-resident for most applications.

3 Tuning the Performance of a Red-Black Tree Implementation

We will illustrate HPB further by using it to analyze the characteristics of a small program, a C++ implementation of red-black trees. Red-black trees are binary search trees (the key value at any node N is greater than the key value at N's left child and less than or equal to the key value at N's right child). Various red-black tree invariants guarantee that the trees are balanced.

In our analysis, we focus on correlated branches and infeasible paths. Using the procedure view, we focused on procedures that executed far fewer paths than they contained. As shown in Figure 1, procedure `RBDelete`, which is used to remove a node from a red-black tree, has 72 paths, but only 10 of these paths executed. Figure 3(a) repeats the entire body of this procedure in a more readable format. The numbers in brackets on the right of lines indicates the number of paths from that line of code to the end of the procedure.

Using HPB, we investigated correlated branches in the 10 executed paths. Using HPB, we found the following dynamic correlations between lines of code in Figure 3(a):

- If line 4 executes then line 16 does not execute. This correlation is clearly a static correlation as line 4 aliases y to z and line 15 tests for the alias.
- If line 5 executes then line 8 executes and lines 11 and 16 do not execute. In this case, the correlations are all static, but a bit more difficult to figure out. If line 5 executes then z is an interior node of the tree, which means that `treeSuccessor` will always return a vertex y which is not the root of the tree, has a `nil` left child and is not equal to z.

We continued this exercise, examining dynamic correlations observed with HPB to determine if they were indeed static correlations. Based on the results of this analysis, we restructured the code of Figure 3(a) to that shown in Figure 3(b). The new code contains 15 paths, 12 of which are feasible. Each of the paths corresponds directly to a path in the original code. However, in the new method, each path contains fewer predicates and fewer instructions than its corresponding path in the original `RBDelete` method. The restructured method improved the performance of the overall implementation about 2 percent, as well as making the code simpler to understand (since there are only 3 infeasible paths, compared to 60 before).

4 Future Work

Future work will concentrate on three areas: adding support for interprocedural path profiles, hot subpath analysis, and graphical views of the control flow graph.

```
 1  RBNode* IntSet0::RBDelete(RBNode* z) {
 2    RBNode *y, *x;

 3  if ((z->left == nilNode) ||     [72]
        (z->right == nilNode))
 4    y = z;
    else
 5    y = treeSuccessor(z);

 6  if (y->left != nilNode)         [24]
 7    x = y->left;
    else
 8    x = y->right;

 9  x->parent = y->parent;

10  if (y->parent == nilNode)       [12]
11    root = x;
12  else if (y == y->parent->left)
13    y->parent->left = x;
    else
14    y->parent->right = x;

15  if (y != z)                      [4]
16    z->key = y->key;

17  if (y->colour == black)          [2]
18    RBDeleteFixup(x);

19  return(y);
}
```

(a)

```
RBNode* IntSet0::RBDelete(RBNode* z) {
  RBNode *y = z, *x;

  if (z->left == nilNode)         [15]
    x = z->right;
  else if (z->right == nilNode)
    x = z->left;
  else {
    y = treeSuccessor(z);
    x = y->right;
    z->key = y->key;
  }

  x->parent = y->parent;

  if (y->parent == nilNode)        [5]
    root = x;
  else {
    if (y == y->parent->left)      [4]
      y->parent->left = x;
    else
      y->parent->right = x;
    if (y->colour == black)        [2]
        RBDeleteFixup(x);
  }
  return(y);
}
```

(b)

Figure 3: Original and restructured implementations of RBDelete. The original implementation (left) has 72 paths, 12 of which are feasible. The restructured implementation (right) has 15 paths, 12 of which are feasible. The numbers in brackets to the right of a line indicate the number of paths from that line to the end of the procedure.

4.1 Interprocedural Path Profiles

Currently, the focus of HPB is on intra-procedural path profiles, such as produced by the PP tool, and intra-procedural performance tuning tasks such as that described in the red-black tree case study. For many programs, especially those written in object-oriented languages, inter-procedural path profiles will be needed in order to support more aggressive performance tuning across method/procedure boundaries. Tools supporting efficient inter-procedural path profiling are around the corner and we plan to investigate how to incorporate such path profiles into HPB soon.

4.2 Concept Analysis for Exploring Subpath Structure

As stated before, set of paths may overlap in such a way as to create an even hotter subpath. Based on our experience with HPB so far, we feel an analysis or view that focuses on hot subpaths will be a useful addition to HPB.

Before presenting hot subpaths to the user, we must determine how to identify these subpaths in an existing path profile. Concept analysis is a technique for identifying groups of objects that have common attributes [Sne96]. Concept analysis provides a useful data structure for reasoning about hot paths. The input to concept analysis is a boolean-valued table in which rows represent objects and columns represent attributes. An entry of the table is true if an object has an attribute and false otherwise. For path profiling, the objects are paths and the attributes are lines of code. The left-margin of the annotated source code view (Figure 2) is a visual representation of this table, where the columns are paths and the rows are lines.

In our domain, the pair (P,L), where P is a set of paths and L a set of lines, is a *concept* if every path in P contains all the lines in L, and no path outside of P contains all the lines in L. Equivalently, (P,L) is a concept if every line in L is contained in every path in P and there is no line outside of L contained in every path in P. Stated yet another way, concepts determine maximal sets of paths with identical lines (and maximal sets of lines with identical paths). The *frequency* of a concept (P,L) is simply the sum of the frequencies of paths in P. The *weighted frequency* of a concept (P,L) is its frequency times the number of instructions constituting the set of lines L. These two metrics can be used to order the concepts. Concepts can also be ordered by

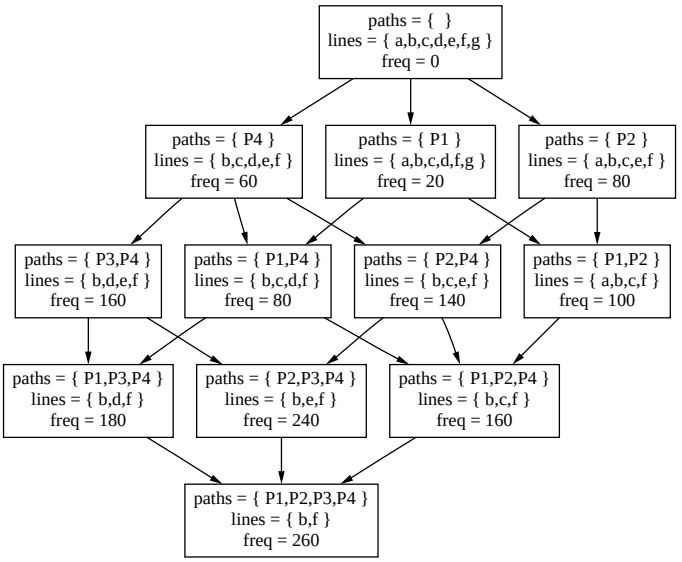

Figure 4: Concept lattice of paths versus lines

set inclusion on paths or lines. In fact, the set of all concepts forms a complete partial order, given by:

$$(P_1, L_1) \sqsubseteq (P_2, L_2) \iff P_1 \subseteq P_2 \iff L_2 \subseteq L_1$$

Figure 4 provides a simple example of the concept lattice that is output from the following four paths:

```
Path   |  Lines/Blocks   | Frequency
-----------------------------------
P1     | a b c d   f g   |       20
P2     | a b c   e f     |       80
P3     |   b   d e f     |      100
P4     |   b c d e f     |       60
```

As can be seen, this picture shows the relationships between path overlap and path frequency. We plan to investigate the use of concept analysis in HPB and perhaps provide a view to allow the user to explore hot subpaths using the concept lattice.

4.3 Graph View

We also plan to add a view of the control-flow graph to HPB. This view will executed paths to be examined in the context of the entire control flow of a procedure. While this is possible in the annotated source code view, a graph view will be a useful complement.

Acknowledgements

Thanks to Dan Hoffman for providing the red-black tree example and many discussions on the topic of path pro-

filing and infeasible paths. Thanks also to Mike Siff for providing his concept analysis implementation.

References

[ABL97] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, June 1997. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation.

[BGS97] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[BL96] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[SE94] Amitabh
Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Notices*, 29(6):196–205, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[Sne96] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.