IGOR: A tool for developing Prolog dataflow analyzers

Magnus Nordin        Thomas Lindgren
                      Håkan Millroth

Computing Science Department, Uppsala University
Box 311, S-751 05 Uppsala, Sweden
Phone: +46 18 182500
Fax: +46 18 511925
e-mail: {mgn,thomas1,hakarm}@csd.uu.se

Box 311, S-751 05 Uppsala, Sweden
Phone: +48–18–182500
Fax: +46–18–511925

Abstract

We describe a tool, IGOR, for implementing, testing, modifying, and evaluating abstract domains for analysis of Prolog programs. A high-level specification language is used for specifying abstract domains that are compiled into Prolog and interfaced with a fixpoint engine to make up a complete analyzer. The compiler automatically generates code for basic domain operations from special domain type definitions. These definition are also used to combine and reduce domains. The special purpose language provides primitives, such as set and lattice operations, and a concise method for specifying abstract interpretation of built-in predicates. We evaluate the tool and show that the high-level specifications are close to an order of magnitude less voluminous than the corresponding Prolog code and that the execution speed of the generated code is close to that of hand-written analyzers.

Keywords: automatic generation, abstract domains, program analysis, logic programming
1 Introduction

Dataflow analysis is increasingly being used for optimization of programming language implementations. However, the engineering effort required to develop analyzers often limits the pace in which new or improved analyses can be tested and evaluated. For example, we found that altering or extending the implementation of a moderately complex (120 Kb of Prolog code) Prolog analyzer took more than one person-week even for relatively straightforward changes. Major redesigns or extensions would probably require complete reimplementation, a task of several person-months.

To improve upon this situation, we have designed and implemented a tool called Igor that, given a high-level operational specification of the analysis domain, generates large parts of the analyzer automatically. The performance of the automatically-generated code is close or equivalent to handwritten code.

In our experience, the tool greatly simplifies implementation, debugging and evaluation of dataflow analyzers. We have used it for designing new analyses, for combining analyses, and for reimplementing other's analyses (while reimplementing several well-known analyses, we found minor errors in some of their specifications—an indication, perhaps, that a tool like this could be useful also to others).

Domain specifications are written in a first-order, statically typed strict functional language. This language has operations for manipulating sets and lattices, for projecting domains, and for combining domains. Specifications are compiled to Prolog code and optionally linked with an analysis framework based on Getzinger's algorithm [10]. There is a completely customizable interface to the automatically-generated domain code for users that want to provide their own fixpoint engines. The system provides support for concise specification of builtin operations and for communicating analysis results to the subsequent phases of the compiler.

A full description of the Igor tool is available elsewhere [15].

2 A simple example

The user specifies domain types. Several kinds of domain types are supported: sets ordered by inclusion, product domains, atomic function domains, finite lattices, recursive domains, and disjunctive domains. When a type definition is compiled, the tool generates Prolog code for the meet, join, and comparison operations on that type. It also provides handles to the top and bottom elements of the domain.

Consider the standard domain for mode analysis of Prolog:
This domain is defined as the type

type mode => order([[any, nonvar, ground, none],
                   [any, var, none]])

The compiler generates code for the operations

- mode_top
- mode_bot
- mode_meet(A, B)
- mode_join(A, B)
- mode_leq(A, B)

where A and B are lattice elements. The user can override these operations with his own if needed.

The mode domain above must be combined with a domain for variable aliasing (since it is not substitution-closed [8]). We can specify an aliasing domain that is a set of sets of variables in a clause C as follows.

type aliasing(C) => set(set(variables(C))).

The combined mode and aliasing domain is specified as a product domain in the following way.

mode_map(C) => variables(C) -> mode.

type mode_and_alias(C) => (mode_map(C), aliasing(C)).

However, we need not associate alias information with the ground element in the mode domain. Hence we define a projection that expresses this fact:
mode_and_alias_proj((ModeMap, Aliasing)) =>
(ModeMap,
   \{X \mid X \leftarrow P, \text{ModeMap} @ X \subseteq \text{ground} \mid P \leftarrow \text{Aliasing} \setminus \{\}\})

This projection removes all ground variables from the aliasing components of the mode_and_alias domain.

3 Language features

Besides basic lattice operations, the language supports several features that are useful when specifying abstract domains.

3.1 Set expressions

The language includes expressions for traversing sets, mapping functions on sets, and universal or existential quantification over set elements. In particular, the use of set expressions allows concise definitions of aliasing properties.

**Example.** Consider the set expression

\{\{f(X) \mid X \leftarrow \text{SubSet}, p(X)\} \mid \text{SubSet} \leftarrow \text{Set}\}

It maps the function $f/1$ on all elements $X$, satisfying property $p$, drawn from the set of sets $\text{Set}$.

**Example.** Consider the existential quantification

exists(X \leftarrow S1, is_subset(X, S2) \setminus X \subseteq \{\})

It checks the existence of an element, $X$, of set $S1$, which is a nonempty subset of set $S2$.

Set expressions are compiled into (possibly nested) loops traversing the sets. Sets of known cardinality are implemented as bit vectors. The type-checker determines if a set can be represented as a bit vector. Measurements show that the choice of using bit vectors or not depends on which set operations the specified domain primarily relies on. Compared to an ordered list representation, the bit vector representation gives more efficient union, intersection, and member operations, at the price of a somewhat higher cost for set traversal.
3.2 Projections

Projections makes it possible to remove redundant elements from a domain, or to bound an infinite domain, by mapping multiple domain elements to a single element. Another application of projections is to decrease analysis complexity by making the domain coarser. Whenever a projection is defined for a particular domain, all operations on the domain are ‘filtered’ through the projection.

**Example.** Assume that a projection `mode_proj` is defined for the domain `mode`. Then `mode_proj(mode_join(A,B))` will replace the original `mode_join(A,B)` operation.

The system does not check that a projection on a domain type is a projection in the strict mathematical sense (i.e., that it is idempotent and extensive). Hence any function over a domain could be defined as a projection.

3.3 Attributed domains

It is sometimes convenient to be able to include references to untyped data in domain declarations. Such data are called *attributes* and the domains they occur in are called *attributed domains*.

**Example.** Consider the domain

```
<table>
<thead>
<tr>
<th>any</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>... num(x_i)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
| none
```

Here the `x`'s are attributes, representing the integers. The domain, call it `numbers`, can be specified by the declaration

```
type numbers => order([[any,num,num(N),none]])
```

Here it is convenient to treat the integers as attributes, since we have no way of declaring an infinite, unordered set in the language.
3.4 Recursive and disjunctive domains

Some domains, such as depth-k abstracted terms, are potentially infinitely wide and unbounded in depth. These domains, which are useful for keeping track of the structure of compound terms, can be described by recursive domains in the language. The elements of recursive domains are trees whose nodes may contain descriptions of functors and argument types. Of course, the user will bound the depth and width of the tree by applying projections in the analysis.

**Example.** Assume that we are interested in tracking nonvariable terms in general, and compound terms in particular. Furthermore, for arguments of the compound terms we are interested in recursively tracking nonvariable and compound terms. This can be achieved with the attributed recursive domain

\[
\text{type term} \Rightarrow \text{order}([\text{any}, \text{nonvar}, \text{str} (F, \text{list} (\text{term})), \text{none}])
\]

Here \( F \) is an attribute representing the functors of compound terms. An example of an element of this domain is \( \text{str} (f, \text{nonvar}, \text{str} (g, \text{any})) \), representing all terms \( f(A, g(B)) \) where \( A \) is a nonvariable and \( B \) is any term.

The language also supports disjunctive domains. Let \( \gamma \) be the 'concretization' function that maps abstract values to terms. A disjunctive domain \( D \) is formed from a domain \( D_0 \) by introducing extra elements. A new element \( \text{or} (\{a_1, \ldots, a_k\}) \), where \( a_1, \ldots, a_k \in D_0 \), is introduced to \( D \), if it is not known that \( \gamma (a_1 \cup \cdots \cup a_k) = \gamma (a_1) \cup \cdots \cup \gamma (a_k) \). This condition can, in general, only be verified by the user. The system adds the new element to the domain unless there is one disjunct, \( a_n \), in \( a_1, \ldots, a_k \) that is greater than all the other members of this set of disjuncts, in which case only \( a_n \) is used.

**Example.** Consider a simple domain for tracking whether numbers are negative, zero, or positive:

\[
\text{type sign} \Rightarrow \text{order}([\text{any}, \text{neg}, \text{none}],
\quad \quad \quad [\text{any}, \text{zero}, \text{none}],
\quad \quad \quad [\text{any}, \text{pos}, \text{none}])
\]

We can make this a disjunctive domain by declaring it as:

\[
\text{type sign} \Rightarrow \text{disj-order}([\text{any}, \text{neg}, \text{none}],
\quad \quad \quad [\text{any}, \text{zero}, \text{none}],
\quad \quad \quad [\text{any}, \text{pos}, \text{none}])
\]
Now we express that an element is, e.g., either positive or zero: \( \text{or(zero, pos)} \).

Recursive and disjunctive domains can be combined to form recursive disjunctive domains.

**Example.** Consider again the recursive domain discussed above.

\[
\text{type term => order([[any, nonvar, str(F, list(term)), none]])}
\]

With this domain we cannot keep track of structures with different functors. For example, the join of \( \text{str(f, [nonvar])} \) and \( \text{str(g, [nonvar])} \) is \( \text{nonvar} \). However, we can keep track of different functors by making the domain disjunctive:

\[
\text{type term => disj_order([[any, nonvar, str(F, list(term)), none]])}
\]

In this domain, \( \text{or(\{str(f, [nonvar]), str(f, [nonvar])\})} \) is the join of \( \text{str(f, [nonvar])} \) and \( \text{str(g, [nonvar])} \).

3.5 **Pragmatics**

A tool like Igor must be able to handle realistically-sized programs, not just small toy benchmarks. This requires support for various engineering issues in the development of analyzers:

- **Normalization.** Most analyses assume that programs are written on a normalized form. Igor provides automatic normalization, with several options adjustable by the user.

- **Inspecting and decomposing programs.** An analyzer often needs to inspect or decompose clauses and procedures in different ways. Igor provides an extensive library to support this.

- **Builtin operations.** Specifying the builtin operations of the source language is often a tedious process. For example, SICStus Prolog provides more than 250 builtin operations, most of which are irrelevant to compiler analyses but which still require handling by the abstract domain in order to analyze real programs. Igor provides support for concisely specifying the effects of collections of builtin operations.

- **Annotated output.** Users can have widely different uses of their analyses. Each use might require a different output format. Igor annotates the program in a way that can be directed by the user. Optionally, the annotated program can be prettyprinted.
4 A larger example

Consider a domain `struct` that keep tracks of structures, lists, and constants:

This domain is specified by the following declaration:

```
type struct => disj_order([  [any,str,str(F,list(struct)),none],  [str,lst(struct),nil]  [str,atomic,atom,nil,none],  [atomic,number,int,none],  [number,float,none]]).
```

We would like to combine this domain with the mode and alias domains from Section 2, to form a more expressive domain. This is straight-forward:

```
type struct_and_mode => (struct, mode).
type struct_and_mode_map(C) => variables(C) -> struct_and_mode.
type descriptor(C) => (struct_and_mode_map(C), aliasing(C)).
```

As when we combined mode and aliasing information in Section 2, we need not associate alias information with the ground element in the mode domain. We use the following projection to achieve this effect.

```
descriptor_proj((StructAndModeMap, Aliasing)) =>
  (StructAndModeMap,
   {{X | X <- P, StructAndModeMap @ X \= (\_,ground)}
    | P <- Aliasing} \ {{}}
  ).
```
<table>
<thead>
<tr>
<th>Domain</th>
<th>Sund</th>
<th>J&amp;L</th>
<th>Str</th>
<th>Deb</th>
<th>Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation time</td>
<td>11.8</td>
<td>6.2</td>
<td>15.5</td>
<td>8.4</td>
<td>15.3</td>
</tr>
<tr>
<td>Original code size (Kb)</td>
<td>3.9</td>
<td>1.8</td>
<td>3.5</td>
<td>2.8</td>
<td>3.9</td>
</tr>
<tr>
<td>Generated code size (Kb)</td>
<td>25.1</td>
<td>14.3</td>
<td>31.7</td>
<td>18.1</td>
<td>28.6</td>
</tr>
<tr>
<td>Size ratio (generated/original)</td>
<td>6.4</td>
<td>7.9</td>
<td>9.1</td>
<td>6.5</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Table 1: Domain and compilation statistics.

Of course, combining the mode and struct domains requires that we change the struct domain declaration as follows.

```prolog
type struct => disj_order([  
    [any,str,str(F, list(struct_and_mode)), none],  
    [str, lst(struct_and_mode), nil],  
    [str, atomic, atom, nil, none],  
    [atomic, number, int, none],  
    [number, float, none]]).
```

5 Evaluation

We evaluate the following: the size of the domain specifications, the time to compile specifications, the efficiency of the generated code compared to hand-coded implementations, and the efficiency of different set representations.

All measurements were made on a Sun 630 MP with a 55 Mhz processor and 128 Mb of memory. The time unit is seconds. The benchmarks were interrupted if they had not completed within 1000 seconds. SICStus Prolog [3] version 2.1.9, with the fastcode option on, was used.

The domains used in the evaluation are Sund, Sundararajan’s domain for freeness, sharing, and linearity [17]; J&L, Jacobs's and Langen's sharing domain [13]; Str, a simple depth-k structure domain; Deb, one of Debray’s substitution-closed type domains [8]; Dep, Debray’s mode and dependency domain [7]. The set of programs analyzed in the evaluation is a subset of the Berkeley benchmarks [16].

5.1 Domain and compilation statistics

The size of the specifications of domains we have implemented range between 1 to 3 pages of non-commented code. These specifications include basic domain operations, abstract unification, abstract interpretation of 30 built-in predicates, and the code to interface the domain with the provided
The size of the uncommented domain specification is often close to the size of the published specification of the abstract domain.

The size of the generated code is 6–9 times larger than the original specification, for our examples.

The compilation time is important as it determines the turn-around time for the system. A compilation typically takes between 5–20 seconds. The majority of this time is spent in the type-checker.

The details of these domain and compilation statistics are given in Table 1.

5.2 Efficiency of generated code

Comparisons of hand-coded with auto-generated domains were performed as follows. The hand-coded freeness, sharing, and linearity analysis (called shfrson) of &-Prolog [12] was compared with Sund. The type, mode, aliasing, linearity, locality and determinism analysis of Reform Prolog [2, 14] was compared with Dep. The compared domains are not identical but similar enough to serve for our approximate comparisons. Only the execution time for analysis is included in the measurements. Program loading, code preparation, presentation of the results and similar phases are left out.

Some entries are left blank in the evaluation. These are for benchmark programs with large numbers of variables and domains that are exponential in the number of variables (Sund, J&L, and &-Prolog’s shfrson-domain).

The efficiency of the generated code compares well with similar hand-coded implementations. As can be seen in Table 2, the performance of the generated code is, on average, well within an order of magnitude of the hand-coded domains. Most of the time the performance of the generated code is within a factor 0.5–3 of the hand-coded domain.

5.3 Set representations

As mentioned previously, bit vectors are used to represent sets wherever possible. As can be seen in Tables 3–4, significant gains can be achieved by using the bit vector representation when the domains rely heavily on union, intersection, and member operations performed on very large sets (Sund and J&L).

6 Related work

The Z1 system [20] allows the programmer to specify an analyzer and an abstract domain which is compiled into executable code. Our system extends the capabilities of Z1 with disjunctive and structure-based domains and more flexible projection operations. Igor is furthermore substantially faster. We do not, however, include the analysis framework in the specifications.
Comparison of hand-coded domains and generated domains

<table>
<thead>
<tr>
<th>Program</th>
<th>#Prolog</th>
<th>Sund</th>
<th>ratio</th>
<th>Reform</th>
<th>Dep</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>6.82</td>
<td>2.93</td>
<td>0.43</td>
<td>4.49</td>
<td>2.26</td>
<td>0.50</td>
</tr>
<tr>
<td>browse</td>
<td>5.33</td>
<td>30.86</td>
<td>5.79</td>
<td>1.06</td>
<td>1.81</td>
<td>1.71</td>
</tr>
<tr>
<td>chatparser</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>23.10</td>
<td>57.92</td>
<td>2.51</td>
</tr>
<tr>
<td>crypt</td>
<td>1.84</td>
<td>0.83</td>
<td>0.45</td>
<td>0.39</td>
<td>1.37</td>
<td>3.51</td>
</tr>
<tr>
<td>divide</td>
<td>3.17</td>
<td>0.26</td>
<td>0.08</td>
<td>0.13</td>
<td>0.56</td>
<td>3.33</td>
</tr>
<tr>
<td>fastmu</td>
<td>0.95</td>
<td>0.45</td>
<td>0.47</td>
<td>1.00</td>
<td>2.30</td>
<td>2.30</td>
</tr>
<tr>
<td>flatten</td>
<td>15.95</td>
<td>3.37</td>
<td>0.21</td>
<td>2.09</td>
<td>4.00</td>
<td>1.91</td>
</tr>
<tr>
<td>metaqsort</td>
<td>3.17</td>
<td>0.86</td>
<td>0.27</td>
<td>0.86</td>
<td>0.84</td>
<td>0.98</td>
</tr>
<tr>
<td>poly</td>
<td>0.78</td>
<td>4.93</td>
<td>6.32</td>
<td>0.43</td>
<td>2.88</td>
<td>6.70</td>
</tr>
<tr>
<td>qsort</td>
<td>0.11</td>
<td>0.10</td>
<td>0.91</td>
<td>0.21</td>
<td>0.36</td>
<td>2.67</td>
</tr>
<tr>
<td>queens</td>
<td>0.14</td>
<td>0.14</td>
<td>1.00</td>
<td>0.21</td>
<td>0.46</td>
<td>2.19</td>
</tr>
<tr>
<td>reducer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.85</td>
<td>5.71</td>
<td>1.48</td>
</tr>
<tr>
<td>serialise</td>
<td>0.51</td>
<td>0.63</td>
<td>1.24</td>
<td>0.76</td>
<td>0.79</td>
<td>1.04</td>
</tr>
<tr>
<td>analyzer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9.03</td>
<td>14.32</td>
<td>1.59</td>
</tr>
<tr>
<td>tak</td>
<td>0.09</td>
<td>0.06</td>
<td>0.67</td>
<td>0.06</td>
<td>0.18</td>
<td>3.00</td>
</tr>
<tr>
<td>zebra</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.52</td>
<td>8.21</td>
<td>15.79</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>0.69</td>
<td>-</td>
<td>-</td>
<td>2.30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Execution time ratios (generated/hand-coded).

Venkatesh [19] designed a denotational semantics specification language augmented with a collecting semantics mechanism for program analysis. The specifications are interpreted rather than compiled.

Tjiang [18] describes a tool that greatly simplifies the implementation of optimizers by using high-level specifications to combine several simpler optimization specifications. This tool works with flow-graphs and is aimed at imperative rather than declarative languages.

Cortesi et al [5] propose two kinds of support for domain construction. *Generic pattern domains* is software support for upgrading simpler domains to include structural information. This upgrade results in more accurate domains. *Open products* is a method for combining domains to obtain a more sophisticated domain. This method is successfully used in IGOR specifications.


Van Roy [16] notes that the use of analysis results is less well-researched, as compared to generic analysis frameworks or abstract domains. Getzinger
Table 3: Analysis execution times.

<table>
<thead>
<tr>
<th>Program</th>
<th>Sund</th>
<th>JDL</th>
<th>Str</th>
<th>Deb</th>
<th>Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits</td>
<td>lists</td>
<td>bits</td>
<td>lists</td>
<td>bits</td>
</tr>
<tr>
<td>boyer</td>
<td>2.93</td>
<td>2.94</td>
<td>2.71</td>
<td>8.94</td>
<td>5.07</td>
</tr>
<tr>
<td>browse</td>
<td>30.86</td>
<td>377.99</td>
<td>55.39</td>
<td>807.95</td>
<td>4.24</td>
</tr>
<tr>
<td>chart parser</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15.95</td>
</tr>
<tr>
<td>crypt</td>
<td>0.83</td>
<td>0.79</td>
<td>0.80</td>
<td>2.23</td>
<td>2.00</td>
</tr>
<tr>
<td>divide</td>
<td>0.26</td>
<td>0.28</td>
<td>0.16</td>
<td>0.19</td>
<td>0.69</td>
</tr>
<tr>
<td>fastmu</td>
<td>0.45</td>
<td>0.81</td>
<td>0.30</td>
<td>0.30</td>
<td>1.27</td>
</tr>
<tr>
<td>flatten</td>
<td>3.33</td>
<td>7.90</td>
<td>18.95</td>
<td>272.16</td>
<td>2.00</td>
</tr>
<tr>
<td>metasort</td>
<td>0.86</td>
<td>1.43</td>
<td>1.08</td>
<td>6.39</td>
<td>0.90</td>
</tr>
<tr>
<td>poly</td>
<td>4.93</td>
<td>17.33</td>
<td>17.51</td>
<td>326.44</td>
<td>1.61</td>
</tr>
<tr>
<td>qsort</td>
<td>0.10</td>
<td>0.13</td>
<td>0.03</td>
<td>0.06</td>
<td>0.62</td>
</tr>
<tr>
<td>queens</td>
<td>0.14</td>
<td>0.16</td>
<td>0.05</td>
<td>0.08</td>
<td>0.55</td>
</tr>
<tr>
<td>reducer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.12</td>
</tr>
<tr>
<td>serialize</td>
<td>0.63</td>
<td>0.81</td>
<td>2.48</td>
<td>31.19</td>
<td>1.09</td>
</tr>
<tr>
<td>analyzer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8.27</td>
</tr>
<tr>
<td>tak</td>
<td>0.06</td>
<td>0.08</td>
<td>0.02</td>
<td>0.04</td>
<td>0.31</td>
</tr>
<tr>
<td>zebra</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.75</td>
</tr>
</tbody>
</table>

Set representation, bit vectors vs ordered lists

[10] performs an evaluation of the advantages gained from of a large number of domains, when used to compile logic programs. In Igor, this type of domain evaluation is facilitated by the provided support for specifying code annotations, information which can be utilized by any subsequent compilation phase.

7 Conclusion

We envision language implementors and researchers to use the Igor tool for several purposes:

- To reduce the effort required for producing a compiler using static analysis.
- To reduce the effort required for quantitative evaluation of new domain designs.
- To test and debug ideas and specifications during domain design.

The tool should be instrumental in helping to change the task of implementing static analysis domains from being a black art into routine tasks on the same level as using lex or yacc for lexical analysis and parsing.

The language does not provide support for type graphs [11] or definite boolean functions [1]. It would be useful to extend the language to support such operations more efficiently.
Set representation, bit vectors vs ordered lists

<table>
<thead>
<tr>
<th>Program</th>
<th>Sund</th>
<th>J&amp;L</th>
<th>Str</th>
<th>Deb</th>
<th>Dep</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>1.00</td>
<td>3.23</td>
<td>0.82</td>
<td>0.89</td>
<td>1.04</td>
</tr>
<tr>
<td>browse</td>
<td>12.25</td>
<td>14.64</td>
<td>1.14</td>
<td>0.93</td>
<td>1.06</td>
</tr>
<tr>
<td>chatparser</td>
<td>-</td>
<td>-</td>
<td>0.67</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>crypt</td>
<td>0.95</td>
<td>2.51</td>
<td>1.06</td>
<td>1.04</td>
<td>0.91</td>
</tr>
<tr>
<td>divide</td>
<td>1.08</td>
<td>1.19</td>
<td>1.06</td>
<td>0.92</td>
<td>1.02</td>
</tr>
<tr>
<td>fastmu</td>
<td>1.80</td>
<td>1.00</td>
<td>1.04</td>
<td>1.65</td>
<td>0.98</td>
</tr>
<tr>
<td>flatten</td>
<td>2.37</td>
<td>14.36</td>
<td>1.07</td>
<td>0.95</td>
<td>1.10</td>
</tr>
<tr>
<td>metaqsort</td>
<td>1.66</td>
<td>5.92</td>
<td>0.98</td>
<td>0.71</td>
<td>1.02</td>
</tr>
<tr>
<td>poly</td>
<td>3.52</td>
<td>18.63</td>
<td>0.91</td>
<td>0.78</td>
<td>1.01</td>
</tr>
<tr>
<td>qsort</td>
<td>1.30</td>
<td>2.00</td>
<td>1.06</td>
<td>0.81</td>
<td>1.04</td>
</tr>
<tr>
<td>queens</td>
<td>1.14</td>
<td>1.60</td>
<td>1.05</td>
<td>0.85</td>
<td>0.93</td>
</tr>
<tr>
<td>reducer</td>
<td>-</td>
<td>-</td>
<td>1.05</td>
<td>0.84</td>
<td>1.12</td>
</tr>
<tr>
<td>serialise</td>
<td>1.29</td>
<td>12.58</td>
<td>0.98</td>
<td>0.82</td>
<td>1.05</td>
</tr>
<tr>
<td>analyzer</td>
<td>-</td>
<td>-</td>
<td>0.99</td>
<td>0.81</td>
<td>0.86</td>
</tr>
<tr>
<td>tak</td>
<td>1.33</td>
<td>2.00</td>
<td>1.06</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>zebra</td>
<td>-</td>
<td>-</td>
<td>1.90</td>
<td>1.56</td>
<td>0.92</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.74</td>
<td>4.02</td>
<td>1.03</td>
<td>0.95</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 4: Execution time ratios (ordered lists/bit vectors).

Acknowledgment

We thank Per Mildner whose frequent critique significantly improved this work (and inspired the name of our tool).

References


