Type Graphs in Practice

Per Mildner, Thomas Lindgren
e-mail: {perm,thomasl}@csd.uu.se
Computing Science Dept., Uppsala University
Box 311, S-751 05 Uppsala, Sweden
Phone: +46–18–18 25 00
Fax: +46–18–51 19 25

Abstract

Type graphs have been proposed as a domain suitable for inferring disjunctive and recursive types of Prolog programs through abstract interpretation.

Of central importance are the operations used to ensure termination of the analysis, i.e., the methods used to introduce recursiveness in the inferred types. We evaluate performance of several such methods on some Prolog programs of realistic size. We show that none of the methods previously proposed in the literature are, in fact, feasible as described.

We also show that the problems are not primarily dependent on the size of the analysed program, but instead on the complexity, and thus size, of the intermediate types. We propose and evaluate some methods to overcome these problems.

We conclude that using the upper bound operation on type graphs will, in practice, make the analysis infeasible for many typical programs.

1 Background

Inferring, at compile time, the recursive structure of arguments that will appear at runtime is important for many advanced compiler optimisation [11, 12].

Optimisations that benefit from such information include replacing expensive general operations with simpler special case operations, e.g., replacing general unification with matching; determining that parts of the program will never fail, thus enabling a more efficient execution model; compile time reuse, i.e., statically determine that a term appearing at runtime can be destructively reused.
2 Relation to other research

Type graphs were introduced by Janssens and Bruynooghe [8] as a domain for inferring disjunctive and recursive types of Prolog programs through abstract interpretation. Their work, including the thesis by Janssens [7] established a firm theoretical basis for type graphs and their associated operations (mainly upper bound of two type graphs, lower bound of two type graphs, comparing the denotations of two type graphs and Depth-k restriction as a way to ensure termination of the analysis). They did not provide results of using the type graph domain on programs of any significant size.

Getzinger, in his thesis [5], evaluated several domains in order to evaluate their impact on the code generated by an optimising Prolog compiler. His attempt to evaluate (rigid) type graphs met with mixed success as it failed to terminate within reasonable time for many of the benchmarks used. He did not attempt to analyse the reasons for this in any depth\(^1\).

Finally, Hen tenryck et al. [6] incorporated rigid type graphs into their generic analyser framework GAIA and proposed a novel widening operator. To our knowledges this is the only previous work that provides information on using type graphs not only for “toy” programs. Their article does not say how numbers or primitives such as \texttt{is/2} and \texttt{arg/3} were handled.

3 Our Analyser

The present work uses an analyser framework, implemented in Prolog, similar to that used by Getzinger in his thesis [5], i.e., a mono-variant top-down analyser. The analyser ignores cut as well as clause-order but observes the left to right evaluation order for clause bodies.

For every predicate a single pair of call and success pattern is maintained.

During analysis of a clause the analyser maintains an abstract substitution, i.e., environment, for every program point. There is basically one program point between every goal in a clause and at the beginning and end of the clause body.

An abstract environment has two components \(T\) and \(SVAL\).

\(T\) maps every program variable in the environment to a set of possible concrete terms, represented by a type graph.

Additionally \(SVAL\) maintains information about what concrete sub-terms will definitely be equal, in the sense of \(==/2\), at run-time.

Primitives

Good abstraction of primitives, i.e., builtins, can often make a big difference in the precision obtained. In particular since output from a unhandled builtins will necessarily be abstracted as \texttt{any} which in turn will annihilite any other information if a disjunctive type is formed.

Primitives are currently handled straightforwardly by using a fixed mapping from the call pattern \texttt{any} to a the proper success pattern. The success\(^2\)

\(^1\)Some of the analyser variants we tried does terminate for these programs.
pattern will for most, but not all, primitives contain no information about aliasing.

4 Our Abstract Domains

As mentioned above the environment provides information about sets of possible values for program variables, represented as type graphs, as well as information about definite equality, often called aliasing, between (subterms of) program variables.

Type Graphs

In its simplest form a type graph is a tree with, in our case, four kinds of nodes; **functor**-nodes, **or**-nodes, **any**-nodes and **num**-nodes.

In addition to this, to allow recursively defined types, a *back arc* from a node is allowed to point to an ancestor of the node.

The denotation of a **num** node is the set of all numbers, the denotation of a **any**-node is the set of all terms.

A **functor**-node has an associated *label*, a name and arity *n* and ordered children numbered 1..*n*. It denotes a set of concrete terms all with the same name and arity as the *label* and with children denoted by the children of the **functor**-node.

A **functor**-node with arity zero is used to abstract atoms, e.g., \(\square\).

A **or**-node has *n* unordered children, *n* > 1 and denotes the union of the denotations of its children.

There are some additional constraints imposed on type graphs, for details see [6, 8].

Note that there are no nodes denoting the set of unbound, i.e., free, variables. Instead an **any**-node will have to be used. The disadvantage to this is that information sometimes is lost when free variables occurs. The advantage is that it makes our domain instantiation-closed [4]. This avoids the need for the expensive and complex tracking of possible aliasing between variables.

Abstract Atomic Values

All numbers are abstracted to the same abstract **num** value. This loses information about specific numeric constants but on the other hand allows us to retain the information that the result of, e.g., the builtin \(\text{is}/2\) will be a number. This was found crucial to get reasonable precision for several benchmarks.

As an example; if we wish to abstract the set of simple arithmetic expressions we would expect to get a disjunctive type with **num** as one of the disjuncts and \(+/2\), \(-/2\), etc as the other disjuncts.

If we lack a way to express that a value is a number then we would always be forced to abstract such an arithmetic expression as the unknown value **any**.

As mentioned above we treat other atomic values as functors with arity zero.
Abstract Aliasing

The basic type graph domain is augmented with definite same-value information, in much the same way as done by Janssens and Bruynooghe [8]. This differs from the pattern domain used for the same purpose in GAIA [2]. The difference is unlikely to affect our results in any significant way as the properties of the type graph operations does not depend on the aliasing information.

5 Type Graph Widening

To ensure termination of the analysis it is necessary that there are finitely many successive approximations of the concrete call and success patterns for a predicate.

This property holds trivially for many simple domains, especially where the domains consist of a finite set of domain values.

For type graphs it does not hold as it is possible to construct an infinite number of type graphs describing successively larger sets of concrete values.

To ensure termination of the analysis for these kinds of domains a widening can be used [3] to ensure that any such sequence of successively less precise approximations is stationary, i.e., that the analyser will reach a final approximation in a finite number of steps.

For type graphs this translates to a method that generalises a type graph by introducing back arcs and/or any-nodes and in the process increasing the set of concrete terms the type graph describes.

By introducing back arcs, what we call folding, the type described by a type graph becomes recursive.

Typically the widening operation is applied when combining a new and an old call pattern for a predicate and when combining a new and an old success pattern for a predicate.

Depth-k Restriction

One way of ensuring termination is to limit the number of times a specific functor can occur on a forward path from the root of the type graph [3] [8, 7]. Assuming a finite number of distinct functors in the analysed program this implies a finite maximum size of a type graph and thus ensures termination.

When a depth restriction is detected it must be resolved by introducing a back arc to an ancestor node with a greater or equal denotation. The existence of such an ancestor node is ensured by forming the upper bound of the selected ancestor node with the node that violated the depth restriction.

---

3I.e., it does not obey the “finite ascending chain” condition.

5A forward path is a path between two nodes that does not traverse any back arcs.

5This happens surprisingly rarely. When analysing the benchmark **nand** with Depth-2 less than ten percent of the depth violation tests succeed.

5Selecting a suitable ancestor node can be done in several ways. We currently select the nearest suitable node where a suitable node is either a functor-node with the same name/arity as the replaced node or an or-node with a child with the same name/arity as the removed node. For all the details see [7].
and then repeating the process until no violation of the Depth-$k$ restriction remains.

We have evaluated the case where $k = 2$, i.e., when no functor is allowed to appear more than twice on a forward path.

In the tables below this method is denoted $Depth-2$.

**Topological Clash**

In order to better guide the generalisation of the type graphs Hentenryck et al. used both the old type graph and the upper bound of the old and new type graph for a particular argument position [6].

Comparing the two type graphs they use the mismatching nodes, what they termed topological clashes, to guide the introduction of back arcs.

When resolving a topological clash the clashing node is replaced by a back arc, either to a node with greater denotation than the replaced node, or, if no such node exist, an upper bound is formed of the clashing node and a suitable ancestor. To ensure termination this upper bound must decrease the size of the type graph. A simple method is to use an any-node but it is possible to obtain better results by actually performing an upper bound and ensuring the size limit in some way.

It is not clear what method was used in [6]. Our implementation uses the ordinary upper bound operation and then applies successively aggressive methods to bring the size down to below the limit. As a last, but in practice never encountered, resort we fall back to using an any-node when the upper bound cannot otherwise be made small enough.

In the tables below this method is denoted $TC$.

**Type Jungles**

Type jungles ensure finiteness by requiring that a particular functor be represented by identical nodes wherever it appears in a type graph. Thus it can be seen as a further restriction upon $Depth-1$.

Type jungles have the nice property that they allow upper bound of two (or more) type jungles to be computed using a particularly compact intermediate representation (basically a dictionary with one entry per functor). This means that for the cases where the analyser would normally compute an upper bound of two type graphs and then do a generalisation, it can instead use a specialised upper bound using type jungles to obtain the widened result directly.

As used here, we still need to transform the type jungle from its compact representation to a type graph. In a companion paper we investigate an analyser using the compact type jungle representation throughout the analysis [9].

In the tables below this method is denoted $Jungle$.

6 Additional Heuristics

We also tried some additional methods piggybacked upon the other widenings in an attempt to alleviate some of their problems.
Fold Equal Nodes

The basic upper bound and intersection operations for type graphs do not produce a type graph that is minimal [8]. Thus there may be nodes with a denotation equal to an ancestor of the node. It is then possible to replace the node with a back arc to the ancestor, thus producing a type graph with the same denotation with fewer nodes. Unlike the other methods this method does not change the denotation of the affected type graph.

The fold-equal-nodes method is denoted with the use of the subscript $EQ$.

Fold More Precise Nodes

It is possible to obtain a smaller type graph with a larger denotation by folding nodes that have an ancestor that subsumes them. This is actually used as the preferred way to resolve topological clashes in widening $TC$, the method of Hentenryck et al..

The folding was applied immediately before the ordinary widening used.

When it applies it can significantly reduce the size of a type graph, often creating the same recursive type graph as the analyser would have determined anyway, but using fewer iterations.

Use of this method is denoted with the use of the subscript $LEQ$.

7 Evaluation

The Benchmarks

We have chosen our benchmarks from the Berkeley benchmark suite. This set of benchmarks are widely known, selected to be representative of real Prolog programs, easily obtainable$^6$ and can be executed using accompanying input data. The Berkeley benchmarks have been used [5] to evaluate the analyser and compiler in the Aquarius system. We have included the “large” benchmarks from this archive.

Additionally we used file aquarius Compiler.pl $^7$, a stand-alone version of the compiler in the Aquarius system.

We have not been able to obtain the benchmarks used by Hentenryck et al [6] in their evaluation of the $TC$ widening$^8$.

Table 1 lists the benchmarks used and Table 2 gives some indication of the size and structure of each benchmark program. The size measures are; the number of procedures, the number of clauses, the number of goals, and the total number of arguments. The size measures does not include the procedures that are unreachable from the main entrypoint. They also do

---


$^7$Available at <http://www.info.ucl.ac.be/people/PVR/aquarius Compiler.pl>. There have been a number of versions of this circulating under the name apc.pl. At least some of them are broken in ways that causes the analyser to determine that the program would fail before doing any useful work.

$^8$The benchmarks available by ftp from Brown University do not seem to be the same as those used in [6] since several size measures differ from those reported for the benchmarks in [6]. Making a meaningful comparison of our results with those reported in [6] using “similar” programs would be difficult, especially for domains as precise as those considered here.
not include calls to undefined procedures, such as builtins for which the analyser have no special handling.

**Synthetic Benchmarks**

We have also includes two very small programs that expose weaknesses in the widenings. `tree4` (Figure 1) recognises unary trees with four different node-labels. `expr` (Figure 2) recognises arithmetic expressions.

```
maintree.  

main :- tree4(.$$).
   
tree4(a).
   tree4(b(T)) :- tree4(T).
   tree4(c(T)) :- tree4(T).
   tree4(d(T)) :- tree4(T).

Figure 1: tree4.pl

main :- expr(.$$).
Note: $X =$ 'NUMBER' (.$$ is used as stand-in for number/1.
expr(X) :- X = 'NUMBER' (.$$).
expr(+(X)) :- expr(X).
expr(-X) :- expr(X).
expr(X+Y) :- expr(X), expr(Y).
expr(X-Y) :- expr(X), expr(Y).
expr(X*Y) :- expr(X), expr(Y).
expr(X/Y) :- expr(X), expr(Y).
expr(X//Y) :- expr(X), expr(Y).
expr(X mod Y) :- expr(X), expr(Y).
expr(integer(X)) :- expr(X).
expr(float(X)) :- expr(X).
expr(X # Y) :- expr(X), expr(Y).
expr(X<<Y) :- expr(X), expr(Y).
expr(X>>Y) :- expr(X), expr(Y).
expr([X]) :- expr(X).

Figure 2: expr.pl

**Measurements**

We measured the time taken for the analysis\(^9\); the number of goals analysed \((\text{Iter.})\), i.e., the number of times a new call-pattern caused a procedure to be analysed; the maximum size\(^10\) of any intermediate type graph, measured

---

\(^9\)The time does not include the time to read and pre-process the analysed program. It also does not include the time for garbage-collection, typically this requires an additional 30%.

\(^10\)The number of nodes and arcs.
Berkeley Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt.pl</td>
<td>Solve a simple crypto-arithmetic puzzle.</td>
</tr>
<tr>
<td>meta_qsort.pl</td>
<td>A meta-interpreter running qsort.</td>
</tr>
<tr>
<td>prover.pl</td>
<td>A simple theorem prover.</td>
</tr>
<tr>
<td>browse.pl</td>
<td>Build and query a database.</td>
</tr>
<tr>
<td>unify.pl</td>
<td>A compiler code generator for unification.</td>
</tr>
<tr>
<td>flatten.pl</td>
<td>Source transformation to remove disjunctions.</td>
</tr>
<tr>
<td>sdda.pl</td>
<td>A data-flow analyser that represents aliasing.</td>
</tr>
<tr>
<td>reducer.pl</td>
<td>A graph reducer based on combinators.</td>
</tr>
<tr>
<td>boyer.pl</td>
<td>An extract from a Boyer-Moore theorem prover.</td>
</tr>
<tr>
<td>simple_analyzer.pl</td>
<td>A data-flow analyser analysing qsort.</td>
</tr>
<tr>
<td>nand.pl</td>
<td>A logic synthesis program based on heuristic search.</td>
</tr>
<tr>
<td>chat_parser.pl</td>
<td>Parse a set of English sentences.</td>
</tr>
</tbody>
</table>

Other Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aquarius_compiler.pl</td>
<td>Optimising Prolog compiler.</td>
</tr>
<tr>
<td>tree4.pl</td>
<td>Recognise unary trees with four node-types.</td>
</tr>
<tr>
<td>expr.pl</td>
<td>Recognise the basic arithmetic expressions</td>
</tr>
</tbody>
</table>

Table 1: The benchmarks. For size measures see Table 2.

on the inputs to (\(|in|\)) and result of (\(|out|\)) the upper bound operation; the maximum size of any type graph in the result (\(|Result|\)). We also measured the size of the result from upper bound after folding equal nodes (\(|min(out)|\)).

For some benchmarks the analysis timed out, using a two minute time limit on all type-graph operations. For these cases we show the intermediate sizes that appeared up to that point.

The Results

We show the results for the unmodified TC and Depth-2 as well as these methods enhanced with folding of equal and more precise nodes.

We also show the result obtained by using Jungle, the widening based on type jungles.

Unmodified TC and Depth-2 As can be seen in Table 3 and Table 4 the maximum intermediate type graph size is sometimes much larger than the sizes of the final type graphs.

Both Depth-2 and TC times out on some of the benchmarks even though TC behaves better.

Of particular interest is their behaviour on tree4 and expr. Both these benchmarks were constructed to model programs that occur in practice but without introducing the precision loss that often helps the analysis terminate on larger programs.

tree4 was modeled after meta_qsort and gives the kind of success pattern obtained from a predicate recognising, e.g., the various constructs in a programming language.
expr is similar but recognises a more realistic sublanguage. A case similar to this occurred when we tried to analyse an explicit definition of the builtin `is/2`.

In contrast to Depth-2 the widening TC nicely handles tree4.

Neither method succeeds in analysing expr, and when they time out they had reached intermediate type graph sizes in excess of two million, translating to at least half a million nodes. In contrast the type-graph that would eventually result from the analysis of expr would have approximately one node per clause, for a total size below sixty.

A polyvariant analyser would likely be more susceptible to the problems exposed by these benchmarks as it would not lose precision as easily as a monovariant analyser such as ours.

As can be seen the size of the programs are not a problem in itself. Both the basic Depth-2 and TC can successfully analyse at least some of the larger programs.

Modified TC and Depth-2 Adding the heuristics of folding equal nodes after all operations and of folding nodes to less precise ancestors significantly reduces the size of the intermediate type graphs occurring for several benchmarks.

$TC_{EQ,LEQ}$ (Table 5) succeeds in analysing all the benchmarks except chat_parser. In particular $TC_{EQ,LEQ}$ quickly reaches the expected final result for expr.

Depth-2$_{EQ,LEQ}$ (Table 6) still fails to analyse expr. The analysis timed

<table>
<thead>
<tr>
<th>Name</th>
<th>#Procs</th>
<th>#Clauses</th>
<th>#Goals</th>
<th>#Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt.pl</td>
<td>9</td>
<td>27</td>
<td>29</td>
<td>18</td>
</tr>
<tr>
<td>meta_qsort.pl</td>
<td>8</td>
<td>26</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>prover.pl</td>
<td>10</td>
<td>33</td>
<td>28</td>
<td>22</td>
</tr>
<tr>
<td>browse.pl</td>
<td>14</td>
<td>29</td>
<td>29</td>
<td>42</td>
</tr>
<tr>
<td>unify.pl</td>
<td>29</td>
<td>63</td>
<td>79</td>
<td>141</td>
</tr>
<tr>
<td>flatten.pl</td>
<td>28</td>
<td>58</td>
<td>55</td>
<td>83</td>
</tr>
<tr>
<td>sdda.pl</td>
<td>28</td>
<td>77</td>
<td>69</td>
<td>78</td>
</tr>
<tr>
<td>reducer.pl</td>
<td>30</td>
<td>95</td>
<td>83</td>
<td>95</td>
</tr>
<tr>
<td>boyer.pl</td>
<td>24</td>
<td>134</td>
<td>34</td>
<td>61</td>
</tr>
<tr>
<td>simple_analyzer.pl</td>
<td>67</td>
<td>136</td>
<td>140</td>
<td>254</td>
</tr>
<tr>
<td>nand.pl</td>
<td>40</td>
<td>136</td>
<td>189</td>
<td>174</td>
</tr>
<tr>
<td>chat_parser.pl</td>
<td>155</td>
<td>494</td>
<td>345</td>
<td>742</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aquarius_compiler.pl</td>
<td>1238</td>
<td>3813</td>
<td>4683</td>
<td>4018</td>
</tr>
<tr>
<td>tree4.pl</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>expr.pl</td>
<td>2</td>
<td>16</td>
<td>24</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Size of benchmark programs.
| Name               | Iter. | \(|in|\) | \(|out|\) | \(|Result|\) | Time  |
|--------------------|-------|--------|--------|----------|-------|
| Berkeley Benchmarks|       |        |        |          |       |
| crypt              | 19    | 37     | 41     | 33       | 1.05  |
| meta_qsort         | 46    | 978    | 1282   | 279      | 21.64 |
| prover             | 46    | 631    | 4733   | 100      | 6.88  |
| browse             | 36    | 290    | 442    | 288      | 204.82|
| unify              | 61    | 127    | 83     | 61       | 2.860 |
| flatten            | 51    | 39     | 60     | 38       | 1.63  |
| sdda               | 61    | 83     | 83     | 59       | 2.83  |
| reducer            | 52    | 137    | 142    | 137      | 2.64  |
| boyer              | TO    | 41154  | 273311 | N/A      | N/A   |
| simple_analyzer    | 121   | 936    | 940    | 451      | 42.58 |
| nand               | 132   | 1185   | 3997   | 303      | 39.91 |
| chat_parser        | TO    | 4602760| 4617870| N/A      | N/A   |
| Other Benchmarks   |       |        |        |          |       |
| aquarius_compiler  | 2734  | 6523   | 27070  | 369      | 212.14|
| tree4              | 2     | 39     | 317    | 12       | 0.15  |
| expr               | TO    | 4251693| 8503379| N/A      | N/A   |

Table 3: \(TC\) with no additional heuristics.

| Name               | Iter. | \(|in|\) | \(|out|\) | \(|Result|\) | Time  |
|--------------------|-------|--------|--------|----------|-------|
| Berkeley Benchmarks|       |        |        |          |       |
| crypt              | 14    | 21     | 25     | 12       | 0.89  |
| meta_qsort         | TO    | 5444   | 44348  | N/A      | N/A   |
| prover             | TO    | 6885   | 84881  | N/A      | N/A   |
| browse             | 25    | 82     | 90     | 48       | 1.55  |
| unify              | 66    | 127    | 82     | 61       | 4.44  |
| flatten            | 54    | 60     | 61     | 38       | 2.36  |
| sdda               | 52    | 48     | 49     | 59       | 2.41  |
| reducer            | 47    | 137    | 137    | 137      | 3.12  |
| boyer              | 39    | 576    | 582    | 582      | 5.05  |
| simple_analyzer    | 130   | 534    | 584    | 94       | 7.99  |
| nand               | 162   | 633    | 1533   | 116      | 51.13 |
| Other Benchmarks   |       |        |        |          |       |
| tree4              | TO    | 1440   | 14673  | N/A      | N/A   |
| expr               | TO    | 2125847| 4251691| N/A      | N/A   |

Table 4: Depth-2 with no additional heuristics.
\begin{table}
\centering
\begin{tabular}{|l|c|c|c|c|c|c|}
\hline
\textbf{TC}_{EQ,LEQ} & & & & & & \\
\hline
\textbf{Name} & \textbf{Iter.} & \textbf{in} & \textbf{out} & \textbf{min(out)} & \textbf{Result} & \textbf{Time} \\
\hline
\textbf{Berkeley Benchmarks} & & & & & & \\
\hline
\texttt{crypt} & 19 & 17 & 21 & 21 & 20 & 0.990 \\
\texttt{meta_qsort} & 223 & 859 & 1778 & 1520 & 279 & 674.25 \\
\texttt{prover} & 43 & 138 & 835 & 381 & 84 & 5.83 \\
\texttt{browse} & 36 & 302 & 973 & 447 & 300 & 210.02 \\
\texttt{unify} & 58 & 127 & 83 & 83 & 61 & 3.28 \\
\texttt{flatten} & 51 & 37 & 85 & 73 & 30 & 2.03 \\
\texttt{sdda} & 57 & 75 & 159 & 159 & 59 & 3.25 \\
\texttt{reducer} & 48 & 131 & 142 & 138 & 131 & 3.43 \\
\texttt{boyer} & 36 & 4462 & 19164 & 8690 & 473 & 51.60 \\
\texttt{simple_analyzer} & 121 & 936 & 940 & 940 & 451 & 46.370 \\
\texttt{nand} & 120 & 255 & 981 & 237 & 134 & 32.71 \\
\texttt{chat Parser} & TO & 3078 & 7275 & 4017 & N/A & N/A \\
\hline
\textbf{Other Benchmarks} & & & & & & \\
\hline
\texttt{aquarius_compiler} & 2714 & 787 & 3148 & 3148 & 414 & 256.32 \\
\texttt{tree4} & 2 & 15 & 69 & 47 & 12 & 0.07 \\
\texttt{expr} & 2 & 101 & 1215 & 1215 & 58 & 0.67 \\
\hline
\end{tabular}
\caption{TC with EQ and LEQ folding.}
\end{table}

\begin{table}
\centering
\begin{tabular}{|l|c|c|c|c|c|c|}
\hline
\textbf{Depth-2}_{EQ,LEQ} & & & & & & \\
\hline
\textbf{Name} & \textbf{Iter.} & \textbf{in} & \textbf{out} & \textbf{min(out)} & \textbf{Result} & \textbf{Time} \\
\hline
\textbf{Berkeley Benchmarks} & & & & & & \\
\hline
\texttt{crypt} & 14 & 13 & 17 & 17 & 12 & 0.98 \\
\texttt{meta_qsort} & 48 & 223 & 524 & 524 & 90 & 6.13 \\
\texttt{prover} & 65 & 302 & 2305 & 1079 & 204 & 13.61 \\
\texttt{browse} & 24 & 54 & 110 & 50 & 48 & 2.09 \\
\texttt{unify} & 58 & 127 & 82 & 82 & 61 & 5.73 \\
\texttt{flatten} & 51 & 37 & 81 & 73 & 30 & 2.78 \\
\texttt{sdda} & 51 & 48 & 46 & 46 & 59 & 3.06 \\
\texttt{reducer} & 47 & 131 & 138 & 138 & 131 & 4.37 \\
\texttt{boyer} & 36 & 708 & 721 & 721 & 714 & 8.83 \\
\texttt{simple_analyzer} & 130 & 534 & 584 & 584 & 94 & 7.99 \\
\texttt{nand} & 137 & 255 & 1126 & 237 & 116 & 46.68 \\
\texttt{chat Parser} & TO & 4724 & 31305 & 27864 & N/A & N/A \\
\hline
\textbf{Other Benchmarks} & & & & & & \\
\hline
\texttt{aquarius_compiler} & N/A & 32041 & 136239 & 44923 & N/A & N/A \\
\texttt{tree4} & 2 & 27 & 61 & 61 & 12 & 0.08 \\
\texttt{expr} & TO & 1810919 & 3621835 & 3621835 & N/A & N/A \\
\hline
\end{tabular}
\caption{Depth-2 with EQ and LEQ folding.}
\end{table}
Table 7: *Jungle* with no additional heuristics.

| Name               | Iter. | |in| | |out| | |Result| | |Time| |
|--------------------|-------|---|---|---|---|---|---|---|---|---|---|
| Berkeley Benchmarks |       |   |   |   |   |   |   |   |   |   |   |
| crypt              |  15   | 17 |  9 | 20 |   |  0.56 |       |   |   |   |   |
| metasort           |  31   | 553| 549|  72|  2.100 |       |   |   |   |   |   |
| prover             |  63   | 316| 256| 134| 8.140 |       |   |   |   |   |   |
| browse             |  26   | 181|  20| 181|120.48 |       |   |   |   |   |   |
| unify              |  58   |  33|  9 |  21|1.800 |       |   |   |   |   |   |
| flatten            |  51   |  37| 29 |  29|1.170 |       |   |   |   |   |   |
| sdda               |  50   |  75|  46|  59|1.740 |       |   |   |   |   |   |
| reducer            |  38   | 105|160 |160 |1.440 |       |   |   |   |   |   |
| boyer              |  36   |1049|1049|1023|18.82 |       |   |   |   |   |   |
| simple_analyzer    | 133   | 451|  41| 451|21.230 |       |   |   |   |   |   |
| nand               | 123   | 209| 221|  93|27.24 |       |   |   |   |   |   |
| chat_parser        | 1587  |61425|46724|46724|511.44 |       |   |   |   |   |   |
| Other Benchmarks   |       |   |   |   |   |   |   |   |   |   |   |
| aquarius_compiler  | 2443  |6315|3156|2388|231.11 |       |   |   |   |   |   |
| tree4              |  2    |  39|  37|  12| 0.09 |       |   |   |   |   |   |
| expr               |       | N/A|N/A | N/A|N/A   |       |   |   |   |   |   |

out for both *chat_parser* and *aquarius_compiler*.

**Type Jungle widening** Using an intermediate type jungle for upper bound and widening seems to work well for most cases. Still it fails for *expr* 11.

The problem seems to be that the process of transforming from the compact type jungle representation to an explicitly represented type graph can produce a very large type graph. As a comparison the type jungle based analyser described in [9] analyses *expr* in just a few seconds.

8 Conclusion

We have shown that the previously proposed type graph widenings *Depth-k* and *TC* does not keep down the size of intermediate type graphs appearing during abstract interpretation.

We also show that for some very small programs, essentially predicates recognizing tree shaped data structures, the analysis reaches huge intermediate type graphs, in some cases consisting of several hundred thousand nodes.

We show that the main problem is the sizes of the intermediate type graphs and that the intermediate type graphs can be very large even for cases where the final result is both small and precise.

Using the heuristic of folding nodes to less precise ancestors shows that

11The analyser runs out of memory, losing the information about the intermediate type graph sizes in the process.
even methods that can keep down the size of type graphs appearing after the widening operation will not, in general, help. The reason for this appears mainly to be that the upper bound operation can create huge resulting type graphs even for input types of modest size.

We evaluated several heuristics of which Type Jungles, a further restriction on Depth-1, seems the most promising. A particular advantage of this method is that it allows an upper bound to be computed in a way that avoids the size problems with the ordinary upper bound. A problem in the current setting is that, even though type jungles allow a compact representation, the explicit representation of a type jungle as a type graph can be prohibitely large. A companion paper [9] investigates the use of type jungles without explicit representation as type graphs.

9 Future Work

It would be interesting to compare the precision obtained using the various method investigated here. In particular we would like to apply the same methods as used in [9], comparing the static precision with the dynamic precision obtained by weighting procedures by their execution frequency and comparing the static and dynamic precision with the types that actually appears when executing the analysed programs.

Another interesting question is to what extent the analyser framework affects the size of the intermediate type graphs. For the simple but problematic benchmarks tree4 and expr it seems obvious that the analyser ought to be able to reach the correct result almost immediately. It would be interesting to investigate if there are general methods that leads to the final results without reaching large intermediate type graphs.
References


