The Impact of Structure Analysis on Prolog Compilation

Thomas Lindgren
Computing Science Dept.,
Uppsala University
e-mail: thomasl@csd.uu.se
Box 311, S-751 05 Uppsala, Sweden
Phone: +48-18-1825 00
Fax: +46-18-51 19 25

Abstract

Structure analyses derive (possibly recursive) descriptions of the shape of data structures. We investigate the use of structure analysis in compiling Prolog. We implement a structure analyzer for Prolog by extending previous type graph analyses, and evaluate its precision and performance on a set of realistic benchmarks. We then attempt to quantify the use of the derived structure information for optimizing compilation.

1 Introduction

Structure analyses find information on the recursive shape of data structures. Such information is potentially useful to reduce overheads in dynamically typed languages, such as Prolog or Lisp, where type checking is conceptually deferred until runtime. This increases the expressiveness of the language at a potential cost in performance.

Dynamically typed languages devote considerable effort to checking that arguments to primitive operations are used consistently with their types. The cost of dynamic type checking has two components: a runtime cost (e.g., testing the tag and functor of a term, even though the test always has the same outcome at runtime), and a compilation cost (e.g., compiling code to handle cases that never occur at runtime).

In contrast, statically typed languages such as Mercury or SML can largely dispense with such checks. Statically typed languages declare types as recursive grammars, and then ensure that all expressions or goals are well-typed by type inference or type checking. Poorly typed programs are rejected and must be rewritten. Statically well-typed programs can omit many runtime type checks.

In contrast, a Prolog compiler normally cannot reject a procedure with a heterogeneous argument (e.g., either a number or an atom). In contrast with statically
typed languages, such type inference must be prepared to give up, wholly or partially, and infer for instance that a procedure returns "any term". At that point, type checking is deferred to runtime. Nevertheless, a number of analyses have been aimed at discovering structure information for logic programs [9, 19, 8, 10].

This paper is concerned with measuring the usefulness of type information for Prolog compilation, as derived by a structure analysis. Rather than considering call/success pattern precision, we attempt to quantify the effects that analysis information will have on compiled code. In particular, we consider the impact on how individual primitives are compiled. Improving the compilation of primitives has been the traditional use of static analysis in optimizing Prolog compilers.

We find that known-value/same-value information combined with a simplification of type graphs, which we call type jungles, provides considerable benefit to an optimizing compiler. For instance, 90% of the unifications in the benchmarks we consider, can be classified as tests or assignments. We also classify a number of other primitives.

The rest of this paper is structured as follows. Section 2 describes our abstract domain for deriving recursive structure information. Section 3 describes the analysis framework, including a supporting pattern domain and a simple method to pass information between disjunctive branches. Section 4 concerns the precision and efficiency of a prototypical analyzer with respect to optimizing compilation. Section 5 further discusses the use of the derived information in an optimizing compiler. Section 6 discusses related work. Section 7 concludes the paper with a summary and suggestions for future work.

2 An abstract domain for structure analysis

2.1 Structure information

The runtime shapes of data structures is approximated by structure descriptions. A structure description tells the compiler how compound terms may be connected. For example, a structure description may tell us that the second argument to a cons cell always is nil or another cons cell.

A type graph is a structure description in the form of a disjunctive rational tree where nodes are function nodes labelled $f/n$ (representing compound terms with functor $f/n$, $n \geq 0$), OR-nodes, or leaf nodes (e.g., representing "any term"). A rigid type graph has only instantiation closed [5] nodes and so does not require aliasing information to remain correct. For instance, a type graph describing all lists of lists of atoms red, white or blue is shown in Figure 1 (a). The interested reader is directed to Refs. [9, 19] for further information on the algorithms used.

Type graphs are powerful and flexible, but may in practice grow very large, sometimes to sizes of millions of words [17]. It is unclear whether the possible precision gain as compared to less precise methods outweighs the long analysis times or extravagant space requirements.

2.2 Type jungles

For the purpose of this work, we chose to restrict the full power of type graphs by folding all nodes with the same functor into a single representative (per function symbol appearing in the type graph) for a given structure description. The result is a type jungle: a structure description where all references to $f/n$ lead to a single node. For instance, consider the type jungle corresponding to all lists of lists of
Figure 1: Type graph (a) and type jungle (b) describing all lists of lists of atoms red, white or blue.
red, white or blue given in Figure 1 (b). Note that cons/2 only occurs once, while in the type graph it occurred twice.

The advantage of using type jungles rather than full type graphs is that the sizes of structure descriptors is bounded by the number of symbols in the program (and usually small). The disadvantage is obviously that type jungles are less precise; in particular, any-nodes may cause imprecise summaries. Our use of a pattern domain [3] appears to compensate for this disadvantage.

Finally, we extended our analyzer to use a rich set of leaf nodes to retain precision in some useful cases. Getzinger’s experience with type graph analysis for compilation [6] indicates that leaf nodes that can express properties such as nonvar or atom may be quite important.

2.3 Leaf nodes
Previous implementations of type graphs [9, 19, 6] have provided only the any leaf node, representing any term. We provided a variety of leaf nodes for three reasons: to express the outcome of type tests, to improve robustness of results and to enable optimizations that were otherwise impossible. The prime example of such an optimization is removing type and overflow checks from arithmetic operations.

Structures and atoms. We added leaf nodes struct and atom to describe compound terms with unknown function symbols and unknown subterms, and atoms with unknown names, respectively. This enables us to reason about properties that are unwieldy in a pure type graph setting. For instance, the set of nonvariable terms is described by “struct or atom or int(lo,hi) or float or stream”. For type graphs without these nodes, we would have to enumerate all functors, atoms and other objects, or accept any as the summary. The new elements are quite useful when we consider type tests such as atom/1, var/1 and others, particularly in conjunction with redundant tests, as described in Section 3.

Furthermore, primitives such as functor/3 or atom/chars/2 can create symbols dynamically, which means some programs would unavoidably lose precision if struct and atom were not available.

Arithmetic. The generic arithmetic of Prolog allows programmers to pass fixnums, bignums, floats or evaluable terms to any arithmetic operator. While this flexibility is occasionally useful, most arithmetic does not require the fully general operations. By detecting such situations, the compiler can omit type and overflow checking code. Ideally, a single fixnum or float addition or comparison can be used.

We added the elements int(lo,hi) and float to the domain. The int element represents the integers $x$ such that $lo \leq x \leq hi$, where $lo$ and $hi$ are integers or the special constants $-\infty, +\infty$. The float element represents the set of floating point numbers. We defined the abstract operations to conform with SICStus Prolog v3.

For termination reasons, we introduced a widening on the integer ranges along the lines of Cousot and Cousot [4]. However, we do not use narrowing to subsequently improve precision. Let FIX be the greatest and -FIX the smallest fixnum.
\[(lo, hi) \nabla (ld', hd') = (ld'', hd'')\]

where

\[
ld'' = \begin{cases} 
lo, & \text{when } lo \leq ld' \\
-	ext{FIX}, & \text{when } -\text{FIX} < ld' < lo \\
-\infty, & \text{otherwise}
\end{cases}
\]

\[
hi'' = \begin{cases} 
hi, & \text{when } hd' \leq hi' \\
\text{FIX}, & \text{when } hi < hi' < \text{FIX} \\
+\infty, & \text{otherwise}
\end{cases}
\]

Streams. We saw an opportunity to improve the input/output facilities of Prolog. Prolog handles I/O through 'stream' objects; unfortunately, the protocol is fairly heavy weight: streams may be named and the names passed to read/write operations. Additionally, dynamic type checking is done to ensure proper use.

It was relatively straightforward to add stream elements that tracked the stream type (read/write and file/terminal), but trying to determine stream state, e.g., whether the stream accepts more reads, was harder. The reason is that stream state survives backtracking; thus, the usual analysis algorithms do not work. We did not attempt to track stream state.

SICStus Prolog treats streams as compound terms. We instead assumed that streams are distinct from standard terms. That is, we assumed that compound/1 tests fail for streams, that unifying a stream with a compound terms fails, and so on.

Groundness. We did after some deliberation not add a ground element to the domain. While it could have improved precision in some cases, it makes the algorithms more complex. Furthermore, it may be easier and/or more flexible to perform mode analysis with groundness as a separate analysis.

3 Analysis framework

Our prototype analyzer is based on the static analysis framework proposed by Debray [5]. An overview is shown in Figure 2. We extended this framework in a number of ways.

We used a variation of Cortesi, Le Charlier and Van Hentenryck's PAT(\#) domain [3], which tracks same-value and known-value information precisely through a clause. The analyzer only propagated same/known-value information within a clause.

Our second extension was to use the clause selection rule and cuts of Prolog to improve precision. Usually, analysis frameworks work with Horn clauses rather than general Prolog clauses, and often discard information available from cuts. (One exception is Ref. [12].)

We used a simple extension of the standard analysis: insertion of redundant tests in the code. Consider the following predicate.

\[
p(X) :-
\begin{align*}
& \text{var}(X) \rightarrow q(X) \\
& \text{r}(X)
\end{align*}
\]

5
We can rewrite it into an equivalent predicate that performs a redundant test.

\[
p(X) :-
\begin{align*}
( & \text{var}(X) \to q(X) \\
& ; \text{nonvar}(X), r(X) \\
) .
\end{align*}
\]

Now, it is straightforward to see that \( r/1 \) is called with a nonvariable argument. In a goal \( (A \to B ; C) \), we only negate \( A \) if \( A \) is an 'interesting' primitive. The analyzer considers type tests, arithmetic and some unifications to be interesting. We introduce the appropriate negated versions of tests when required, e.g., \text{nonvar}/1, \text{nonatom}/1, \text{noninteger}/1 and similar.

Compound or user-defined tests are not considered for negation. This simplifies the rewriting, at the possible expense of some precision. For instance, the test

\[
( ( \text{var}(A) ; \text{atom}(A) ) \to B1 ; B2 )
\]

could be rewritten into

\[
( ( \text{var}(A) ; \text{atom}(A) ) \to B1 ; \text{nonvar}(A), \text{nonatom}(A), B2 )
\]

This is not done at present.

The introduced tests are later used to annotate the program, but are themselves deleted from the annotated program.

For some experiments we introduced explicit code to perform indexing [13], in the form of type tests and name tests. This enabled a substantial number of redundant tests to be introduced, with a consequent gain in precision.
4 Empirical results

4.1 Experimental setup

We choose to study the Berkeley benchmarks, since they are well-known and intended to represent a typical Prolog workload\textsuperscript{1}.

Our interest in this evaluation is to consider the usefulness of the information derived by the analyzer. This is quite different from the conventional method of reporting precision per call or success argument. For example, consider the following program:

```prolog
main :- p(a).
p(X) :- q(X).
q(X) :- r(X), r(_,).
r(X) :- X = a.
```

Measuring the (monovariant) precision of call/success arguments will report that p/1 and q/1 are called with atom a as an argument, while r/1 is called with an unknown argument. Thus, we will conclude that precision is quite good: two out of three arguments have a very precise type.

But consider the execution of the program. Type information will not reduce the cost of p/1 calling q/1 or q/1 calling r/1\textsuperscript{2}. Instead, the compiler will attempt to use type information to strength-reduce the unification X = a into something less expensive than a full unification. Unfortunately, our monovariant analyzer will conclude that X is unknown and so will be unable to optimize the unification. Type information enabled no optimizations in this code, an unexpectedly poor result when we consider only the call/success precision.

For this reason, we choose to evaluate the type jungle domain by looking at how individual primitive operations could be optimized by considering type information. (A companion paper \textsuperscript{[1]} measures precision more conventionally, and also shows that analysis times are very good.)

Measurements were done as follows.

- The benchmark programs were analyzed and the annotated programs dumped on disk.
- A second pass traverses the annotated programs and classifies each primitive operation according to a number of optimization patterns peculiar to that primitive.

We concentrated on a number of the most common primitive operations: unification and arithmetic. We also looked at how a complex meta predicates such as functor/3 can be strength-reduced. We say that an operation is reducible if it can be entirely done at compile-time or if it can be replaced by a less expensive operation given the available type information.

\textsuperscript{1}The Berkeley benchmarks were used in the Aquarius Prolog project for this purpose.

\textsuperscript{2}However, uninitialized variable analysis may in general reveal that output arguments need not be saved in the clause environment. This reduces the cost of procedure calls by reducing the size of the environment. We will not further consider this topic in this paper since there are simple and efficient analyses that detect uninitialized arguments, e.g. \textsuperscript{[15]}. 

\textsuperscript{7}
4.2 Measurements

We analyzed the Berkeley benchmark suite and classified the primitive operations therein.

Primitive operations were counted as follows: the program was subjected to indexing expressed in Prolog (which introduces type tests, unifications and functor tests), and all unifications normalized (i.e., broken up into sequences of simple unifications). Prior to type analysis, an uninitialized variable analysis [15] is performed, which may introduce several versions of each source-level predicate, and which also deletes unused predicates. After analysis, annotation deletes primitives that always fail as well as code depending on that primitive; failed and unexecuted primitives are not counted.

As an example.

\[
\text{main :- ( } X = f(a) ; X = g \text{ ), p(X).}
\]
\[
p(f(a)).
\]
\[
p(f(b)).
\]
\[
p(g). 
\]
\[
p(h). 
\]

Prior to analysis, the program is converted into the form where indexing is explicit.

\[
\text{main :- ( } X = f(Y) \text{, } Y = a \text{ ; } X = g \text{ ), p(X).}
\]
\[
p(A) :-
\]
\[
\begin{align*}
\text{ ( } \text{var}(A) \rightarrow & \nonumber \\
\text{ ( } A = f(B), B = a & \\
\text{ ; } A = g(B), B = b & \\
\text{ ; } A = g & \\
\text{ ; } A = h & \\
\text{ ) } & \\
\text{; atom}(A) \rightarrow & \\
\text{ ( } A = g \rightarrow \text{true } & \\
\text{; } A = h \rightarrow \text{true } & \\
\text{ ) } & \\
\text{; struct}(A) \rightarrow & \\
\text{ ( } \text{functor}(A,f,1) \rightarrow & \\
\text{ A = f(B), B = a } & \\
\text{; functor}(A,g,1) \rightarrow & \\
\text{ A = g(B), B = b } & \\
\text{ ) } & \\
\text{). } & 
\end{align*}
\]

The analyzer either duplicates a clause body at all occurrences or constructs a new predicate consisting of that clause only\(^3\). At present, a new predicate is constructed if the clause is “too large” and would otherwise be duplicated. In our example, no new predicates are formed.

Analysis will show that p/1 is called with f/1 or g/0 as toplevel functor, which the analyzer will simplify into the following.

\(^3\)The analyzer takes proper care to handle cuts correctly.

8
main :- ( X = f(Y), Y = a ; X = g ), p(X).
p(A) :-
    ( fail ->
    ; atom(A) ->
        ( A = g -> true  % always
        ; fail
    )
    ; struct(A) ->  % always
        ( functor(A,f,1) ->  % always
            A = f(B), B = a  % always
        ; fail
    )
).

Predicate p/1 would be counted as three ‘always’ unifications, two type tests and one functor/3 call. Thus, a large number of primitive operations that appear in the source code will not be counted if the analyzer can prove they will never be executed. Furthermore, indexing introduces new primitive operations in the code, which will be counted separately at their different occurrences.

4.3 Unifications

We begin by studying unifications. The concrete numbers are shown in Table 1.

For unifications, we used the following classifications.

Assign. One of the arguments is uninitialized, so the unification is an assignment or a store.

Always. Both arguments have the same name (functor or atom), so unification always succeeds.

Compare. Both arguments have the same type (structure or constant). Unification reduces to a comparison (for atoms or fixnums) or load+compare (for structures and floats).

Test. Both arguments are bound, but do not have precisely the same type. (E.g., one type may be a subset of the other.) Unification reduces to some tests.

Full. At least one of the arguments is unknown while the other is bound or unknown. Full unification is required.

General. General unifications appear in the interval of 0.038 of the total number of unifications, with a median of 0.10 and an average of 0.08. The type analysis is reasonably successful in strength-reducing general unification, but usually not entirely successful. Some programs (e.g., crypt, fast_mmu or flatten) have a considerable fraction of general unifications while the “inner loop” benchmarks (nreverse, queens_8 and similar) entirely dispense with general unification.

Assign. Assign unifications are the most common, appearing from 0.08 to 1.00 of the total number of unifications. The median is 0.500, and the average is 0.56 of the total. Thus, uninitialized unifications may be the most important category to track. Doing so will enable a compiler to reduce the total code volume substantially since uninitialized unifications yield very compact native code.
Always. Always unifications are relatively common: they range from 0.0-0.40 of the total number of unifications while the average is 0.13, while the median is 0.11. Why do always unifications appear? There are three main reasons that we have observed.

The first situation appears in switches. Consider the following example.

\[
\text{append(A,B,C) :-}
\text{( A = [ ] -> ...}
\text{ ; A = [X1|Xs] -> ...)}
\]

If we know that A is a list, then the analyzer will see that the second unification always succeeds. While this is common in list iteration, we have also observed it in programs such as chat.parser. There, a switch over numerous alternatives will end with the last unification always succeeding.

The second source of always unifications is when abstract datastructures are used. If an argument is always a three-tuple and is always unified with a three-tuple, that unification always succeeds. This situation occurs in nand.

A third source of always-unifications is indexing. Indexing may insert extra unifications to select among clauses. This can make subsequent unifications redundant. For example:

\[
\text{append(A,B,C) :-}
\text{( var(A) -> ...}
\text{ ; atom(A) -> A = [], B = C}
\text{ ; ...)}
\]

Assume that A is known to be a cons or the empty list. After the atom/1 test, A must be the empty list. Thus, the unification A = [] becomes an always unification.

Compare. Compare unifications are quite common. They are 0.0-0.413 of the total unifications, with a median of 0.16 and an average of 0.08.

Test. Test unifications appear more rarely. They range from 0.0-0.13 of the total, with a median of 0.06 and an average of 0.05. This is somewhat surprising; we would have thought compare unifications to be less common than test unifications prior to these measurements.

Consider an example program.

\[
\text{append(A,B,C) :-}
\text{( A = [ ] -> ...}
\text{ ; A = [X1|Xs] -> ...)}
\]

If we know that A is a list, then the first unification (with the empty list) is a test-unification, since A at that point can be either an atom or a cons cell. (Note that the unification still can be implemented very efficiently, by comparing A with a constant representing the empty list.)

---

4This is due to a redundant test saying A is never the empty list in the second branch, in fact.
<table>
<thead>
<tr>
<th>Program</th>
<th>Always</th>
<th>Assign</th>
<th>Test</th>
<th>Compare</th>
<th>Full</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>66</td>
<td>665</td>
<td>1</td>
<td>199</td>
<td>4</td>
<td>935</td>
</tr>
<tr>
<td>browse</td>
<td>6</td>
<td>35</td>
<td>-</td>
<td>18</td>
<td>21</td>
<td>80</td>
</tr>
<tr>
<td>chat-parser</td>
<td>340</td>
<td>1360</td>
<td>151</td>
<td>515</td>
<td>108</td>
<td>2474</td>
</tr>
<tr>
<td>crypt</td>
<td>3</td>
<td>8</td>
<td>-</td>
<td>43</td>
<td>50</td>
<td>104</td>
</tr>
<tr>
<td>divide10</td>
<td>-</td>
<td>11</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>fast_mmu</td>
<td>3</td>
<td>65</td>
<td>1</td>
<td>7</td>
<td>33</td>
<td>109</td>
</tr>
<tr>
<td>flatten</td>
<td>21</td>
<td>75</td>
<td>-</td>
<td>13</td>
<td>47</td>
<td>156</td>
</tr>
<tr>
<td>log10</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>11</td>
<td>140</td>
<td>1</td>
<td>9</td>
<td>14</td>
<td>175</td>
</tr>
<tr>
<td>mu</td>
<td>4</td>
<td>47</td>
<td>9</td>
<td>18</td>
<td>33</td>
<td>111</td>
</tr>
<tr>
<td>nand</td>
<td>114</td>
<td>193</td>
<td>63</td>
<td>102</td>
<td>16</td>
<td>488</td>
</tr>
<tr>
<td>nreverse</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>ops8</td>
<td>-</td>
<td>14</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>18</td>
</tr>
<tr>
<td>prover</td>
<td>7</td>
<td>226</td>
<td>-</td>
<td>18</td>
<td>27</td>
<td>278</td>
</tr>
<tr>
<td>qsort</td>
<td>4</td>
<td>7</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>14</td>
</tr>
<tr>
<td>queens_8</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>19</td>
</tr>
<tr>
<td>query</td>
<td>51</td>
<td>83</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>208</td>
</tr>
<tr>
<td>serialise</td>
<td>11</td>
<td>19</td>
<td>-</td>
<td>11</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>simple</td>
<td>85</td>
<td>184</td>
<td>34</td>
<td>29</td>
<td>64</td>
<td>396</td>
</tr>
<tr>
<td>tak</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>times10</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>unify</td>
<td>10</td>
<td>73</td>
<td>7</td>
<td>15</td>
<td>20</td>
<td>125</td>
</tr>
<tr>
<td>zebra</td>
<td>11</td>
<td>11</td>
<td>-</td>
<td>17</td>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>734</strong></td>
<td><strong>3247</strong></td>
<td><strong>292</strong></td>
<td><strong>1037</strong></td>
<td><strong>465</strong></td>
<td><strong>5815</strong></td>
</tr>
<tr>
<td><strong>Ratio</strong></td>
<td>0.13</td>
<td>0.56</td>
<td>0.05</td>
<td>0.18</td>
<td>0.08</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 1: Unification categories. **Always.** Always succeeds (e.g., select components of compound term). **Assign.** One or both arguments is uninitialized. **Test.** Unification reduces to tag-test followed by compare. (Both arguments are bound.) **Compare.** Unification reduces to direct compare of constants or functors. **Full.** One or both arguments has unknown type.
**Final notes.** A subset of the benchmark programs are versions of the archetypal derive program supplied with different inputs. It is interesting to note that the compiler can achieve a form of partial evaluation for some versions.

For example, in `divide10`, the program looks like:

```plaintext
main :- divide10.

divide10 :- d(((((x/x)/x)/x)/x)/x)/x)/x)/x)/x)/x,x,,).

d(U*V,X,DU+DV) :- !,
    d(U,X,DU),
    d(V,X,DV).

d(U-V,X,DU-DV) :- !,
    d(U,X,DU),
    d(V,X,DV).

d(U*V,X,DU*V+U*DV) :- !,
    d(U,X,DU),
    d(V,X,DV).

d(U/V,X,(DU*V-U*DV)/(\^ (V,2))) :- !,
    d(U,X,DU),
    d(V,X,DV).

d(\^ (U,N),X,DU*N*(\^ (U,N1))) :- !,
    integer(N),
    N1 is N-1,
    d(U,X,DU).

d(-U,X,-DU) :- !,
    d(U,X,DU).

d(exp(U),X,exp(U)*DU) :- !,
    d(U,X,DU).

d(log(U),X,DU/U) :- !,
    d(U,X,DU).

d(X,X,1) :- !.

d(_,_,0).
```

The analyzer derives that the first argument of `d/3` only may be \(/2 or x/0, while the second argument is x/0 and the third argument is uninitialized. This enables the following simplification:

```plaintext
main :- divide10.

divide10 :- d(((((x/x)/x)/x)/x)/x)/x)/x)/x)/x)/x,x,,).

d(U/V,X,(DU*V-U*DV)/(\^ (V,2))) :- !,
    d(U,X,DU),
    d(V,X,DV).

d(_,_,1).
```

The new `d/3` can be compiled much more efficiently than the old version: if the first argument is a structure, jump to clause one; otherwise, jump to clause two.
arith_op(X,0p,Y,Z) :-
( fixnum(X) ->
  ( fixnum(Y) ->
    (Z0,0v) := perform(X,0p,Y),
    (0v == overflow ->
      Z := fix_to_big(Z0,0v)
    ; Z := Z0
    )
    ; float(Y) ->
      ...
    ; bignum(Y) ->
      ...
    ; Y0 := eval(Y),
      arith_op(X,0p,Y0,Z)
  )
; float(X) ->
  ...
; bignum(X) ->
  ...
; X0 := eval(X),
  arith_op(X0,0p,Y,Z)
).

Figure 3: Structure of an arithmetic operation. Implicit type casts are made, incoming structure arguments are evaluated into numbers and fixnum arithmetic must check for overflows on the result.

Note that clause two need not check that argument one is x/0; that is implied by the type information.

This optimization is not possible merely by using Hindley-Milner type inference, since it takes calling context into account. HM type inference, in contrast, is a bottom-up process.

4.4 Arithmetic operations and comparisons

We next turn to the use of numeric information. It can be used to strength-reduce arithmetic operations and to simplify comparison operations.

Arithmetic operations are classified in two ways. A template arithmetic operation is shown in Figure 3. Type information may allow us to omit some type tests and possibly overflow tests.

Direct. Both arguments are fixnums and the result is a fixnum. In this case, no type checking or overflow checking is needed.

D_ov. Both arguments are fixnums, but result may be bignum. In this case, no type checking is required, but an overflow test is needed.

Check. Both arguments are numbers, but the version of the predicate to be used must be selected (e.g., unary minus must select whether float, fixnum or bignum minus should be used).
**C.conv.** Both arguments are numbers, but type conversions (e.g., bignum to float or fixnum to bignum) may be required.

Note that other cases are possible, though they never appear in the benchmarks considered here. For example, general arithmetic evaluation was never needed in these benchmarks.

Type analysis allows the compiler to omit useless type tests. In the following, we consider the common binary operations, such as addition or subtraction.

If both arguments have known type, then a specialized version of the operation can be used directly. This saves two tests and considerable code. Furthermore, type conversions can be done immediately.

If both arguments are known to be integers, then a reduced version of the operation can be emitted.

If the result is known to be a fixnum, then overflow checking is not needed.

Other binary operations may have a slightly different structure, e.g., bitshifts or modulo operations. We will not consider them further.

For unary operations such as sin/1, cos/1 and so on, it may be possible to delete type checks and type casts on the argument.

Arithmetic comparisons are essentially simpler cases of arithmetic operations. We only consider what the argument types were in this case.

**Arithmetic operations.** We now consider the classification of arithmetic operations. First, as we can see, arithmetic is substantially less common than unifications. Only 118 arithmetic operations were used in the Berkeley benchmark suite.

We also see that most arithmetic falls into the category of requiring type checks and possible type casts (64% of the total). However, we also see that a substantial number of operations are direct arithmetic operations (35%) and that the majority of those do not even require overflow checking after performing the operation (22%). This is encouraging.

A closer look at the results reveals that many arithmetic operations increment or decrement a counter, which is often bounded by some fixnum. Such arithmetic can often be seen to be entirely fixnum arithmetic.

Other examples are not so easy. Take the crypt benchmark; here there are predicates along the lines of:

odd(1).
odd(3).
odd(5).
odd(7).
odd(9).

Our widening will say that odd/l has an argument in the range (1, FIX), not in the range (1,9). The more precise interval could be arrived at by using narrowing, which is not done at present. (In fact, reordering the clauses of odd/l could yield an even less precise (-FIX,FIX) result in this instance.)

This imprecision will propagate into the rest of crypt. For example:
main :-
odd(A), even(B), even(C),
even(E),
mult([C,B,A], E, [I,H,G,F|X]),
...
...
mult(AL, D, BL) :- mult(AL, D, 0, BL).

mult([A|AL], D, Carry, [B|BL]) :-
X is A*D+Carry,
B is X mod 10,
NewCarry is X // 10,
mult(AL, D, NewCarry, BL).
mult([], _, Carry, [C,Cend]) :-
C is Carry mod 10,
Cend is Carry // 10.

In our implementation, we cannot rule out that A*D might overflow since the analyzer believes A and D can be any fixnum (due to our imprecise widening). An analyzer using narrowing might be able to eliminate the overflow check⁵ by observing that A and D are in fact small fixnums.

Comparisons. We next turn to arithmetic comparisons. Again, there are few such operations in the benchmarks as compared to unifications: only 150 of them. Considering Table 3 we see that 29% of all comparisons are two fixnums being compared; that 56% of all comparisons compare two integers, and that 85% of all comparisons compare two numbers. The nand benchmark yields almost all of the unknown compares, which is due to a loss of precision when numbers are stored in recursive datastructures.

Final notes. The arithmetic domain enables some interesting simplifications. For example, in a version of the program read (not used as a benchmark), the following code occurs. Predicate read_tokens/3 is the sole caller of read_special/3.

read_tokens(C1, Dict, Tokens) :-
( ... ; C1 < 127 --> special character
read_special(C1, Dict, Tokens)
)...
...
read_special(247, Dict, Tokens) :- (division sign)
read_symbol(247, Dict, Tokens).
read_special(215, Dict, Tokens) :- x (multiplication sign)

⁵In this case, the analyzer applies the widening even though it is not strictly necessary. Since odd/1 is non-recursive, a widening might not be needed.
Table 2: Arithmetic operations. **Direct.** Both args have same type; no overflow possible. **D.ov.** As Direct, but overflow possible. **Check.** Type checking needed but no type casting. (Arguments are numbers.) **C.conv.** Type checking needed, maybe type casting. (Arguments are numbers.)
<table>
<thead>
<tr>
<th>Program</th>
<th>Arg. types</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>browse</td>
<td>(int,fix)</td>
<td>1</td>
</tr>
<tr>
<td>chat_parser</td>
<td>(fix,fix)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(num,fix)</td>
<td>2</td>
</tr>
<tr>
<td>crypt</td>
<td>(num,fix)</td>
<td>28</td>
</tr>
<tr>
<td>divide10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fast_num</td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(int,int)</td>
<td>9</td>
</tr>
<tr>
<td>flatten</td>
<td>(fix,fix)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>(int,int)</td>
<td>3</td>
</tr>
<tr>
<td>log10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>meta_qsort</td>
<td>(any,any)</td>
<td>1</td>
</tr>
<tr>
<td>mu</td>
<td>(fix,fix)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>(num,fix)</td>
<td>4</td>
</tr>
<tr>
<td>nand</td>
<td>(any,any)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(any,fix)</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(int,any)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(int,fix)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(int,int)</td>
<td>6</td>
</tr>
<tr>
<td>nreverse</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ops8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prover</td>
<td>(num,fix)</td>
<td>10</td>
</tr>
<tr>
<td>qsort</td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td>queens_8</td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(fix,int)</td>
<td>2</td>
</tr>
<tr>
<td>query</td>
<td>(int,int)</td>
<td>2</td>
</tr>
<tr>
<td>serialise</td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td>simple</td>
<td>(fix,fix)</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>(int,fix)</td>
<td>10</td>
</tr>
<tr>
<td>tak</td>
<td>(fix,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(int,fix)</td>
<td>1</td>
</tr>
<tr>
<td>times10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>unify</td>
<td>(fix,fix)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>(int,fix)</td>
<td>4</td>
</tr>
<tr>
<td>zebra</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>

Table 3: Arithmetic comparisons. Types of compared arguments shown along with number of compares fitting that description.
functor(Term,F,N) :-
  ( var(Term) ->
    atomic(F),
    fixnum(N),
  ( N ::= 0 ->
    Term = F
  ; N > 0,
    N <<= max_arity,
    S := build_struct(F,N),
    Term = S)
  ; atomic(Term) ->
    N = 0, F = Term
  ; struct(Term) ->
    (Fn,Ar) := function_symbol_arity(Term),
    F = Fn,
    N = Ar)

Figure 4: Pseudo code definition of functor/3 (omits raising exceptions).

read_symbol(215, Dict, Tokens).
...

Evidently, the programmer forgot that C1 is never more than 126 when read_special/3 is entered. The analyzer did not forget, and automatically eliminated the redundant clauses of read_special/3.

4.5 Functor/3

Finally, we shall consider the use of functor/3 in the benchmark programs. Functor/3 is introduced by indexing and also appears 'naturally' in the code. The measurements are shown in Table 4.

Functor/3 is a bimodal predicate: used in one mode, it builds a structure; used in another, it checks the shape of an existing structure. It is defined in pseudo code in Figure 4.

As can be seen from that figure, the compiler can strength-reduce the primitive considerably given proper type information.

- If Term is uninitialized, functor/3 reduces to consistency checks on F and N, followed by building a structure and assigning Term that value. (The consistency checks might be reducible in turn.)

- If functor of Term is known, then functor/3 reduces to two unifications (which can be strength-reduced in turn).

- If functor of Term is known to be an atomic or a structure, then functor/3 essentially reduces into two unifications (along with loading the functor if needed).
• If Term is bound, the variable test can be deleted.

• Various consistency checks can fail or be reduced at compile-time: N is not a proper fixnum in the allowed range, F is not an atomic object that can match Term.

A total of 187 calls to functor/3 appear in the Berkeley benchmarks, including those introduced by the compiler for indexing purposes. Out of those, only 7 had an unknown first argument, while 6 had an uninitialized first argument.

We classify the rest of the functor/3 calls as follows: if the functor and arity are known, and the term functor is known, then the operation is known (to succeed). If the term functor is not known, but functor and arity are, then the operation is a test. If functor and arity are uninitialized, the operation is a load. Otherwise, it is other.

We see that 15 calls are known, 143 calls are tests, 13 calls are loads, and only 4 calls are other. Thus, the analyzer can resolve the vast majority of calls to functor/3 (91%) as simple tests or loads, or as no-ops. However, it should again be noted that many of these functor/3 calls are introduced by indexing.

5 Using type information in an optimizing compiler

We note that type information mainly enables the compiler to delete tests that will always or never hold. The main effect of doing so is that code size is reduced, while the number of instructions to be executed is less affected. Thus, we may expect the sizes of binaries to shrink more than performance is improved.

There are two second-order effects that may improve performance further than the apparent reduction of executed instructions.

• Some compilers emit primitive operations as out-of-line procedures to avoid code size blowups. For example, most Prolog compilers have a common-case fixnum path for arithmetic and call an out-of-line procedure otherwise. SICStus Prolog uses threaded code extensively in its runtime system to avoid excessive code size.

By reducing code size strongly, the threaded-code overhead can be eliminated by the compiler. This in turn enables a multitude of optimizations, such as instruction scheduling, improved register allocation and improve classical dataflow analyses.

• In practice, a compiler is a soft realtime program: the user is not prepared to wait ‘too long’ for an answer. Emitting less code means the compiler can spend longer time on the remaining code, including the use of code duplicating transformations.

We have investigated none of the above optimizations at this time. Nevertheless, it is clear that type information can aid a sophisticated compiler. Unsophisticated compilers, on the other hand, may not see great gains by putting their faith solely to type analysis.

6 Related work

Several analyses for deriving structure information for Prolog or Lisp have been proposed previously [9, 8, 19, 22, 6, 22]. Most have concentrated on defining or
<table>
<thead>
<tr>
<th>Program</th>
<th>Category</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>(bound,uninit,uninit)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(struct,known,known)</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>(uninit,atom,fix)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(unknown,uninit,uninit)</td>
<td>1</td>
</tr>
<tr>
<td>browse</td>
<td>(unknown, known, uninit)</td>
<td>2</td>
</tr>
<tr>
<td>chat-parser</td>
<td>(bound,known,known)</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>(struct,known,known)</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>6</td>
</tr>
<tr>
<td>flatten</td>
<td>(bound,uninit,uninit)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(bound,known,known)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(struct,uninit,uninit)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(uninit,atom,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(known,uninit,uninit)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>1</td>
</tr>
<tr>
<td>metasort</td>
<td>(bound,known,known)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(struct,known,known)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>1</td>
</tr>
<tr>
<td>mu</td>
<td>(bound,known,known)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>1</td>
</tr>
<tr>
<td>prover</td>
<td>(struct,known,known)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>1</td>
</tr>
<tr>
<td>serialise</td>
<td>(known,known,known)</td>
<td>1</td>
</tr>
<tr>
<td>simple</td>
<td>(bound,uninit,uninit)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>(struct,atom,fix)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(struct,uninit,uninit)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(struct,known,known)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>(uninit,atom,fix)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(uninit,unknown,unknown)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(unknown,uninit,uninit)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>(known,known,known)</td>
<td>4</td>
</tr>
<tr>
<td>unify</td>
<td>(bound,unknown,unknown)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(struct,uninit,uninit)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>(struct,unknown,unknown)</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>187</td>
</tr>
</tbody>
</table>

Table 4: Summary of use of functor(Term,Functor,Arity), given as a triple (T,F,A) signifying respective types. Benchmarks not shown did not call functor/3.
efficiently implementing the analysis operations and paid relatively little attention to the outcome of the analysis. Van Hentenryck et al provide call/success pattern precision for a number of benchmarks [19], but do not relate their results to compiler optimization.

To our knowledge, only Getzinger [6] has studied the usefulness of type graphs for compilation. He implemented a version of the type graphs of Janssens and Bruynooghe [9] in the context of Aquarius Prolog, and found (a) that the analyzer was too inefficient for general use (e.g., requiring several hours to analyze some small benchmarks and not terminating for several other programs) and (b) that the analysis results only provided a 1% improvement on the results of less costly domains. In part, this was due to lacking struct, atom and similar leaf nodes, and thus being unable to express the results of type tests.

The Python CMU Common Lisp compiler [16] uses type propagation to optimize the generic arithmetic of Common Lisp. In particular, Common Lisp has an extensive system of type declarations which Python can use to strength-reduce arithmetic. It is unclear to which extent Python is capable of strength-reducing arithmetic without declarations.

Bigot and Debray [1] consider how to optimize the use of untagged integers and floating point values in a logic programming context. They do not consider bignum arithmetic and concentrate on passing untagged accumulators. Their method is quite effective on numeric (mainly scalar) kernels, but relies on user declarations to see, e.g., that all elements in a list are floating point numbers.

We use redundant tests to propagate information between proof tree branches while retaining a simple analysis framework. This approach is quite limited in its scope and it might be useful to consider approaches such as the cardinality analysis of Braems et al [2]. For instance, propagating assertions $X < Y$ through the program is orthogonal to our range arithmetic and may provide opportunities for further optimizations.

7 Conclusion and future work

We proposed a simple structure analysis domain, type jungles. The domain can be viewed as a restriction of type graphs whereafter arithmetic ranges and imprecise leaf nodes are added on that domain.

We assessed the gains from type jungle analysis by classifying how primitive operations would be compiled given the available type information. By looking at how the type information is used, rather than call/success pattern precision, we can assess the usefulness of type information more easily. High precision due to a more complex structure analysis may not yield better code than a quite simple analysis, if the precise information does not enable more optimizations to be done.

We found that only 8% of all unifications (ranging over a set of 23 benchmarks) needed general unification, while fully 69% always succeeded or required only term building.

Furthermore, we found that arithmetic operations could be substantially simplified by static analysis, even when fixnums can seamlessly overflow into bignums. Even so, 35% of all arithmetic operations did not require type testing (not even to determine whether the input was a fixnum or a bignum). General arithmetic evaluation was never needed; we found that arguments were always numbers.

Arithmetic comparisons were more varied. Even so, 89% of all comparisons were
of numbers (and 29% compared two fixnums). Some interesting optimizations are possible by tracking numeric ranges.

Finally, we considered the use of functor/3 in the benchmarks, and found that our structure analysis could resolve 91% of the functor/3 calls into tests, loads or no-ops.

Thus, we conclude that structure analysis based on type jungles, combined with uninitialized variable analysis, can strongly improve the

7.1 Future work

Four main lines of work present themselves. First, we would like to compare type jungles with other domains using the same measurement techniques. How do type jungles compare to full type graphs or to a less precise structure analysis?

Second, various extensions to the current abstract domain suggest themselves. There is some evidence that narrowing would be useful to further optimize arithmetic. Groundness can be quite useful when structure information is lost.

Third, improvements to the analysis framework may yield further benefits. Here, we consider mainly polyvariance. A polyvariant structure analyzer should balance the number of versions generated (including code size and analysis time) against analysis precision, and must judiciously choose which versions to merge. Polyvariance can also help in avoiding precision loss by keeping imprecise call sites apart from precise ones.

Finally, the results should be verified in an optimizing compiler and be extended to a larger set of benchmarks. Structure analysis of very large programs is still a fairly open topic, though some work has been done [17]. An optimizing compiler can change these results in two directions: it can dampen their impact, by compiling imprecise cases well as compared to precise cases; or it can enhance our results by introducing new optimizations since so much useless code can be removed. At this time, we are mainly interested in the latter.

7.2 Acknowledgements

My thanks to Per Mildner for his comments on this work.

References


