Property Inference for Maple: An Application of Abstract Interpretation

Jacques Carette\textsuperscript{1}  Stephen Forrest\textsuperscript{1}

\textsuperscript{1}McMaster University
1280 Main St. W., Hamilton, ON, Canada

Calculemus 2007
Outline

1. Introduction
   ▶ The Problem
   ▶ Static Analysis
   ▶ Properties of Interest

2. Tools
   ▶ Abstract Interpretation
   ▶ Constraints
   ▶ Recurrences

3. Results

4. Methodology and Design

5. Conclusion
Consider the following basic features of Maple

1. Imperative and Functional style (including higher-order functions)
2. Dynamic typing (one can create a ‘halts’ type!)
3. Polymorphism: ad hoc, parametric, even intensional
4. Reflection and Reification
5. Dependent “types”
6. First-class “types”
The Problem: Understanding Maple code

Consider the following basic features of Maple
1. Imperative and Functional style (including higher-order functions)
2. Dynamic typing (one can create a ‘halts’ type!)
3. Polymorphism: ad hoc, parametric, even intensional
4. Reflection and Reification
5. Dependent “types”
6. First-class “types”

Solutions?
1. No traditional type system can handle all these.
2. Type-and-effect systems help a little.
3. Axiom/Aldor/Focal-style types do not capture enough [but contain many useful ideas]
4. Haskell type classes also fall short [but are another source of ideas]
We wish to “infer”

1. as much as possible (precision)
2. without executing the code, (offline)
3. while terminating,
4. and remaining sound.
We wish to “infer”

1. as much as possible (precision)
2. without executing the code, (offline)
3. while terminating,
4. and remaining sound.

(Classical) Solution: Look for approximatable properties of

1. Our values
2. Our programs
Properties of Interest

▶ Values
  ▶ type! Approximation implemented: set of possible *surface types*
  ▶ Sequence length. Approx: length range (like 5 . . . 12) \( \mathbb{I}(\mathbb{N}) \)
  ▶ Length (nops – number of operands). Approx: range \( \mathbb{I}(\mathbb{N}) \)
  ▶ “Dependence”. Approx: set of (possible) dependencies
Properties of Interest

- **Values**
  - type! Approximation implemented: set of possible *surface types*
  - Sequence length. Approx: length range (like 5 \ldots 12) $\mathbb{N}$
  - Length (nops – number of operands). Approx: range $\mathbb{N}$
  - “Dependence”. Approx: set of (possible) dependencies

- **Programs**
  - Variable reads. Approx: range $\mathbb{N}$
  - Variable writes. Approx: range $\mathbb{N}$
  - **Reaching definition** (current assignment). Approx: sets of (variable, program location).
Properties – Applications

- Variable with no type means code contains a mistake.
- Variable read 0 times can be eliminated (and associated pure computations).
- Variable read once can have their definition inlined.
- Sequence length = 1..1 means the variable is not a sequence!
- Dependence = ∅ means value is “pure”
- Dependence = \{Digits\} means ??
Consider:

```plaintext
SimpleInt := proc(x, n) local c, i;
    if n=-1 then log(x)
    else c := 1/(n+1);
        for i from 1 to n+1 do c := c*x end do;
    c;
end if;
end proc:
```

- Obfuscated way of computing $\int x^n \, dx$ for $n \geq -1$.
- Though `SimpleInt` contains no explicit types, a “successful” run clearly imposes constraints upon the input. Using these, want to deduce (amongst other facts) that the result is an “expression” and $n \in \mathbb{Z}$.
- We also want to be as precise as possible in estimating the behaviour of the assignment in the loop context.
(Classical) Abstract Interpretation

- General Methodology particularly well suited to program analyses.
- Let $C$ and $A$ be complete lattices, of Concrete and Abstract values.
- A pair $\alpha : C \to A$ (abstraction) and $\gamma : A \to C$ (concretization) of monotonic, continuous lattice functions.
- Pair is a Galois connection if $\forall c. c \sqsubseteq_C \gamma(\alpha(c))$ and $\forall a. \alpha(\gamma(a)) \sqsubseteq_A a$.
- Given $f : C \to C$ and $g : A \to A$, $g$ is a sound approximation of $f$ iff if
  $$\forall c. \alpha(f(c)) \sqsubseteq_A g(\alpha(c))$$
- To all program points, transfer function $f_{ij} : C \to C$
(Semi-classical) Constraints

- (Like modern type systems): first pass just collects constraints.
- Rich constraint language:
  - lattice values (ex: $0 \ldots 2, 1 \ldots 1$, sets of types, etc),
  - lattice operators (ex: $\land, \lor, \sqsubseteq$),
  - logical operators (and only for now)
  - abstract interpretation operations, (like lifted $\oplus$ on intervals, as well as widenings)
  - recurrences ($v_3(n + 1) = v_3(n) \oplus 1 \ldots 1$)
- Second pass: solve constraints (being formalized)
Recurrences (1) encode the iteration of any transfer function over any lattice.

Loops (and recursion) induce (2) recurrences
- sometimes trivial,
- often simple

Monoids:
- Set of iterates is a monoid in the space of transfer functions
- Most of our lattices have monoid structures ($\oplus$ for $\mathbb{I}(\mathbb{N})$, $\cup$ for sets)

Formalize in constraint language
- add a few extra symbols to language
- add constraint generators
- add a new “iteration counter” variable

Ad hoc solvers (for now)
A Worked Example

IsPrime := proc(n::integer)
local S, result;
S := numtheory:-factorset(n);
if nops(S) > 1 then
    result := (false, S);
else
    result := true;
end if;
return(result);
end proc:

Maple procedure which returns a boolean result indicating whether the argument is prime. In the event it is composite, the set of factors is also returned as the second element of an expression sequence.
Each of the indicated locations is a program point with associated properties, which is the basis for the constraint system. The constraints are derived from Maple’s operational semantics.
A Worked Example (cont)

Constraints for the Expression Sequence Length analysis:

\[ V_2 = V_3, V_4 = \text{ProcInitVal}(0), \]
\[ V_4 \leq \text{ES}(1 \ldots 1), V_3 \leq \text{ES}(1 \ldots 1), V_6 \leq \text{ES}(1 \ldots 1), \]
\[ V_7 = V_8, V_8 = \text{ES}(1 \ldots 1) \oplus V_9, V_{10} = \text{ES}(1 \ldots 1), \]
\[ V_{11} = V_7 \lor V_{10} \]

- \( \text{ES}(x \ldots y) \) denotes a expression size approximation,
- \( \text{ProcInitVal}(u) \) denotes the initial size of a parameter,
- \( v_n \) denotes a variable corresponding to the value of the program point \( n \).

\( \lor \) and \( \oplus \) are the join and summation operations in \( \mathbb{I}(\mathbb{N}) \).
Another Example

\[
p := \text{proc}(u) \text{ local } x, y;
\quad x := 2, 3, 4, 5; \; y := 1;
\quad \text{while } y < 10 \text{ do } x := (x, 1); \; y := y \times u; \; \text{end do};
\quad (x, y);
\text{end proc}:
\]

The two assignments within the while-loop body induce the following recurrences in the Expression Sequence Length Analysis:

\[
\text{LoopIC}(x) = \text{ES}(4 \ldots 4) \quad \text{LoopIC}(y) = \text{ES}(1 \ldots 1)
\]

\[
\text{LoopStepFinal}(x) = \text{LoopStepInit}(x) \oplus \text{ES}(1 \ldots 1) \quad \text{LoopStepFinal}(y) = \text{ES}(1 \ldots \infty) \land T_{\text{PROD}}(\text{LoopStepInit}(x), \text{ES}(1 \ldots 1))
\]

- \(T_{\text{PROD}}(a, b)\) is the induced transfer function (on \(I(\mathbb{N})\)) for a product.
- The meet with \(\text{ES}(1 \ldots \infty)\) is needed because Maple does not allow a product of length 0.
Another Example (cont)

When we solve these recurrences, we obtain the results:

\[
\text{LoopFinalVal}(x) = \text{ES}(1 \ldots 1) \otimes \text{ESize} (\text{NumLoopSteps})
\]

\[
\text{LoopFinalVal}(y) = \text{ES}(1 \ldots 1)
\]

- \text{NumLoopSteps} is a symbolic quantity which we may be able to determine by context, or by performing other analyses. If so, we can obtain an exact estimate on the size of \(x\).

- Maple semantics implies that

\[
\text{T}_{\text{PROD}}(\text{ES}(1 \ldots 1), \text{ES}(1 \ldots 1)) = \text{ES}(1 \ldots 1)
\]

so our recurrence for \(y\) has an exact solution.
Our approach can sometimes detect programming errors.

```plaintext
p := proc () local L;
    L := [1, 2];
    sin(L);
end proc:
```

- As \( L \) is a list, it is unacceptable as an argument to \( \sin \).
- Our Surface Type Analysis recognizes this by estimating the set of surface types which \( L \) may match as \( \emptyset \).
- Generally, estimates corresponding to \( \bot \) in our lattice signify programming errors.
### Results (overall)

<table>
<thead>
<tr>
<th>Expression Sequence Length</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local with estimate $\neq [0 \ldots \infty]$</td>
<td>862</td>
</tr>
<tr>
<td>Local with finite upper bound</td>
<td>593</td>
</tr>
<tr>
<td>Local with estimate $[1 \ldots \infty]$</td>
<td>374</td>
</tr>
<tr>
<td>Local with estimate $[0 \ldots 1]$</td>
<td>43</td>
</tr>
<tr>
<td>Solvable loop recurrences</td>
<td>127</td>
</tr>
<tr>
<td>Total analyzed</td>
<td>1276</td>
</tr>
</tbody>
</table>

**Figure:** Expression sequence length analysis on Maple library source
## Results (overall)

<table>
<thead>
<tr>
<th>Surface Type</th>
<th>Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local type is $\subseteq \mathcal{T}_{\text{Expression}}$</td>
<td>827</td>
</tr>
<tr>
<td>Local w/ fully-inferred type</td>
<td>721</td>
</tr>
<tr>
<td>Local whose value is a posint</td>
<td>342</td>
</tr>
<tr>
<td>Local whose value is a list</td>
<td>176</td>
</tr>
<tr>
<td>Local whose value is a set</td>
<td>56</td>
</tr>
<tr>
<td>Solvable loop recurrences</td>
<td>267</td>
</tr>
<tr>
<td>Total analyzed</td>
<td>1330</td>
</tr>
</tbody>
</table>

**Figure:** Surface type analysis on Maple library source
Methodology and Design

Methodology:

- All analyses based on abstract interpretation and constraints
- Analyses divided into 2 stages: constraint generation and constraint solving.
- Try to leverage the underlying CAS (ex: recurrences)

Design:

- Constraint Generation is completely generic (i.e. parametric in the abstract domain)
- Constraint generation done in 2 passes through AST:
  1. Determine local constraints
  2. Aggregate constraints into system
- Constraint solving is partly generic, partly ad hoc
Conclusion and Future Work

Done:
- Definitely improve “error” reporting
- Improves understanding of *extra* polymorphism in Maple code
- Enables a lot of further work

Still needs done:
- Extend approach to additional properties
- Use various *product lattices* for analysis domains
- Handle more varieties of recurrences
- Use these analyses in other tools (e.g. mint, compiler, partial evaluator)