Property Inference for Maple: An Application of Abstract Interpretation

Jacques Carette[1] Stephen Forrest[1]

[1]McMaster University
1280 Main St. W., Hamilton, ON, Canada

Calculemus 2007

Outline

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"

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]

From Inference to Static Analysis

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

From Inference to Static Analysis

- 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 $5 \ldots 12$) $\mathbb{I}(\mathbb{N})$
  - 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 $5 \ldots 12$) $\mathbb{I}(\mathbb{N})$
  - Length (nops – number of operands). Approx: range $\mathbb{I}(\mathbb{N})$
  - “Dependence”. Approx: set of (possible) dependencies
- Programs
  - Variable reads. Approx: range $\mathbb{I}(\mathbb{N})$
  - Variable writes. Approx: range $\mathbb{I}(\mathbb{N})$
  - **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 $= \emptyset$ means value is "pure"
- Dependence $= \{Digits\}$ means ??

Example

Consider:

```
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 $n \geqslant -1$.
- Though `SimpleInt` contains no explicit types, a “successful” run clearly imposes constraints upon the input. Using these, want to 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 $\alpha : C \rightarrow A$ (*abstraction*) and $\gamma : A \rightarrow C$ (*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 $f : C \rightarrow C$ and $g : A \rightarrow A$, 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 $f_{ij} : C \rightarrow C$

(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: $\wedge$, $\vee$, $\sqsubseteq$),
  - logical operators (and only for now)
  - *abstract interpretation operations*, (like lifted $\oplus$ on intervals, as well as widenings)
  - *recurrences* ($\mathtt{V}_3(n+1) = \mathtt{V}_3(n) \oplus 1 \ldots 1$)
- Second pass: solve constraints (being formalized)

(New?) Symbolic Recurrences over monoidal lattices

- 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.

```
IsPrime := proc(n::integer)
   local S, result;
   S 2 := numtheory:-factorset(n); 3
                               4
   if nops(S) 6 > 1 then
5     result 7 := (false, S 9); 8
   else
      result := true;
             10
   end if;
   return(result);
end proc:          11
```

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:

$$
\begin{aligned}
\mathtt{V}_2 = \mathtt{V}_3, \mathtt{V}_4 = \mathtt{ProcInitVal}(0), \\
\mathtt{V}_4 \preceq \mathtt{ES}(1\ldots1), \mathtt{V}_3 \preceq \mathtt{ES}(1\ldots1), \mathtt{V}_6 \preceq \mathtt{ES}(1\ldots1), \\
\mathtt{V}_7 = \mathtt{V}_8, \mathtt{V}_8 = \mathtt{ES}(1\ldots1) \oplus \mathtt{V}_9, \mathtt{V}_{10} = \mathtt{ES}(1\ldots1), \\
\mathtt{V}_{11} = \mathtt{V}_7 \bigvee \mathtt{V}_{10}
\end{aligned}
$$

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

$\bigvee$ and $\oplus$ are the join and summation operations in $\mathbb{I}(\mathbb{N})$.

Another Example

```
p := proc(u) local x, y;
    x := 2,3,4,5; y := 1;
    while y < 10 do x := (x,1); y := y*u; end do;
    (x, y);
end proc:
```

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

$$\begin{aligned}
\mathtt{LoopIC}(x) &= \mathtt{ES}(4\dots 4)\\
\mathtt{LoopIC}(y) &= \mathtt{ES}(1\dots 1)\\
\mathtt{LoopStepFinal}(x) &= \mathtt{LoopStepInit}(x) \oplus \mathtt{ES}(1\dots 1)\\
\mathtt{LoopStepFinal}(y) &= \mathtt{ES}(1\dots\infty) \wedge \mathrm{T}_{\mathrm{PROD}}(\mathtt{LoopStepInit}(x), \mathtt{ES}(1\dots 1))
\end{aligned}$$

- $\mathrm{T}_{\mathrm{PROD}}(a, b)$ is the induced transfer function (on $\mathbb{I}(\mathbb{N})$) for a product.
- The meet with $\mathtt{ES}(1\dots\infty)$ is needed because Maple does not allow a product of length 0.

Another Example (cont)

When we solve these recurrences, we obtain the results:

$$\begin{aligned}\texttt{LoopFinalVal}(x) &= \texttt{ES}(1\ldots 1) \otimes \texttt{ESize}(\texttt{NumLoopSteps})\\ \texttt{LoopFinalVal}(y) &= \texttt{ES}(1\ldots 1)\end{aligned}$$

- 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

  $$\mathrm{T}_{\mathrm{PROD}}(\texttt{ES}(1\ldots 1), \texttt{ES}(1\ldots 1)) = \texttt{ES}(1\ldots 1)$$

  so our recurrence for y has an exact solution.

Error Detection

Our approach can sometimes detect programming errors.

```
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 $\perp$ in our lattice signify programming errors.

Results (overall)

Expression Sequence Length	Procedures
Local with estimate $\neq [0 \ldots \infty]$	862
Local with finite upper bound	593
Local with estimate $[1 \ldots \infty]$	374
Local with estimate $[0 \ldots 1]$	43
Solvable loop recurrences	127
Total analyzed	1276

Figure: Expression sequence length analysis on Maple library source

Results (overall)

Surface Type	Procedures
Local type is $\subsetneq \mathcal{T}_{\text{Expression}}$	827
Local w/ fully-inferred type	721
Local whose value is a posint	342
Local whose value is a list	176
Local whose value is a set	56
Solvable loop recurrences	267
Total analyzed	1330

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 throught 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)