ParaScope: Advanced Static Analysis for ParaSail, a Parallel Specification and Implementation Language

Tucker Taft
AdaCore Inc

IBM Programming Languages Day
T. J. Watson Research Center
November 2015
Where we are going

• Background on ParaSail and ParaScope
• Conservative Data Race Detection
• Advanced Static Analysis:
  – Abstract Interpretation and alternatives
• Value-number approach to Abstract Interpretation in ParaScope Static Analyzer
• Inferring pre- and post- conditions in ParaScope
ParaSail and ParaScope

- **ParaSail** – Parallel Specification and Implementation Language
  - Pervasively Parallel Language, both implicit and explicit
  - Safety through Simplification:
    - No Global Variables
    - No Parameter Aliasing
    - No Re-Assignable Pointers
    - No Unstructured Locking or Signaling
  - Hoare-like syntax for explicit assertion/precondition/postcondition/invariant
  - Conservative Data Race Detection built into ParaSail front end

- **ParaScope** – ParaSail Static Catcher of Programming Errors
  - Re-engineered Abstract Interpretation Approach using Value Numbers
  - Integrated into LLVM-Based Code Generator
    - Immediate Feedback to Programmer about all possible Run-Time Errors
      - Identifies possible violations of assertions/preconditions/...
    - Feedback to Code Generator to optimize generated LLVM
Structure of ParaSail Code Generator and ParaScope

ParaSail source file → ParaSail parser → Abstract syntax tree → Static & Dynamic Semantic Analysis → Program output

- Parython
- Sparkel
- Javallel parser

ParaSail Analyzer

In Ada
In ParaSail
3rd Party

object modules → IIC → LLVM instructions

Code Generator

PSVM instructions

Error Messages
Conservative Data Race Detection in ParaSail front end

class Race_Cond is
  exports
    func Update_Both(var X : Integer;
       var Y : Integer) is
      X := X + 1 || Y := Y - 1;
    end func Update_Both;
    func Update_One(var X : Integer;
       Y : Integer) is
      X := X + 10;
    end func Update_One
    func Update_And_Return(var X : Integer)
         -> Integer is
      X := X + 100;
      return X - 50;
    end func Update_And_Return
    func Return_Ref(ref X : Integer)
         -> ref Integer is
      return X;
    end func Return_Ref;
end class Race_Cond;

Race_Cond::Update_Both(A, B);
Race_Cond::Update_Both(A, A);
** W/W Data Race on A ----- ^ - ^

Race_Cond::Update_One(A, A);
** W/R Data Race on A ----- ^ - ^

Race_Cond::Update_One(A, -A);
A := Race_Cond::Update_And_Return(A);
Race_Cond::Update_One
 (A, Race_Cond::Return_Ref(B));

Race_Cond::Update_One
 (Race_Cond::Return_Ref(A), <-+
  Race_Cond::Return_Ref(A));  | ** W/R Data Race on A -------^ ----- +

Race_Cond::Update_One
 (Race_Cond::Return_Ref(A),
  Race_Cond::Return_Ref(B));
Data Race Detection handles up-level references

26     func dummy() is
27         var Up : I1 := create()
28         func B(I : I1);
29             func A() is
30                 var X : I1 := create()
31                     B(X)
32         end func A
33         func B(I : I1) is
34             read(I)
35             bump(Up)
36         end func B
37         func C() is
38             var Y : I1 := create()
39                 B(Y)
40             read(Up)
41         end func C
42         func D() is
43             var Y : I1 := create()
44                 B(Y)
45             B(Up)  //  Data Race with Up
46         end func D
47             A()                      //  Data races
48             C()                      //  Warning: W/W Data Race on Up
49             D()                      //  Warning: W/R Data Race on Up
50     block
51         A()                      //  Warning: W/R Data Race on Up
52         A()                      //  Warning: W/W Data Race on Up
53         C()  //  Data races
54     end block
55     end func dummy
ParaScope:
Advanced Static Analysis based on
Re-Engineered Abstract Interpretation
What is Abstract Interpretation?

- Approximates the set of possible states of all variables at each point in a program, to allow proofs for safety, security, or correctness.
- Iterates until a fixed-point, then checks for violations.
- *Constructs* the set of possible values, rather than searching through them (handles large ranges).
- Represents relationships, e.g. $2Y > X$, using, e.g. polygons/polyhedrons

X in 1..8, Y in 1..8, $2Y > X$: 

![Graph showing the relationship between X and Y with the inequality $2Y > X$]
Alternative Program Proof Techniques

- **Model Checking (e.g. SPIN model checker)**
  - Searches through state space for states that violate desirable properties
  - State explosion is a challenge; may limit loop iterations
  - Symbolic Model Checking can help

- **Formal Proof (e.g. SPARK 2014 toolset)**
  - Constructs a series of Verification Conditions (VCs) that represent desired safety, security, or correctness properties that should hold at various points in the program
  - Use SMT Solver or equivalent to prove each Verification Condition
  - Use timeout to determine VC cannot be proved
  - Typically relies on programmer to provide pre/postconditions, loop invariants, etc.
What is the problem with “classic” Abstract Interpretation?

- Polyhedral representation of relationships between variables is fundamentally limiting (e.g. $Y > B - X*Z/A$)
- Many approaches exist (courtesy of WikiPedia):
  - congruence relations on integers
  - convex polyhedra (high computational costs)
  - "octagons"
  - difference-bound matrices
  - linear equalities
- Other issues:
  - Initial value set for inputs may require exploring all paths that reach procedure
  - May require a driver or harness to provide realistic input values
What is the problem with “classic” Abstract Interpretation?

- **Polyhedral representation of relationships between variables** is fundamentally limiting (e.g. \( Y > B - X*Z/A \))
- **Many approaches exist** (courtesy of WikiPedia):
  - congruence relations on integers
  - convex polyhedra (high computational costs)
  - "octagons"
  - difference-bound matrices
  - linear equalities
- **Other issues**:
  - Initial value set for inputs may require exploring all paths that reach procedure
  - May require a driver or harness to provide realistic input values
Can we Re-Engineer Abstract Interpretation to solve this?

Basic Trick used in ParaScope Static Analyzer:

- Trick first learned in 1982 from Bob Morgan for Ada compiler
- Use “Value Numbers” to represent value of computing *interesting* expressions (e.g. “Y * 2 – X”)
- Associate Value Sets (Vsets) with Value Numbers (VNs)
- Unlike variables, value numbers don’t change in value over time
  - but what we know about them does
- Value set “shrinks” when we do a conditional jump
  - e.g. *if* Y*2 > X *then* ...
    - in *then* part we know “Y*2-X” in 1 .. +inf
    - in *else* part we know “Y*2-X” in –inf .. 0
- Also shrinks when we do a check or an assertion
  - e.g. *assert* X + Y > Z → “X+Y-Z” in 1 .. +inf
How do Value Numbers simplify value-set determination?

- Only need to represent simple sets of integers, floats, or addresses for each value number (no polyhedrons!)
- Relationships between VNs are represented in value-number definition table (aka computation table)
- All variables/expressions with same value share a VN
- Each basic block and edge of Control Flow Graph (CFG) has its own map from VN to Value Set
- When one VN’s Vset shrinks, we can efficiently propagate it to all related VNs in same map

Typical VN=>Vset Map:
- VN1 => \{1..4\}
- VN4 => inverse\{null\}
- VN7 => \{0..+inf\}
- VN9 => \{&obj2,&obj4\}
Typical Control Flow Graph

Entry
BB_1

BB_2

BB_3

BB_4

BB_5

Pre-conditions

edge 1 state

edge 2 state

edge 3 state

edge 4 state

post-conditions

phis: 1,3

phis: 8,9

Typical edge state:
VN1 => {1..4}
VN4 => inverse{null}
VN7 => {0..+inf}
VN9 => {&obj2,&obj4}

Back edge
Overall 3-phase Structure of ParaScope Static Analyzer

ParaScope Phase Responsibilities

**CFG**
- Identifies Basic Blocks and Builds (Sequentialized) CFG
- Converts into Single-Static-Assignment form

**SSA**
- Globally Assigns Origin IDs and Value Numbers

**PVP**
- Propagates Value Sets and Gives Warnings
Static Single Assignment/Global Value Numbering (SSA/GVN) phase

- Walk the “Control Flow Graph” of basic blocks and find loops, etc.
- Assign a unique “value number” to every fetch of a variable and every computation
- Use “phi” value numbers at join points to represent alternative values
  - E.g. if X > Y then Max := X else Max := Y end if; Max == ?  => PhiVN(X, Y)
- “Kappa” node introduced to represent value of potential alias after assignment
Goals of Possible Value Propagation (PVP) Phase

• Compute Possible Value Set for every Value Number in every Basic Block
  – Map of Value number to Value set

• Check for failures of run-time checks, user assertions, and preconditions of called routines
  – Initially assume checks will pass and thereby infer Pre/Postconditions
  – Iterate until a fixed point
  – Make final pass to report checks that still fail

Typical VN=>Vset Map:
VN1 => {1..4}
VN4 => inverse{null}
VN7 => {0..+inf}
VN9 => {&obj2,&obj4}
PVP Iterative Cycle

- Works one basic-block (BB) at a time.
- Initializes the “checks” table for each BB
  - Summary of all “implicit” and “explicit” checks performed in BB for each VN
- Applies checks of BB and then propagates changes in VN value sets until they stabilize
  - Propagate “down” to constituent VNs, then “up” to composite VNs
  - e.g. VN1 = VN2 * VN3
    - Shrink VN1, propagate “down” to VN2 and VN3;
    - Shrink VN3, propagate “up” to VN1.
- Computes VN state for each BB and for each outgoing edge
  - Saves edge states for later iterations
  - Saves exit-block state for pre/postconditions
Example of PVP warning messages

```ada
func Scope_Test
  (X : optional Integer;
   Y : optional Integer) -> Integer is
  var R : Countable_Range<Integer>
    := X .. Y
  **       ^ --- ^ Operands might be null
  R := R.First + 2 .. R.Last + 2
  {R.Last - 2 == Y}
  **                   ^ -- Assertion Might Fail §
  var L : List<Integer> :=
    (Elem => 3, Next => null)
  const A := L.Elem
  L.Elem := L.Elem + 5
  const B := L.Elem + A
  L.Next := (Elem => B, Next => null)
  {L.Next.Elem == 11}
  {B == A + 6}
  **          ^ -- Assertion will fail here
  {B > X}
  **       ^ -- Assertion might fail
  if X is null then
    ** Edge 9 is dead; cannot reach here
    return 6
  elsif Y is null then
    ** Edge 12 is dead; cannot reach here
    {(X not null) == #true}
    return X
  else
    {X is null and Y is null}
  **                   ^ -- Assertion will fail here
  if X > Y then
    {X - Y > 0}
    return X - Y
  else
    {X - Y <= 0}
    null
  end if
  end if
  **                       ^ -- Uninitialized function result here
end Scope_Test
```

§ False Positive (associativity NYI)
Inferring Preconditions in PVP phase

- **Each Input** (parameter or global) is given an “Input VN” to represent its (unknown) *initial value*
- **Vset for Input VN is full possible range of type**
  - *e.g.* Inp_VN1 in $-2^{31} .. +2^{31}-1$
- **As we apply checks and assertions Vset for Input VN may shrink, eliminating the “bad” values.**
- **VNs corresponding to combinations of Inputs and Literals** (e.g. Inp_VN1 – Inp_VN2 * 2) **might also undergo checks/Assertions and might shrink** (directly or indirectly)
- **At exit block, Vset of Input VN or combination thereof represents the “good” values** (those that survived the checks and assertions), i.e. a *precondition*

*e.g.* Inp1 – Inp2 * 2 in $1 .. +\text{inf}$; Inp2 in $-\text{inf} .. -1 \mid +1 .. +\text{inf}$

➔ *Preconditions*: Inp1 > Inp2 * 2; Inp2 != 0
Inferring Postconditions in PVP phase

Same principle applies for Postconditions ...

- In Exit Block, Vset associated with a VN that represents the final value of some Output (parameter, global, or function result) or combination thereof, represents possible values upon completion, i.e. a postcondition

- Example:

  ```
  proc Incr(X : in out Integer) is
      X := X + 1
  end proc Incr
  
  initial value of X is Inp_VN1: Inp_VN1 in -2^{31} .. +2^{31}-1
  final value of X is VN2 = Inp_VN1 + 1: VN2 in -2^{31}+1 .. +2^{31}
  check that X + 1 doesn’t overflow: VN2 in -2^{31}+1 .. +2^{31}-1
  propagates to: Inp_VN1 in -2^{31} .. +2^{31}-2
  ➔ precondition: X’Initial <= 2^{31}-2
  ➔ postcondition: X’Final in -2^{31}+1 .. +2^{31}-1; X’Final = X’Initial+1
  ```
Screen shot showing Inferred Pre/Postconditions

```
-- Subp: fsw_example

-- Preconditions:
--   N >= 1

-- Postconditions:
--   A = One-of{1, 101, N - 1}
--   A in (0..121 | 789..2^{31}-2)
--   B = One-of{2, 102, N}
--   B in (1..122 | 790..2^{31}-1)
--   B = A + 1

-- Test Vectors:
--   N: {123..456}, {457..789}, {1..122 | 790..2^{31}-1}
--   A: {1}, {101}
--   B: {2}, {102}

procedure Fsw_Example (N : Natural;
   A, B : out Natural) is
begin
   case N is
      when 123 .. 456 =>
         A := 1;
         B := 2;
      when 457 .. 789 =>
         A := 101;
         B := 102;
      when others =>
         A := N - 1;
         B := N;
   end case;
   pragma Assert (B - A = 1);
end Fsw_Example;
```
Summary

- **ParaSail Simplifies Conservative Data Race Detection**
  - Lack of Re-Assignable Pointers and Parameter Aliasing simplify problem
  - Can do checks at compile-time
  - Simplifies later static analysis

- **ParaScope Advanced Static Analysis Integrated with Compiler**
  - Uses Re-Engineered Abstract Interpretation based on Value Numbering
  - Assumes No data races are possible so can treat as sequential program
  - Gives immediate feedback to programmer about all possible run-time errors

- **Re-engineered value-number based approach in ParaScope:**
  - Can represent arbitrary relationships between variables
  - Uses efficient mechanism to propagate information between value numbers
  - Can infer both numeric and symbolic pre/postconditions so no need for drivers/harnesses or top-down walk