Bringing Static Analysis to the Masses:
S. Tucker Taft
October 2010
Outline of Presentation

- Why aren’t the masses using static analysis yet?
- What can we do about it?
- Integration into the development process
  - Integration into the IDE
  - Integration into the build process
  - Integration into the compiler
  - Integration into the language
- The design of ParaSail
  - *Parallel Specification and Implementation Language*
Why aren’t the Masses using Static Analysis yet?

- This very question asked 10 days ago on Linked-In Static Code Analysis Group (by Steve Heffner)

- Many answers, many scapegoats:
  - Blame the customers?
    - Organizational laziness
    - Insecure programmers
  - Blame the marketers?
    - Early versions oversold
    - Current versions undersold
    - “Static Analysis” is a boring name
  - Blame the tools?
    - Too slow
    - Too much noise
    - Difficult to incorporate into build process
  - Blame the President?! (it is an election year after all)
What can we do about it?

- Make static analysis an integral part of the development process rather than after-the-fact
- Provide “One Button” ease of use
- Run at a speed comparable to rest of build
  - Incremental analysis
  - Provide multiple depths of analysis
    - Similar to compiler optimization levels
- Provide success/failure indicator which can determine overall success/failure of build
  - Analogous to gcc’s “warnings are errors” (-Werror)
  - False positives must be easy enough to accommodate by suppressing or making a benign change
  - Get tools to agree on what is/is not a problem
Levels of Integration

- Integration into the IDE
  - IDE plugin architecture should make this easier
  - e.g. Eclipse panel combines compiler and analyzer messages
  - Just check one box, or click on one menu item to produce static analysis results

- Integration into the Compiler/Linker
  - Use compiler’s front end
  - Avoids front end incompatibilities and quirks
  - No need for separate configuration for target, subdirectories, libraries, etc.
  - Examples: Green Hills DoubleCheck and AdaCore CodePeer
  - Static analysis can then be seen as Enhanced Compile-Time Checking -- less threatening?
Eiffel helped to popularize notion of integrating annotations with language

SPARK is example of this based on Ada

JML and standard annotations like @nonnull do this for Java

But... These still rely on run-time checks and/or on separate tools -- we want compile-time checking.

Can we require compile-time enforcement of all user annotations and all language-defined checks (e.g. array indexing, null pointer, etc.) as part of the language definition?

- Java sticks “toe” in the water with initialization of local variables
Is It Time to Design a Language for Safe and Secure Parallel Programming?

- What is New?
  - Hardware is no longer getting any faster
    - It is getting more parallel, and hence more difficult to program safely
  - Safety and Security is now everyone’s concern
    - Everything is networked
  - Deep and Precise Static Analysis is coming of age
    - We can do sophisticated things in the compiler/linker

- What is True?
  - 80+% of safety-critical systems are developed in C and C++, two of the least safe languages invented in the last 40 years
  - In 10 years, many chips will have 64+ cores
  - Software has become the focus of more and more investment in almost all industries (e.g. 40% of R&D for automobiles)
Designing A New Language

- **ParaSail** -- *Parallel Specification and Implementation Language*

- Designed to make parallel programming safe and convenient

- All checking is done at compile-time
  - No run-time checking, no run-time exceptions
  - No race conditions
  - User-definable safety and security constraints

- Heavy duty static analysis done by the compiler
  - Program fails to compile if compiler can’t prove assertions
What makes ParaSail Interesting?

- Pervasive (implicit and explicit) parallelism

- Inherently safe:
  - preconditions, postconditions, constraints, etc., integrated throughout the syntax
  - no global variables; no dangling references
  - no run-time checks -- all checking at compile-time
  - no run-time exceptions

- Small number of flexible concepts:
  - Modules, Types, Objects, Operations

- User-defined literals, indexing, aggregates, physical units checking

- It’s hot off the presses
Parallelism in ParaSail

- Parallel by default
  - parameters are evaluated in parallel
  - have to work harder to make code run sequentially

- Easy to create even more parallelism
  - Process(X) || Process(Y) || Process(Z);

- Lock-based and lock-free concurrent objects
  - Lock-based objects also support queued access
  - User-defined delay and timed call based on queued access

- No global variables
  - Can only access or update variable state via parameters

- Compiler prevents aliasing and unsafe access to non-concurrent variables
Examples of ParaSail Parallelism

\[ Z := F(U) + G(V); \quad // F(U) and G(V) eval’ed in parallel \]

\[ \text{Process(A) || Process(B) || Process(C); \quad // All 3 in parallel} \]

\textbf{for} X \Rightarrow \textbf{Root} \textbf{then} X.\text{Left} || X.\text{Right} \textbf{while} X \textbf{not null} \textbf{concurrent loop} \[
\begin{align*}
& \text{Process(X);} \quad // \text{Process called on each node in parallel} \\
& \textbf{end loop;} \\
\end{align*}
\]

\textbf{concurrent interface} Box\langle Element is Assignable\rangle is
\[
\begin{align*}
& \textbf{function} \text{Create()} \rightarrow \text{Box;} \quad // \text{Creates an empty box} \\
& \textbf{procedure} \text{Put}(M : \text{locked var} \text{Box; E : Element}); \\
& \textbf{function} \text{Get}(M : \text{queued var} \text{Box}) \rightarrow \text{Element}; \quad // \text{May wait} \\
& \textbf{function} \text{Get\_Now}(M : \text{locked const Box}) \rightarrow \text{optional Element}; \\
& \textbf{end interface} \text{Box;} \\
\end{align*}
\]

\textbf{type} Item\_Box \textbf{is} Box\langle Item\rangle; \\
\textbf{var} My\_Box : \text{Item\_Box} := \text{Create();}
Annotations in ParaSail

- Preconditions, Postconditions, Constraints, etc. all use the same Hoare-like syntax: \{X \neq 0\}

- All assertions are checked at compile-time
  - no run-time checks inserted
  - no run-time exceptions to worry about

- Location of assertion determines whether is a:
  - precondition (before “\(\rightarrow\)”)  
  - postcondition (after “\(\rightarrow\)”)  
  - assertion (between statements)  
  - constraint (in type definition)
Examples of ParaSail Annotations

interface Stack <Component is Assignable>; Size_Type is Integer>> is

   function Max_Stack_Size(S : Stack) -> Size_Type {Max_Stack_Size > 0};

   function Count(S : Stack) -> Size_Type
       {Count <= Max_Stack_Size(S)};

   function Create(Max : Size_Type {Max > 0}) -> Stack
       {Max_Stack_Size(Create) == Max and Count(Create) == 0};

   function Is_Empty(S : Stack) -> Boolean
       {Is_Empty == (Count(S) == 0)};

   function Is_Full(S : Stack) -> Boolean
       {Is_Full == (Count(S) == Max_Stack_Size(S))};

   procedure Push(S : ref var Stack {not Is_Full(S)}; X : Component)
       {Count(S') == Count(S) + 1};

   function Top(S : Stack {not Is_Empty(S)}) -> Component;

   procedure Pop(S : ref var Stack {not Is_Empty(S)})
       {Count(S') == Count(S) - 1};

end interface Stack;
type Age is new Integer<0 .. 200>;
type Youth is Age {Youth <= 20};
type Senior is Age {Senior >= 50};

----------------------------------

defunction GCD(X, Y : Integer {X > 0 and Y > 0}) -> Integer

{GCD > 0 and GCD <= X and GCD <= Y and
X mod GCD == 0 and Y mod GCD == 0} is

var Result := X; {Result > 0 and X mod Result == 0}
var Next := Y mod X; {Next <= Y and Y - Next mod Result == 0}

while Next != 0 loop
{Next > 0 and Next < Result and Result <= X}
const Old_Result := Result;
Result := Next; {Result < Old_Result}
Next := Old_Result mod Result;
{Result > 0 and Result <= Y and Old_Result - Next mod Result == 0}
end loop;

return Result;
end function GCD;
ParaSail has four basic concepts:

- **Module**
  - has an Interface, and Classes that implement it
  - `interface M <Formal is Int<>> is ...

- **Type**
  - is an instance of a Module
  - `type T is M <Actual>;

- **Object**
  - is an instance of a Type
  - `var Obj : T := T::Create(...);

- **Operation**
  - is defined in a Module, and
  - operates on one or more Objects of specified Types.
User-defined Indexing, Literals, etc.

- User-defined indexing
  - Any type with **operator** “[]” defined
  - Indexing function returns **ref** to component of parameter

- User-defined literals
  - Any type with **operator** “from_univ” defined from:
    - Univ_Integer (42), Univ_Real (3.141592653589793)
    - Univ_String (“Hitchhiker’s Guide”), Univ_Character (‘π’)
    - UnivEnumeration (#red)

- User-defined ordering
  - Define single binary **operator** “=?” (pronounced “compare”)
  - Returns #less, #equal, #greater, #unordered
More Examples of ParaSail

concurrent class Box <Element is Assignable<>> is
    var Content : optional mutable Element; // starts null and can change size
exports
    function Create() -> Box is // Creates an empty box
        return (Content => null);
    end function Create;

procedure Put(M : locked var Box; E : Element) is
    M.Content := E;
end procedure Put;

function Get(M : queued var Box) -> Element // May wait
    queued until Content not null is
        const Result := M.Content;
        M.Content := null;
        return Result;
end function Get;

function Get_Now(M : locked const Box) -> optional Element is
    return M.Content;
end function Get_Now;
end class Box;
abstract concurrent interface Clock <Time_Type is Ordered<> is
  function Now(C : Clock) -> Time_Type;
procedure Delay_Until(C : queued Clock; Wakeup : Time_Type)
   {Now(C')} >= Wakeup}; // queued until Now(C) >= Wakeup
end interface Clock;

concurrent interface Real_Time_Clock<...> extends Clock<...> is
  function Create(...) -> Real_Time_Clock;
  ...
end interface Real_Time_Clock;

var My_Clock : Real_Time_Clock <...> := Create(...);
const Too_Late := Now(My_Clock) + Max_Wait;

select // multi-way parallel queued call
  const Data := Get(My_Box) => Process(Data);
  || Delay_Until(My_Clock, Wakeup => Too_Late) =>
     Put_Line(Out_Stream, "My_Box not filled in time");
end select;
type Node_Kind is Enum < [#leaf, #unary, #binary] >;
...
for X => Root while X not null loop
  case X.Kind of
    #leaf =>
      Process_Leaf(X);
    #unary =>
      Process_Unary(X) ||
      continue loop with X => X.Operand;
    #binary =>
      Process_Binary(X) ||
      continue loop with X => X.Left ||
      continue loop with X => X.Right;
  end case;
end loop;
Parallel N-Queens Solution

**interface** N_Queens <N : Univ_Integer := 8> is
   // Place N queens on an NxN checkerboard so that none of them can
   // "take" each other. Return vector of solutions, each solution being
   // an array of columns indexed by row indicating placement of queens.

   **type** Chess_Unit is new Integer<-N*2 .. N*2>;
   **type** Row is Chess_Unit {Row in 1..N};
   **type** Column is Chess_Unit {Column in 1..N};
   **type** Solution is Array<optional Column, Indexed_By => Row>;

   **function** Place_Queens() -> Vector<Solution>
   {for all S of Place_Queens: for all C of S: C not null};
end interface N_Queens;
class N_Queens is
    type Sum_Range is Chess_Unit {Sum_Range in 2..2*N};
    type Diff_Range is Chess_Unit {Diff_Range in (1-N) .. (N-1)};
    type Sum is Set<Sum_Range>;
    type Diff is Set<Diff_Range>;
exports
    function Place_Queens() -> Vector<Solution>
    {for all S of Place_Queens: for all C of S: C not null}
    is
        var Solutions : concurrent Vector<Solution> := [];
        *Outer_Loop*
        for (C : Column := 1; Trial : Solution := [.. => null];
            Diag_Sum : Sum := []; Diag_Diff : Diff := []) loop
            // Iterate over the columns
            ...
            Solutions |= Trial;
            ...
        end loop Outer_Loop;
        return Solutions;
    end function Place_Queens;
end class N_Queens;
Parallel N-Queens Solution (cont’d)

```java
function Place_Queens() -> Vector<Solution> is
  var Solutions : concurrent Vector<Solution> := []; *Outer_Loop*
  for (C : Column := 1; Trial : Solution := [.. => null];
    Diag_Sum : Sum := []; Diag_Diff : Diff := []) loop // over the columns
    for R in Row concurrent loop // over the rows
      if Trial[R] is null and then
        (R+C) not in Diag_Sum and then (R-C) not in Diag_Diff then // Found a Row/Column combination that is not on any diagonal
          if C < N then // Keep going since haven't reached Nth column.
            continue loop Outer_Loop with (C => C+1,
              Trial => Trial | [R => C],
              Diag_Sum => Diag_Sum | (R+C),
              Diag_Diff => Diag_Diff | (R-C));
          else // All done, remember trial result.
            Solutions |= Trial;
          end if;
        end if;
      end loop;
  end loop Outer_Loop;
  return Solutions;
end function Place_Queens;
```
How does ParaSail Compare to ...

- C/C++ -- built-in safety; built-in parallelism
- Ada -- eliminates race conditions, increases parallelism, eliminates run-time checks, simplifies language
- Java -- eliminates race conditions, increases parallelism, avoids garbage collection, no run-time exceptions, compile-time checks against security constraints
Some of the Open Issues in ParaSail

- If we eliminate pointers, what about “references”?  
  - if references, when and where?

- If no global variables, how best to provide access to global “singleton” objects from environment  
  - such as “the” database or “the” user or “the” filesystem  
  - “Context” object with singletons as components passed to main subprogram?

- How to standardize how “smart” compiler is at proving assertions  
  - Open source algorithm?  
  - Detailed specification of inference and simplification rules?
interface Float_With_Units
<Base is Float<>; Name : Univ_String; Short_Hand : Univ_String;
Unit_Dimensions : Array <Element_Type => Univ_Real,
    Index_Type => Dimension_Enum> := [.. => 0.0]; Scale : Univ_Real>
is

operator "from_univ"(Value : Univ_Real)
{Value in Base::First*Scale .. Base::Last*Scale} -> Float_With_Units;

operator "to_univ"(Value : Float_With_Units) -> Result : Univ_Real
{Result in Base::First*Scale .. Base::Last*Scale};

operator "+"(Left, Right : Float_With_Units) -> Result : Float_With_Units
{[[Result]] == [[[Left]] + [[[Right]]]};

operator "=?"(Left, Right : Float_With_Units) -> Ordering;

operator "*"(Left : Float_With_Units; Right : Right_Type is Float_With_Units<>)
-> Result : Result_Type is Float_With_Units<
    Unit_Dimensions =>
    Unit_Dimensions + Right_Type.Unit_Dimensions>
{[[Result]] == [[[Left]] * [[[Right]]]};

operator "/"(Left : Left_Type is ...
end interface Float_With_Units;

type Meters is Float_With_Units<Name => "centimeters", Short_Hand => "cm",
    Unit_Dimensions => [#m => 1.0, #k => 0.0, #s => 0.0], Scale => 0.01>;}
Conclusions

- Static analysis hasn’t reached the masses yet
- Integration into the development process is essential
  - Ideally into the compiler/linker
- Integration into the language is the ultimate step -- it becomes a non-optional part of the process
- When designing a new language, can unify and simplify
- Can focus on new issues
  - pervasive parallelism
  - integrated annotations enforced at compile-time
- Read the blog if you are interested...

http://parasail-programming-language.blogspot.com
11 Cypress Drive
Burlington, MA 01803-4907

Tucker Taft
tucker.taft@sofcheck.com

http://parasail-programming-language.blogspot.com

+1 (781) 750-8068 x220