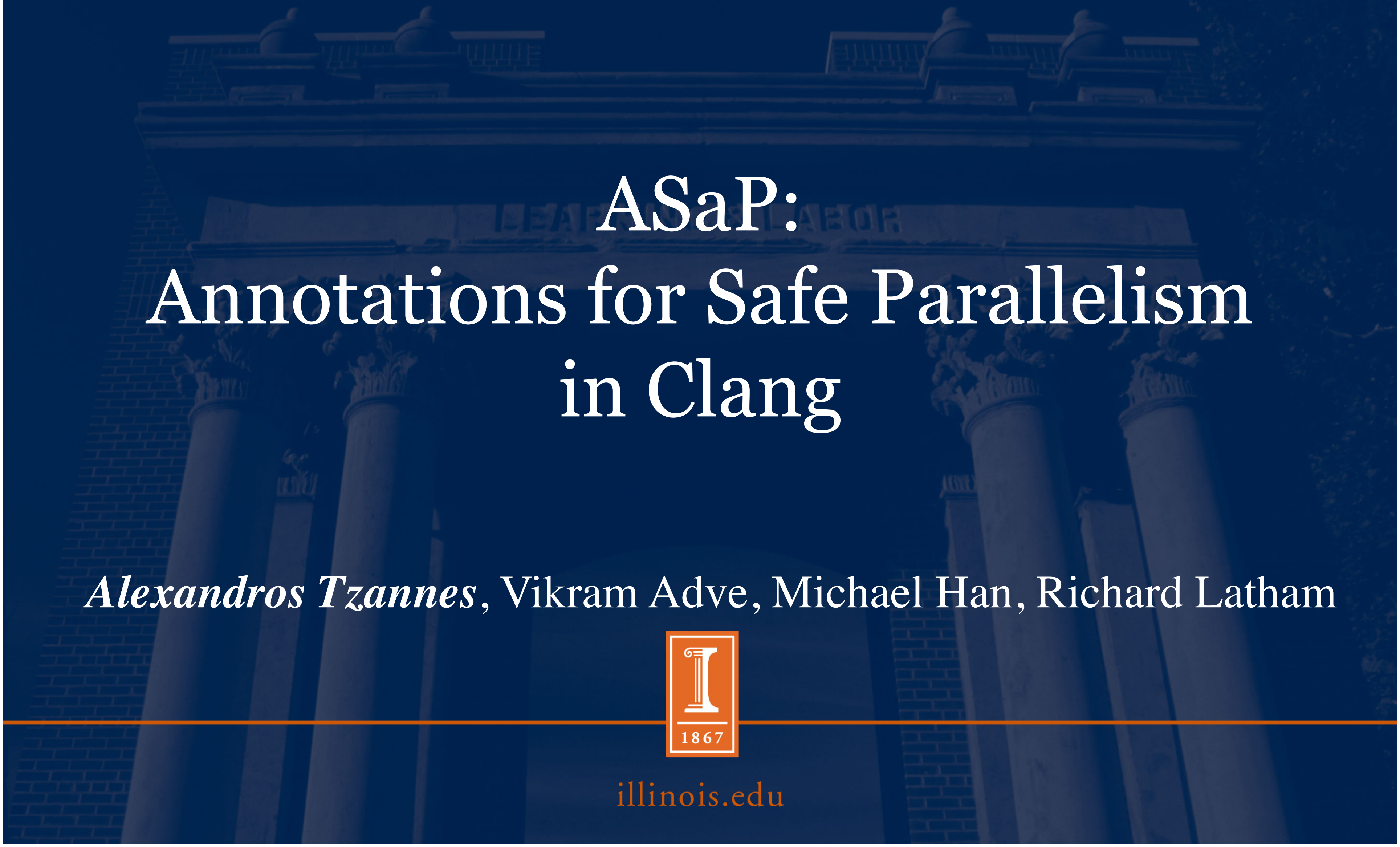
UNIVERSITY OF ILLINOIS

AT URBANA-CHAMPAIGN



ASaP: Annotations for Safe Parallelism in Clang

Alexandros Tzannes, Vikram Adve, Michael Han, Richard Latham

1867

Motivation

Debugging parallel code *is hard!!*

- Many bugs are hard to reason about & reproduce
  - Data-races, atomicity violations, deadlocks, …

Existing tools

- Dynamic (e.g., race-detectors, …)
  - No guarantees, overheads, false positives
- Static Analysis
  - False positives, interprocedural analysis, limited scope

Annotations for Safe Parallelism

Static analysis

- ✓ Strong Guarantees:
  - ✓ Race Freedom
  - ✓ Strong Atomicity
- ✓ Modular checking (one function at a time)
- ✓ Annotation Based
  - ✓ Rich Expressiveness
  - ✓ Checked!
  - ✓ Annotations to silence false positives
- ✗ Annotation Burden
  - ✓ Full Annotation Inference – In progress!

A little history

- Deterministic Parallel Java
  - [OOPSLA09] Deterministic Fork-Join Algorithms
  - [ASE09] Partial Annotation Inference
  - [POPL11] Adds Disciplined Non-Determinism
    [ECOOP11] Parallel Frameworks
  - [PPoPP13] Tasks w. Effects Java
- ASaP
  - Collaboration w. Autodesk [2010 - today]
  - Implementation of ASaP in Clang

Collaboration w. Autodesk

- Weekly meetings *(w. Michael Han & Richard Latham)* :
  - active feedback into the design of ASaP
  - E.g., common parallelism patterns to support
  - E.g., flag functions that write to globals or statics

Goal:

- Static checking of ASM thread safety requirements
  - 3~4 MLOC internal Autodesk library
  - ASM uses structured parallelism internally
    - *parallel_for, scoped locks, …*
  - External (client) parallelism may be unstructured

Collaboration w. Autodesk (2)

Focus on use of ASM as a thread-safe library

- Library API challenges:
  - Is library code thread safe w.r.t its API spec?
  - Check that client code honors API parallelism restrictions

Outline

- What do these ASaP annotations look like/do?
- How is this ASaP checker designed/built?
  - Architecture, Implementation, …
- Nice prototype! What else will come standard?
  - Expressiveness (patterns & parallel APIs)
  - Annotation Inference

Outline

- *What do these ASaP annotations look like/do?*

- How is this ASaP checker designed/built?
  - Architecture, Implementation, …

- Nice prototype! What else will come standard?
  - Expressiveness (patterns & parallel APIs)
  - Annotation Inference

illinois.edu

Regions & Effects

- A *Region* contains one or more memory locs
  - Hierarchical regions supports various patterns
  - Control aliasing *(E.g., ptr in R1 points to R2)*
- *Effect Summaries* describe a function's effects
  - E.g., reads/writes over regions
  - Make checking modular
  - Checked – not trusted!
- Parallel Safety = Non-Interference of Effects

Example 1: Field Distinction

```
class Point {

    int <Rx> X;
    int <Ry> Y;

    void setX(int _X) writes Rx { X = _X; }
    void setY(int _Y) writes Ry { Y = _Y; }

    void set(int _X, int _Y) writes Rx, Ry {
        parallel_invoke({setX(_x);}, {setY(_y);});
    }
};
```

Example 1: Field Distinction

```
class Point {
        region Rx, Ry;
        int<Rx> X;
        int<Ry> Y;

        void setX(int _X) writes Rx  { X = _X; }
        void setY(int _Y) writes Ry  { Y = _Y; }

        void set(int _X, int _Y) writes Rx, Ry  {
                parallel_invoke({setX(_x);}, {setY(_y);});
        }       // {writes Rx} # {writes Ry}
};
```

Example 1: Field Distinction

```
class Point {
        region Rx, Ry;
        int<Rx> X;
        int<Ry> Y;

        void setX(int _X) writes Rx { X = _X; }
        void setY(int _Y) writes Ry { Y = _Y; }

        void set(int _X, int _Y) writes Rx, Ry {
                parallel_invoke({setX(_x);}, {setY(_y);});
        }       // {writes Rx} # {writes Ry}
};
```

Example 1: Actual C++ Syntax

```
class [[asap::region(“Rx, Ry”)]] Point {

        int X [[asap::arg(“Rx”)]];
        int Y [[asap::arg(“Ry”)]];

        void setX [[asap::writes(“Rx”)]] (int _X) { X = _X; }
        void setY [[asap::writes(“Ry”)]] (int _Y) { Y = _Y; }

        void set [[asap::writes(“Rx, Ry”)]] (int _X, int _Y)  {
                parallel_invoke( [this, _X] () {setX(_X);},
                                 [this, _Y] () {setY(_Y);} );
        }
};
```

Example 2: Object Distinction

```
class Point<region P> {
        int<P> X;
        int<P> Y;
        void set(int _X, int _Y) writes P {
                X = _X; // writes P
                Y = _Y; // writes P (cannot parallelize)
}       };

region R1, R2;
void foo(Point<R1> &P1, Point<R2> &P2) writes R1, R2 {
        parallel_invoke({P1.set(1,2);}, {P2.set(3,4);} );
}
```

Example 2: Object Distinction (2)

```
class Point<region P> {
        int<P> X;
        int<P> Y;
        void set(int _X, int _Y) writes P {
                X = _X; // writes P
                Y = _Y; // writes P (cannot parallelize)
}       };

< region P1, region P2,  P1:* # P2:* >
void foo(Point<P1> &P1, Point<P2> &P2) writes P1, P2 {
        parallel_invoke({P1.set(1,2);}, {P2.set(3,4);} );
}
```

illinois.edu

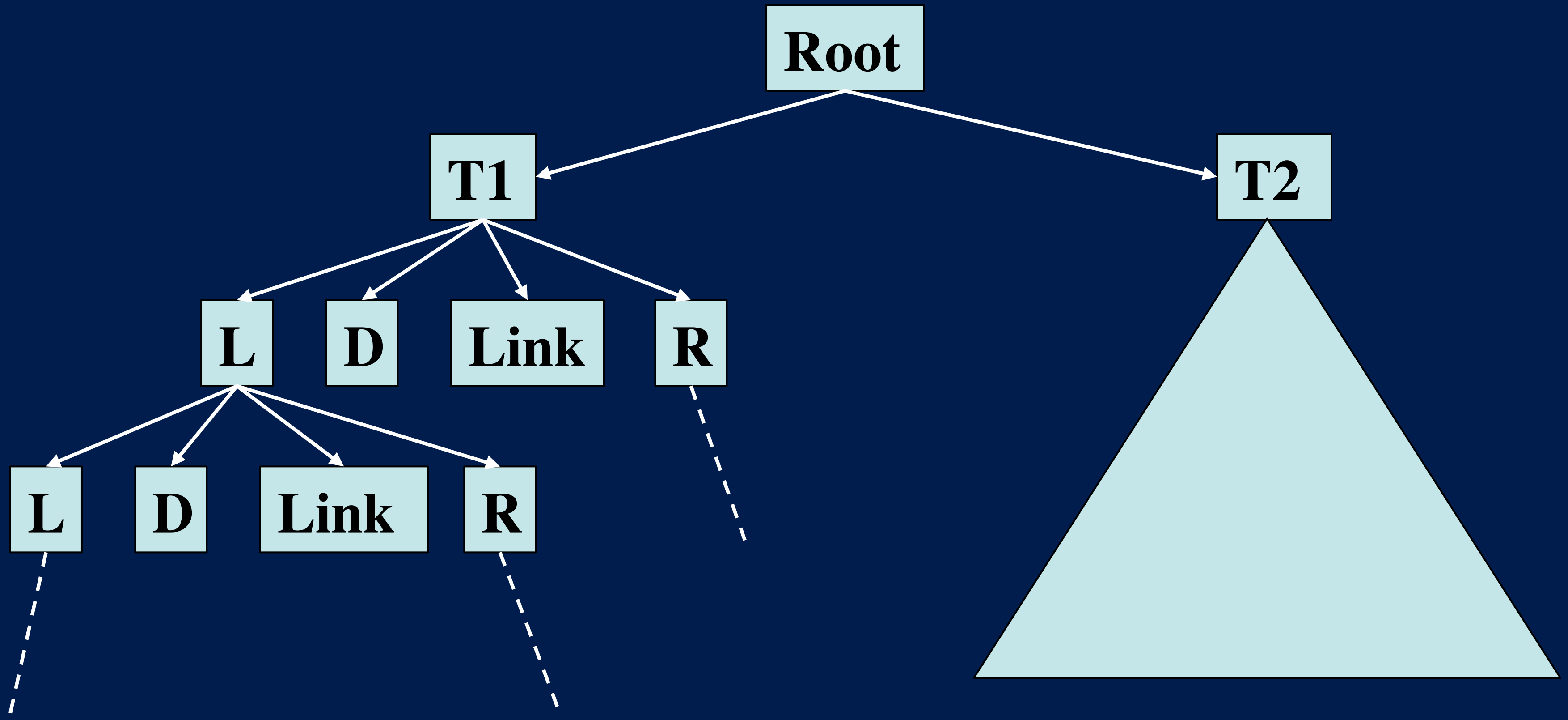
Example 3: Object & Field Distinction

```
class Point<region P> {  region X, Y;
        int<P:X> X;
        int<P:Y> Y;
        void set(int _X, int _Y) writes P:X, P:Y {
                parallel_invoke({X=_X;},
                                {Y=_Y;} ); // P:X # P:Y
}       };

region R1, R2;
void foo(Point<R1> &P1,  Point<R2> &P2)
         writes R1:X, R1:Y, R2:X, R2:Y {
        parallel_invoke({P1.set(1,2);}, {P2.set(3,4);} );
                // {R1:X, R1:Y} # {R2:X, R2:Y}
}
```

Example 3: Object & Field Distinction

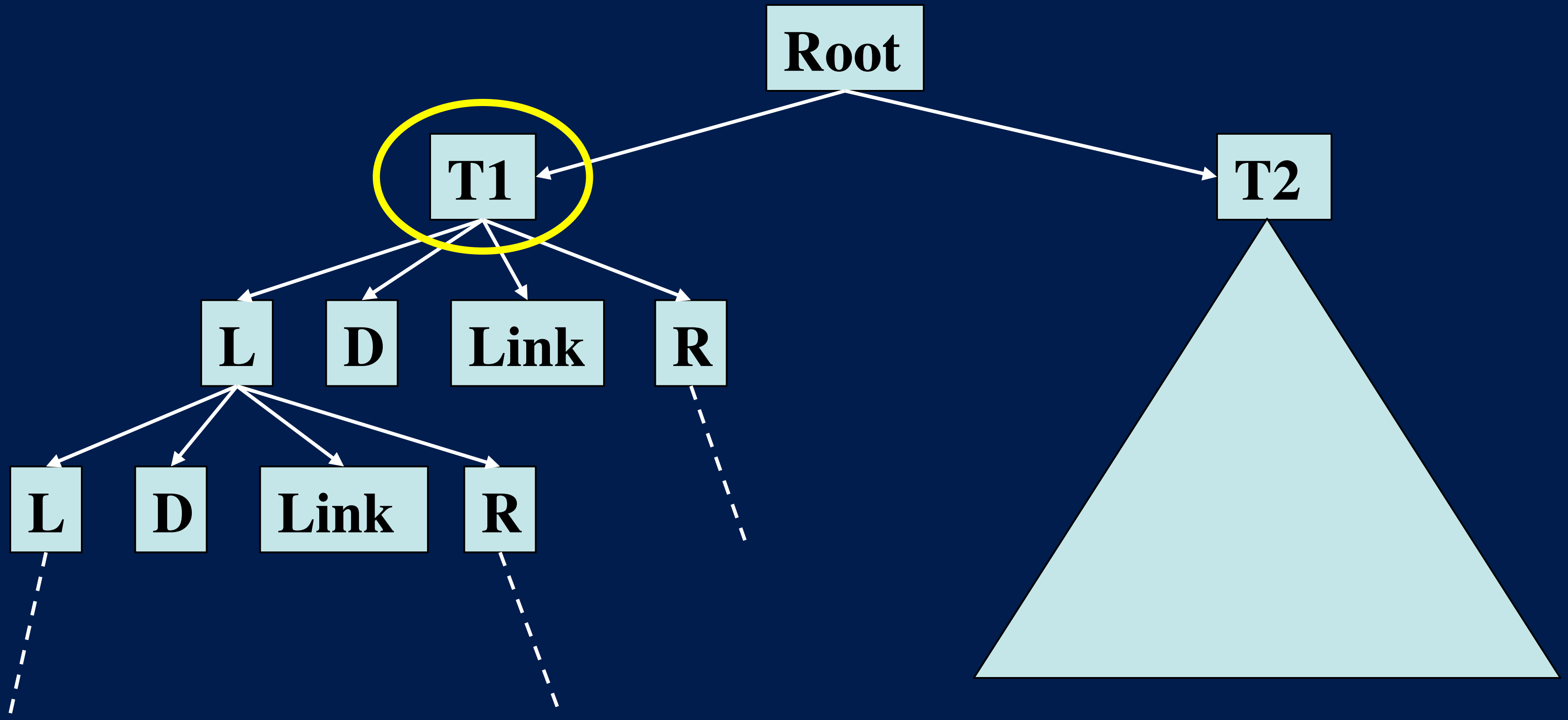
```
class Point<region P> {  region X, Y;
        int<P:X> X;
        int<P:Y> Y;
        void set(int _X, int _Y) writes P:X, P:Y {
                parallel_invoke({X=_X;},
                                {Y=_Y;} ); // P:X # P:Y
}       };

region R1, R2;
void foo(Point<R1> &P1,  Point<R2> &P2)
         writes R1:*, R2:* {
        parallel_invoke({P1.set(1,2);}, {P2.set(3,4);} );
                // {R1:X, R1:Y} # {R2:X, R2:Y}
}
```

RPLs (Under/Included)



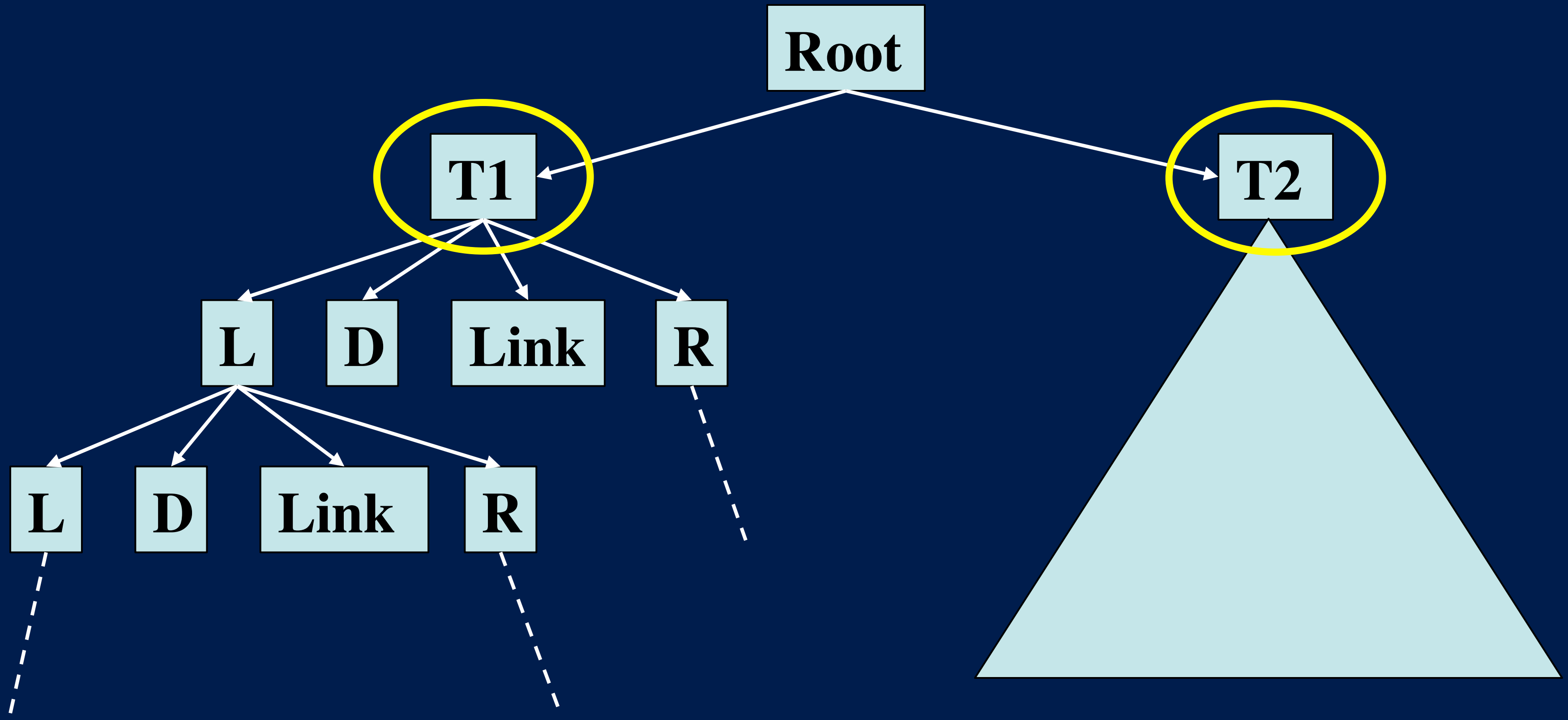
illinois.edu

RPLs (Under/Included)



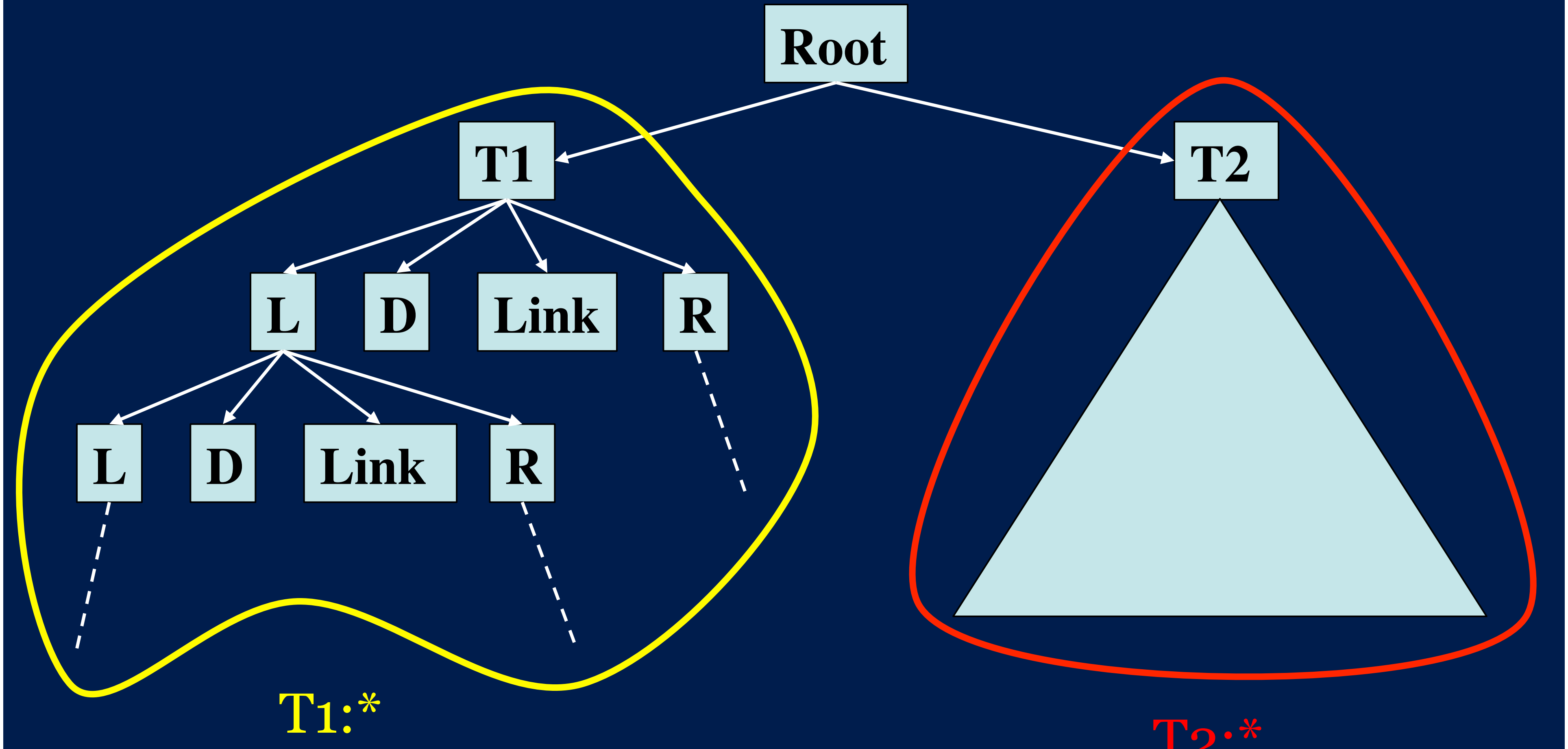
- T1 = Root:T1

RPLs (Under/Included)



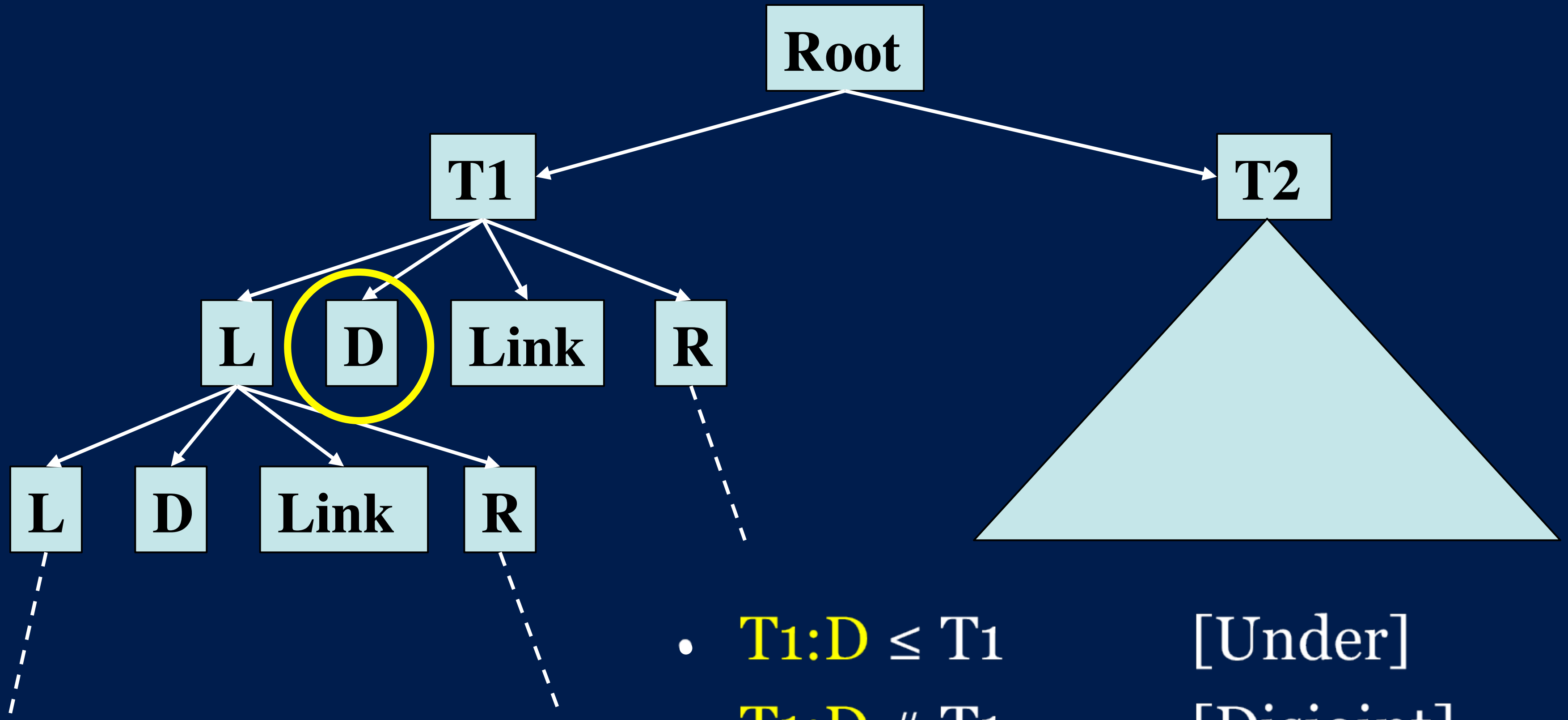
- T1 = Root:T1
- T2 = Root:T2, T1 # T2

RPLs (Under/Included)



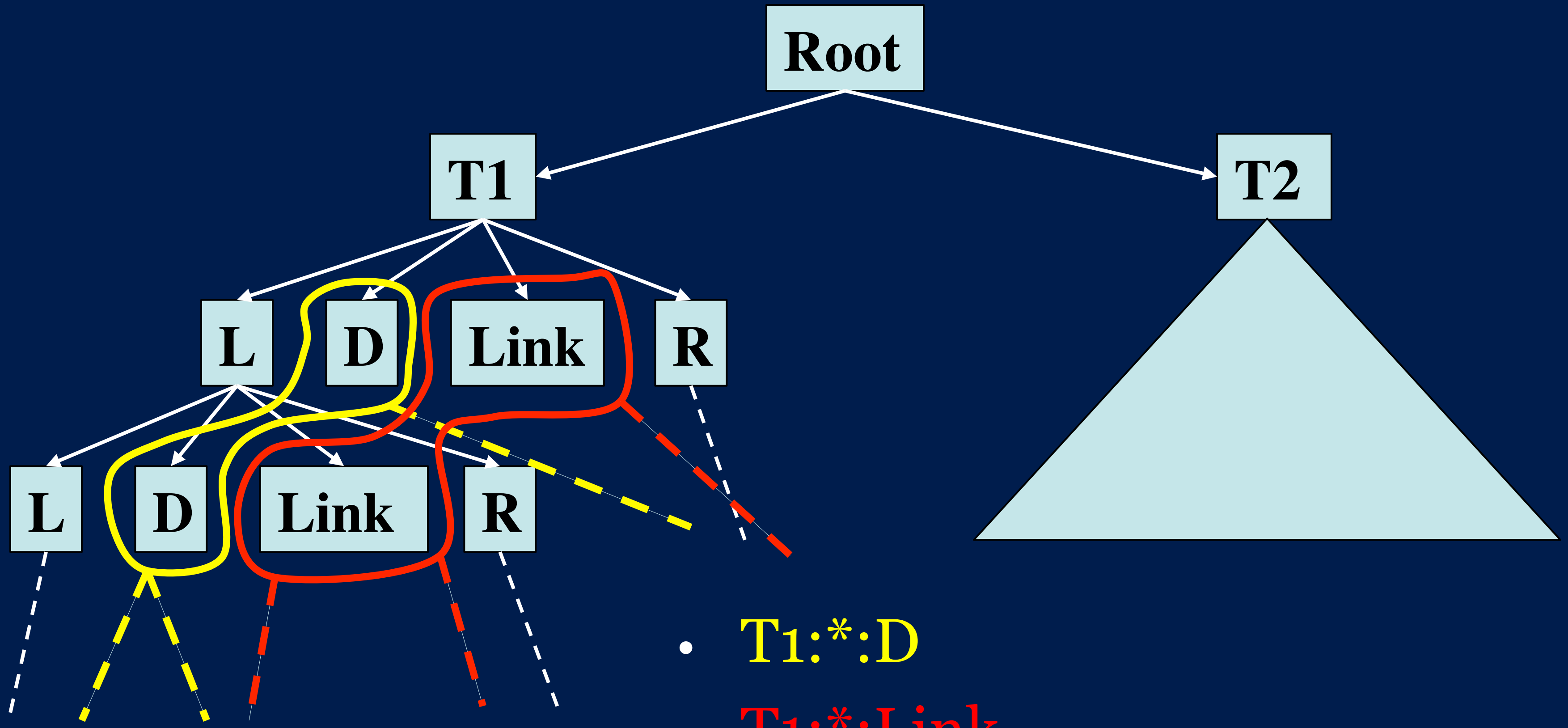
T1:* # T2:*

RPLs (Under/Included)



- T1:D ≤ T1 [Under]
- T1:D # T1 [Disjoint]
- T1:D ⊆ T1:* [Included]

RPLs (Under/Included)



- T1:*:D
- T1:*:Link
- T1:*:D # T1:*:Link

Example 4: Recursion!

```
class ListNode<region P> {      region D, N, Link;
   int<P:D>   Data;
   ListNode<P:N>  * <P:Link>   Next;

   void setAll(int X) reads P:*:Link  writes P:*:D {
      parallel_invoke(
         {Data=X;},
         { if (Next)
              Next->setAll(X);}
      );
}  };
```

{ writes P:D } # { reads P:Link, P:N:*:Link writes P:N:*:D }

Example 4: Recursion!

```
class ListNode<region P> {        region D, N, Link;
   int<P:D> Data;
   ListNode<P:N> *<P:Link> Next;

   void setAll(int X) reads P:*:Link   writes P:*:D {
      parallel_invoke(
         {Data=X;},
         { if (Next)
              Next->setAll(X);}
      );
}   };
```

Example 4: Recursion!

```
class ListNode<region P> {        region D, N, Link;
   int<P:D> Data;
   ListNode<P:N> *<P:Link> Next;

   void setAll(int X) reads P:*:Link   writes P:*:D {
      parallel_invoke(
         {Data=X;},
         { if (Next)
              Next->setAll(X);}
      );
}  };
```

Example 4: Recursion!

```
class ListNode<region P> {        region D, N, Link;
   int<P:D> Data;
   ListNode<P:N> *<P:Link> Next;

   void setAll(int X) reads P:*:Link   writes P:*:D {
      parallel_invoke(
         {Data=X;},                    // writes P:D
         { if (Next)                   // reads P:Link
              Next->setAll(X);}        // invokes setAll [P←P:N]
      );                               // -> reads P:N:*:Link   writes P:N:*:D
}  };
```

{ writes P:D } # { reads P:Link, P:N:*:Link writes P:N:*:D }

Demo!

Recap (Expressiveness)

- Distinguish by
  - Object
  - Field
  - Index (arrays)
    - *Future support*

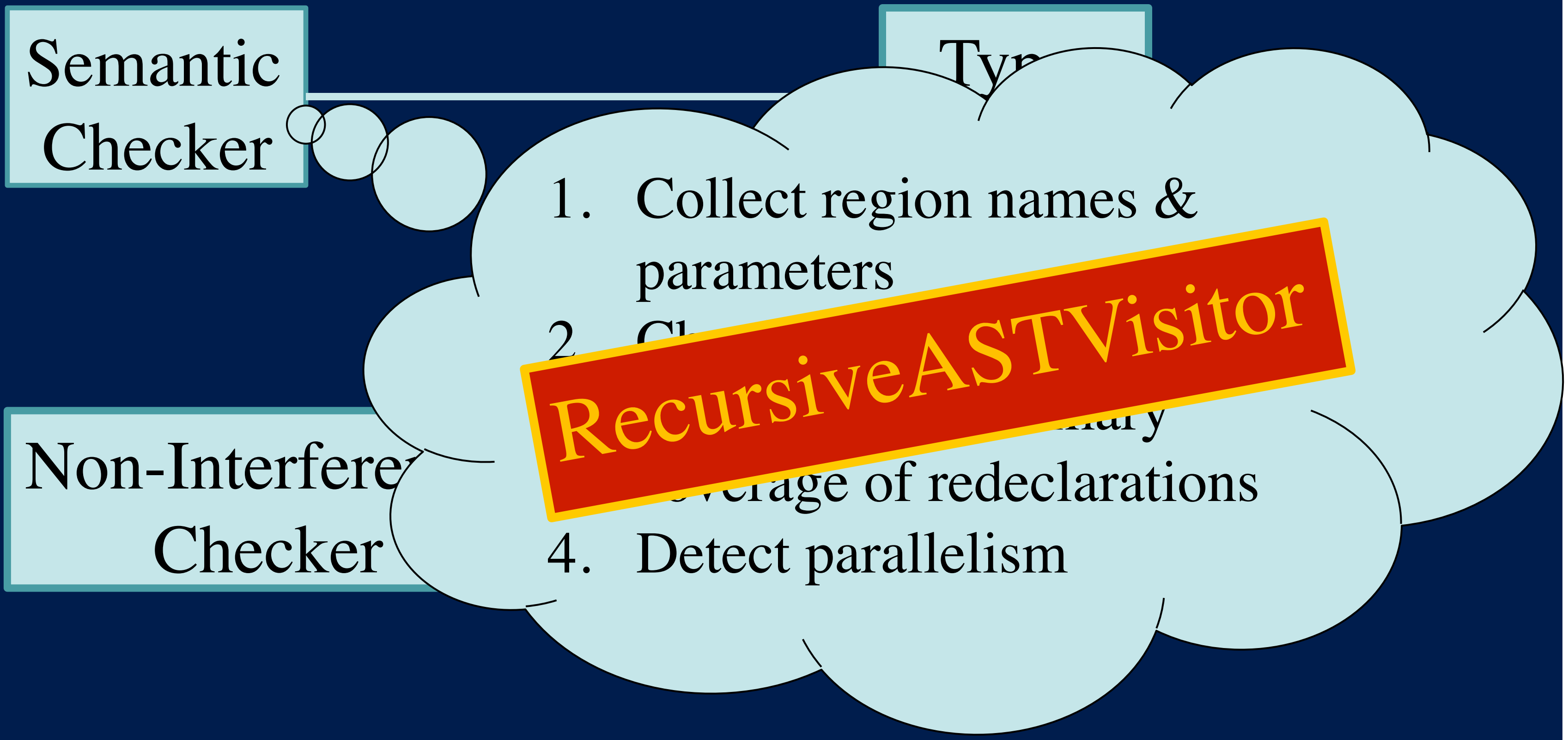
Outline

- What do these ASaP annotations look like/do?

- *How is this ASaP checker designed/built?*
  - *Architecture, Implementation, …*

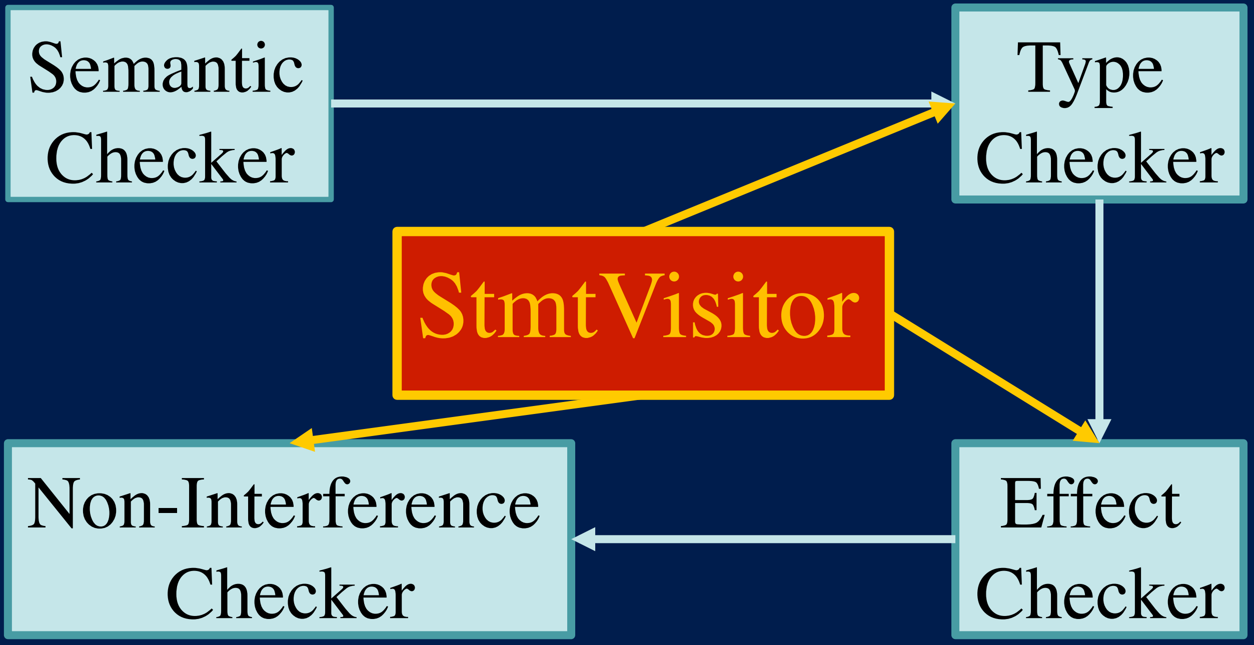
- Nice prototype! What else will come standard?
  - Expressiveness (patterns & parallel APIs)
  - Annotation Inference

ASaP Checker Architecture



1. Collect region names & parameters
2. Ch
mary
overage of redeclarations
4. Detect parallelism

ASaP Checker Architecture



illinois.edu

ASaP Checker Architecture

lib/StaticAnalyzer/Checkers

- Does not rely on any of the analyses
  - Could rewrite as clang plugin or using tooling infrastructure
- Passes use RecusiveASTVisitor & StmtVisitor
- Custom Symbol Table
  - types extended w. regions & effects
- 36 files, 6174 LOC

Contributions to Clang *(by Michael Han)*

- C++11 attribute patches
  - EmptyDecl AST node (+ Attributes)
  - PR14922: Printing Attributes
  - Improve diagnostics for C++11 Attributes
  - C++11 [dcl.attr.grammar] p4
  - Updates to Clang Attribute documentation
  - …
- 1 bug & fix
  - RAV visit parameter declarations of implicit fns

Scope/Limitations

Non-Interference limited to structured parallelism

- Fork-Join

Assumptions about Program

- Type Safe
- Memory Safe

C++11

- Attributes not allowed on:
  - function calls, template type parameters
- #includes & fwd decls can break soundness
  - Can’t guarantee cross TU declaration consistency

Current Implementation Limitations

- We don’t analyze stdlibc
- Not supported yet (i.e. we don’t analyze & warn about)
  - Type-unsafe casts (no warning produced – flag?)
  - Function Pointers (need complex type annotation)
  - Variadic functions
  - “Non-uniform” unions (e.g., {int x; int *p;})
  - Lambdas
  - Bitfields
  - …

Wishlists

- Clang
  - Easier integration with clang driver
    - *Invoking Custom Checker, Checker Specific flags*
  - Pluggable Type System Support
    - *Dream on ☺*

- C++
  - Attributes on expressions
  - Attributes on template parameters
  - #include be gone! Modules

Outline

- What do these ASaP annotations look like/do?

- How is this ASaP checker designed/built?
  - Architecture, Implementation, …

- *Nice prototype, what else will come standard?*
  - *Expressiveness (patterns & parallel APIs)*
  - *Annotation Inference*

illinois.edu

Parallel APIs

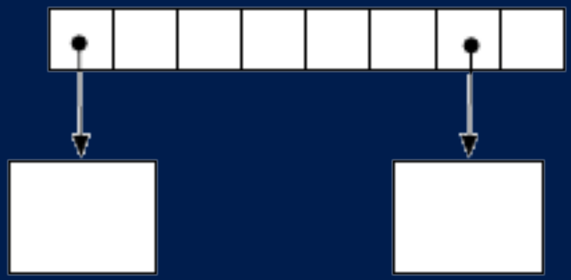
- tbb::
  - parallel_for, parallel_reduce, parallel_scan
  - ...
- concurrent::
  - all of the above
- ...

- Annotation for common parallelism API pattern?
  - Not require implementing support for each API

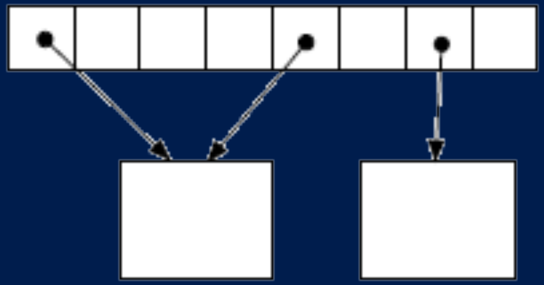
Checked Library API Annotations

- API Context Annotations / Usage Constraints
  - E.g., f(T x<R1>) || f(T x<R2>) iff R1:*#R2:*
- Access permissions
  - E.g., writes Global/Static

Index Parameterized Arrays

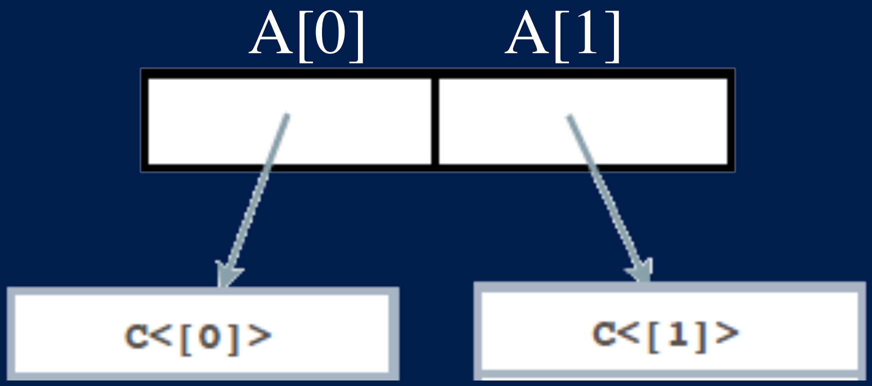


Disjoint



Non-Disjoint

C<R:[i]>* A[]<R:[i]> = new C<R:[i]>* [2];



Annotation Inference

1. Partial: Effect Inference [ASE09]
   - Interprocedural
   - Solving effect summary coverage constraints.

2. Full: Region & Effect Inference [In progress]
   - Solving 3 types of constraints at the same time:
     - *Non-interference, effect summary coverage, subtype*

Annotation Inference: Example

```
class ListNode<P> {
  double Value <π>
  ListNode *Next <π1, π2>;
  void setAllTo(double V) E {

    parallel_invoke(
      { Value = V; },
      { if (Next) Next->setAllTo(V); });
}  }  };
```

Annotation Inference: Example

```
class ListNode<P> {
    double Value <π>
    ListNode *Next <π1, π2>;
    void setAllTo(double V) E {

        parallel_invoke(
            { Value = V; },                              // writes π
            { if (Next) Next->setAllTo(V); });// reads π1, E[P←π2]
}  }  };
```

Annotation Inference: Example

```
class ListNode<P> {
  double Value <π>
  ListNode *Next <π1, π2>;
  void setAllTo(double V) E {
                                  {rd π1, wr π, E[P←π2] }⊆ E
    parallel_invoke(
      { Value = V; },                         // writes π
      { if (Next) Next->setAllTo(V); });// reads π1, E[P←π2]
}  }  };
```

Locks

- Start with scoped locks
  - Take advantage of *Thread Safety Annotations*
  - May need extensions to reason about aliasing

- Extend to other locking patterns
  - E.g., hand-over-hand

ASaP Clang Checker: Conclusions

- Strong Static Guarantees
  - Expressive annotations via C++11 attributes
- Functional Basic Prototype
- Much more functionality to come
  - Annotation Inference
  - Library API contracts
  - …

Contact

- Alexandros Tzannes:
  - atzannes@illinois.edu
  - atzannes@gmail.com