Dark Information and Graph Grammars in Automated Software Design

Don Batory
w. Rui Gonçalves, Bryan Marker, Janet Siegmund
Department of Computer Science
University of Texas at Austin
Good Morning!

• Last century, I worked in DBMSs implementations
• It fundamentally shaped my view of software development

• Quick survey:
  • How many of you understand Relational Query Optimization?
  • How many of you understand Rule-Based Relational Query Optimization?
    » unequivocal success story

• You’re in for a treat...
My Hobby is Reading Popular Physics

- Roughly 80 percent of the mass of our universe is made up of material that scientists cannot directly observe.
  - dark matter emits no light or energy.
  - Scientists know it exists to explain motions of galaxies, unexpected bending of light in empty space.
  - Dark matter is poorly understood.
Dark Matter and Dark Knowledge

- Roughly 80 percent of the mass of our universe is made up of material that scientists cannot directly observe
  - **dark matter** emits no light or energy
  - scientists know it exists to explain motions of galaxies, unexpected bending of light
  - dark matter is poorly understood

- Software design is a series of decisions whose effects are seen in programs, but are not directly observable
  - **dark knowledge** is fleeting, implicitly known by programmers, not present in source code
  - we know it exists to explain programs
  - dark knowledge poorly understood
Why Dark Knowledge is Important

- Software design starts with a spec
- With domain and engineering knowledge, series of decisions maps spec to a program
- Trail of decisions vanishes, losing vital information to maintain and evolve program correctly
Connection to Language Design

- Instance of these ideas
- Language & implementation is a series of decisions whose effects can be seen, but are not explicitly encoded
- Same problems as other software
- Lose vital design information needed for maintenance and evolution
Fundamental Problem that Resurfaces

- Variant is well-known to automated software design community
  - 1992 CACM Baxter’s Design Maintenance System

- Encode “dark knowledge” as meta-policies for self-adaptive and self-managing software
  - design decisions are explicitly encoded in software
  - can be made automatically (or perhaps human in the loop)

- Müller claims this is a very different approach to software development
Challenge: How to Capture Dark Information?

- Design by Transformation (DxT) is an effective way: encode dark information as xforms

- Each of these transformations are typically performed manually

- If each transformation is automatic, you could automate program design & development
  - can be done for well-understood domains

- This is the mindset of DxT
To Me, Program Derivation is...

- Analogous to grammars and parse trees (and deductive program synthesis)

- Grammar $\mathcal{G}$ for a language $\mathcal{L}$ is a set of building blocks called productions
  - rules to describe how to form sentences (in the language’s alphabet) that are valid according to the language’s syntax

- Set of derivable sentences is language $\mathcal{L}$
  - derivation of $S$ is “proof” that $S \in \mathcal{L}$
Connection to Dark Knowledge

- A parse tree encodes the not-so dark knowledge used in deriving a sentence

- There’s no evidence of these productions in the sentence itself
  - only their after-effects

- When parse trees are coupled with extra info to produce an AST, can automate program manipulations, such as program refactoring
  - knowledge of a sentence’s derivation is key to automate its manipulation
How I Think about Program Derivation

- Typical programming languages and their grammars are 1D; sentences are 1D strings.

- Think in $n$-dimensions, let $n = 2$.

- A program is a graph of computations – think of dataflow graphs where nodes are computations and edges indicate the flow of data.

- I suspect there is a theoretical distinction 1D and $n$D grammars (to be discussed later).

Different colors mean different functionalities.
Grammars $\rightarrow$ Graph Grammars

• Generalizations of Chomsky string grammars

• Extends concatenation of strings to a “gluing” of graphs

• Productions of the form:

\[ Graph_{left} \rightarrow Graph_{right} \]

• Derivations are of the form:

\[ Graph_{initial} \Rightarrow^* Graph_{final} \]
Example Graph Derivation

$\text{Graph}_{\text{initial}}$

$\text{Graph}_{\text{final}}$
If You Are a Graph Grammar Enthusiast...

• DxF grammars follow the:

Algebraic (double-pushout) hyper-graph grammar approach

• Unfortunately...
  
a) above is more general than we need

b) yet to find a non-obvious theorem that gives us any leverage
what started all of this: the design of

**UPRIGHT: STATEFUL CRASH FAULT TOLERANT SERVER**

ABSTRACT

The UpRight library seeks to make Byzantine fault tolerance (BFT) a simple and viable alternative to crash fault tolerance for a range of cluster services. We demonstrate UpRight by producing BFT versions of the Zookeeper lock service and the Hadoop Distributed File System (HDFS). Our design choices in UpRight favor simplifying adoption by existing applications; performance is a secondary concern. Despite these priorities, our BFT Zookeeper and BFT HDFS implementations have performance comparable with the originals while providing additional robustness.

occur with some regularity and can have significant consequences [2, 7, 30] and although the research community has done a great deal of work to improve BFT technologies [1, 8, 11, 12, 18, 19, 32-34], deployment of BFT replication remains rare.

We believe that for practitioners to see BFT as a viable option they must be able to use it to build and deploy systems of interest at low incremental cost compared to the CFT systems they build and deploy now; BFT systems must be competitive with CFT systems not just in terms of performance, hardware overheads, and availability, but also in terms of engineering effort.
Initial Graph of Upright

- As only ~15 people on earth understood the design (certainly not me), very challenging reverse engineering task
Initial Graph of Upright

- As only ~15 people on earth understood the design (certainly not me), very challenging reverse engineering task
Initial and Final Graphs for Synchronous Crash Fault Tolerant Servers

simple – gets a LOT more complicated
What Does This Mapping Do (Offhand)?

Removes **single points of failure (SPOFs):** box that if it fails would cause the server abstraction to fail.

*simple – gets a LOT more complicated*
Derivation
Derivation
Derivation
Derivation
Pause – Where We Are...

We have gone from 3 SPOFs to 8 SPOFs – but these are easier to remove
Derivation
Initial and Final Graphs
Essential Idea

- Graph grammar $GG$ is a pair:
  \[ GG = (g, P) \]
  - $g$ is a starting graph; $P$ is a set of graph productions
  - $L(g, P)$ = set of graphs that can be derived by applying the rules in $P$ to $g$
  - Generalize: $g \in large\_set\_of\_graphs$
Distinctions Between 1D & 2D Grammars

1. OK if there are multiple graph derivations
   - multiple ways to derive a program

2. 1D grammar productions (rewrites) don’t need a proof of correctness
   - what’s to prove??
   - should prove graph rewrites correct

3. Resulting designs & derivations are Correct By Construction
that's cute but...

WHO CARES?
Practical Motivation

- **Software Engineering (SE)** largely aims at techniques, tools to aid masses of programmers whose code is used by hordes
  - these programmers need all the help they can get

- Many domains where programming tasks are so difficult, only a few expert programmers – and their code is used by hordes
  - these experts need all the help they can get too
Our Focus is on...

• Dataflow domains:
  – nodes are computations
  – edges denote node inputs and outputs

• General: Virtual Instruments (LabVIEW), applications of streaming languages...

• Our domains:
  • Crash Fault-Tolerant File Servers
  • Parallel Relational Join Algorithms
  • Distributed-Memory, Sequential, and Shared-Memory Dense Linear Algebra Kernels
Approach

- Domain experts produce “Big Bang” spaghetti diagrams (dataflow graphs)

- Better: derive dataflow graphs from domain knowledge

- Prove correctness of each step:

  **Correct By Construction**

- Parallelization of Hash Joins in the Gamma Database Machine (~1990)
DENSE LINEAR ALGEBRA
Generating Dense Linear Algebra Kernels

- Portability of DLA kernels is a problem:
  - may not work – distributed memory kernels don’t work on sequential machines
  - may not perform well
  - choice of algorithms to use may be different
  - cannot “undo” optimizations and reapply others
  - if hardware is different enough, code kernels from scratch
Why? Because Performance is Key!

• Applications that make DLA kernel calls are common to scientific computing:
  • simulation of airflow, climate change, weather forecasting

• Applications are run on extraordinarily expensive machines
  • time on these machines = $$
  • higher performance means quicker/cheaper runs or more accurate results

• Application developers naturally want best performance to justify costs
Distributed Memory DLA Kernels

- Deals with SPMD (Single Program, Multiple Data) architectures
  - same program is run on each processor but with different inputs
- Expected operations to support are fixed – but with lots of variants

<table>
<thead>
<tr>
<th>BLAS3</th>
<th># of Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed DLA Kernels

- Deals with SPMD (Single Program, Multiple Data) architectures
  - same program is run on each processor but with different inputs
- Expected operations to support are fixed – but with lots of variants

<table>
<thead>
<tr>
<th>BLAS3</th>
<th># of Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gemm</td>
<td></td>
</tr>
<tr>
<td>Hemm</td>
<td></td>
</tr>
<tr>
<td>Her2k</td>
<td></td>
</tr>
<tr>
<td>Herk</td>
<td></td>
</tr>
<tr>
<td>Symm</td>
<td></td>
</tr>
<tr>
<td>Syr2k</td>
<td></td>
</tr>
<tr>
<td>Tmm</td>
<td></td>
</tr>
<tr>
<td>Trsm</td>
<td></td>
</tr>
</tbody>
</table>
Distributed DLA Kernels

- Deals with SPMD (Single Program, Multiple Data) architectures
  - same program is run on each processor but with different inputs
- Expected operations to support are fixed – but with lots of variants

<table>
<thead>
<tr>
<th>BLAS3</th>
<th># of Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gemm</td>
<td>12</td>
</tr>
<tr>
<td>Hemm</td>
<td>8</td>
</tr>
<tr>
<td>Her2k</td>
<td>4</td>
</tr>
<tr>
<td>Herk</td>
<td>4</td>
</tr>
<tr>
<td>Symm</td>
<td>8</td>
</tr>
<tr>
<td>Syr2k</td>
<td>4</td>
</tr>
<tr>
<td>Trmm</td>
<td>16</td>
</tr>
<tr>
<td>Trsm</td>
<td>16</td>
</tr>
</tbody>
</table>
12 Variants of Distributed Gemm

\[ C := \alpha \text{op}(A) \text{op}(B) + \beta C \]

- Where: \( \text{op}(A) = \begin{cases} A & \text{Normal} \\ A^T & \text{Transposed} \end{cases} \) and: \( \text{op}(B) = \begin{cases} B & \text{Normal} \\ B^T & \text{Transposed} \end{cases} \)

- Specialize implementation for distributed memory based on \( A, B, \) or \( C \) is largest

\[ 4 \times 3 = 12 \]

- The same for other operations
Further

- Want to optimize “LAPACK-level” algorithms which call DLA and BLAS3 operations:
  - solvers
  - decomposition functions (ex. Cholesky factorization)
  - eigenvalue problems

- Generate high-performance algorithms for these operations too

- Our work mechanizes the decisions of experts on van de Geijn’s FLAME project, leveraging Elemental Distributed DLA library (Poulson)
  - rests on 20 years of polishing elegant layered designs of DLA libraries and their computations
## Performance Results

<table>
<thead>
<tr>
<th>Machine</th>
<th># of Cores</th>
<th>Peak Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argonne’s BlueGene/P (Intrepid)</td>
<td>8,192</td>
<td>27+ TFLOPS</td>
</tr>
<tr>
<td>Texas Advanced Computing Center (Lonestar)</td>
<td>240</td>
<td>3.2 TFLOPS</td>
</tr>
</tbody>
</table>

- DxT automatically generated & optimized Elemental code for BLAS3 and Cholesky FLAME algorithms
- Benchmarked against ScaLAPACK
  - vendors standard option for distributed memory machines; auto-tuned or manually-tuned
  - only alternative available for target machines
BLAS3 Performance on Intrepid

![Bar graph showing performance (GFLOPS) of various BLAS3 operations on Intrepid, comparing ScaLAPACK and DxTer.](chart.png)

Performance (GFLOPS)

- Gemm NN, Gemm NT, Gemm TN, Gemm TT, Symm LL, Symm RL, Symm LU, Symm RU, Syr2k LN, Syr2k LT, Syr2k UN, Syr2k UT, Syrk LN, Syrk LT, Syrk UN, Syrk UT, Trmm LLNN, Trmm RLNN, Trmm LLLN, Trmm LLTN, Trsm LLNN, Trsm LLTN, Trsm LLLN, Trsm LLTN, Trsm LUNN]
Cholesky Performance on Lonestar

![Graph showing performance of Cholesky algorithms on Lonestar]

- **Performance (GFLOPS)**
- **Matrix Dimension Size**
- **DxT**
- **ScaLAPACK**
We (Bryan) Found

- Error(s) in Elemental Library
- Several instances where the Expert forgot to apply an optimization
- Or used the wrong algorithm

DxT-Generated Code is Shipped today with Elemental which is used in National Laboratories and Industry
# State-of-Art and Our Group's Vision

<table>
<thead>
<tr>
<th>State-of-Art</th>
<th>Our Group's Vision</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dense linear algebra libraries exist as code.</td>
<td>• Don’t develop linear algebra libraries in specific language for specific architecture by hand.</td>
</tr>
<tr>
<td>• Rewritten manually as architecture du jour changes, and it changes rapidly.</td>
<td>• Instead, develop tools, techniques, and theories to encode algorithms, expert knowledge, and information about target architectures to generate desired code libraries automatically.</td>
</tr>
<tr>
<td>• Library development lags far behind architecture development.</td>
<td>• performs ≥ than hand-coded</td>
</tr>
<tr>
<td>• Attaining sub-optimal performance on latest &amp; greatest machines carries a big price.</td>
<td>• eliminates mistakes by experts</td>
</tr>
<tr>
<td></td>
<td>• faster, cheaper, better</td>
</tr>
</tbody>
</table>
DxT Limitations and Legacy

- Not limited to DLA
  - components can have state
  - DxT was designed originally to explain stateful designs (ex. Upright)

- Apply to any data-flow domain where mechanization of rote and/or high-performance code is needed
  - huge numbers and diversity of such domains (see LabVIEW)

- Other projects with similar goals and approaches:
  - Rule-based Relational Query Optimization (circa 1990)
  - SPIRAL (linear xforms, signal proc. 2000-)
  - Tensor Contraction Engine (2005-)
  - Built-To-Order BLAS (2008-)
I told you what we been doing. now i’ll tell you details on

**HOW DXT WORKS...**
# DxT MetaModel Basics

<table>
<thead>
<tr>
<th><strong>Interface</strong> ($i$)</th>
<th>$in_1 \rightarrow i \rightarrow out_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong> ($\alpha$)</td>
<td>$in_1 \rightarrow \alpha \rightarrow out_1$</td>
</tr>
<tr>
<td><strong>Primitive Implementations</strong> ($\mathcal{P}$)</td>
<td>$in_1 \rightarrow \mathcal{P} \rightarrow out_1$</td>
</tr>
</tbody>
</table>
Rewrite Rules: Refinements

- Replaces an interface with an implementation

- Rewrites define domain specific theorems

- Rewrites are not context free
  - interfaces have preconditions (standard)
  - RHS have additional preconditions (unusual for classical refinements)
Example: Sorting Streams

\[ in_1 \rightarrow \text{sort} \rightarrow out_1 \]

\[ in_1 \rightarrow \text{quick} \rightarrow out_1 \]

\[ in_1 \rightarrow \text{merge} \rightarrow out_1 \]

\[ in_1 \rightarrow \text{hash} \rightarrow \text{sort} \rightarrow \text{merge} \rightarrow out_1 \]

\[ in_1 \rightarrow out_1 \quad ; \text{stream already in desired order} \]
When to Apply Rewrites?

- **Liskov Substitution Principle (LSP)**
  - hallmark of OO design
  - if \( A \) is a subtype of \( I \) then objects of type \( A \) can be substituted for objects of type \( I \) without altering program correctness

| \( pre (A) \Rightarrow pre(I) \) | preconditions can’t be strengthened |
| \( post (I) \Rightarrow post(A) \) | postconditions can’t be weaker |

- This clearly is not what is going on with graph rewrite rules – another principle involved
Perry Substitution Principle (PSP)

• Inscape (1988), Perry said box $A$ is **upward compatible** with box $I$ if:

| $pre (A) \Rightarrow pre(I)$ | preconditions can be stronger |
| $post (A) \Rightarrow post(I)$ | postconditions can’t be weaker |

• $A$ requires and provides at least the same as $I$

• Important, practical alternative to LSP that dominates the DxT universe
  • take into account local conditions surrounding an interface to perform rewrite
  • essential to graph rewriting
Rewrite Rules: Optimizations

• Replaces a graph with another graph

\[ \text{in}_1 \rightarrow a_1 \rightarrow \text{out}_1 \quad \Rightarrow \quad \text{in}_1 \rightarrow a_2 \rightarrow \text{out}_1 \]

or

• Abstracts a graph to an interface which is then refined

\[ \text{in}_1 \rightarrow i \rightarrow \text{out}_1 \quad \Leftrightarrow \quad \text{in}_1 \rightarrow a_1 \rightarrow \text{out}_1 \]

\[ \Rightarrow \quad \text{in}_1 \rightarrow a_2 \rightarrow \text{out}_1 \]

• Optimizations are essential for high-performance and practical dataflow designs
Optimizations Have No Counterpart in 1D Grammars

$A : \ a \ B \ c \ | \ \ldots \ ;$

$B : \ b \ | \ \ldots \ ;$

• Sentence of this grammar is $abc$

• Suppose composition $bc$ implements interface $Z$

$Z : \ bc \ | \ q \ ;$

• Domain experts know that $bc$ is inefficient and can be replaced by $q$ which is faster

• Abstract sentence $abc$ to $aZ$

• Refine $aZ$ to $aq$

• Essence of optimization – dissolve modular boundaries to expose inefficiencies

• Fundamental difference (mentioned earlier) between 1D and $n$D grammars: refinements and optimizations part of $n$D grammars
Optimizations Across Modular Boundaries
Very Familiar!

• Very similar to optimization techniques in compilers for functional languages

• Except:
  • graphs not trees
  • grammars used to derive software designs that are correct-by-construction
  • software engineers (a.k.a domain experts) should be the ultimate purveyors of this knowledge – not tool builders
What Else?

- Standard “trick” from functional programming language and static analysis playbooks:

  Abstract Interpretation

- Suppose we want to know the performance of an algorithm:

\[ i \rightarrow \sigma \]

\[ i_\pi \rightarrow \sigma_\pi \]

- Standard Interpretation

- Performance Interpretation
What Else?

• Standard “trick” from functional programming languages playbook:

Abstract Interpretation

• Suppose we want to know the post conditions of an algorithm:

Standard Interpretation

POST Interpretation
What Else?

• Standard “trick” from functional programming languages playbook:

Abstract Interpretation

• Suppose we want to evaluate the preconditions of the boxes of an algorithm

**Compose interpretations:** \( Pre \cdot Post \)
In MDE and Automatic Programming Terms...

- Recall Slide:

  **Essential Idea**

  - Graph grammar $GG$ is a pair:
    \[
    GG = (g, P)
    \]
  - $g$ is a starting graph; $P$ is a set of graph productions
  - $L(g, P)$ = set of graphs that can be derived by applying the rules in $P$ to $g$
  - Generalize: $g \in \text{large set of graphs}$

- Give us an abstract design $g$ (PIM), using our productions $P$ we can:
  - generate space of all algorithms (PSMs), estimate performance of each, and choose cheapest
  - how we generate efficient DLA algorithms – **Rule-Based Relational Query Optimization**
BEST PART OF DxT...
Simple Enough to Teach Undergraduates

• Our thought: when you have a “proof” or “derivation” of a design
  • you’ve hit the jackpot
  • you’ve turned dark knowledge into white (explicit) knowledge

• Response

sle-62
David Letterman Countdown Order: Top 3

1. Work lacks motivation

2. Why will you succeed where others have not?

3. Refinement is not a transformation
Tried User Study (ICPC 2012)

- To 1st group we presented Big-Bang design for Gamma; presented Gamma’s derivation to 2nd
  - posed questions that we graded
  - result: no difference – no obvious benefit to derivations
- Perhaps Gamma was too simple – tried Upright
  - repeated experiment
  - result: no difference – no obvious benefit to derivations
Idea!

- Maybe students (and referees) have no experience developing software in this manner
- It is not that they can’t understand the ideas
- And perhaps they have no domain-knowledge
- They can’t appreciate what we are telling them
- Second-hand evidence for this last conjecture
What are Refinements for DGEMM?

- Who would know this?
  - A1: a few people that have deep knowledge of DLA
  - A2: very few of them know all the rules and how they are applied
Grammars vs. Graph Grammars

• Easy to interpret a 1D grammar
  • there is very little that you have to know

• Graph grammars – especially how they are used in DxT – are quite different
  • you need deep knowledge of a domain to appreciate most graph rewrites
  • most people do not have such knowledge; very few do

• Why?
  • as Cordell Green says, “it takes effort”

sle-67
Our Conjecture

Majority of people in SE and DLA have no experience building or thinking about software in a DxT manner...

They can’t appreciate what we are telling them -- Looks like too many efforts before that have “failed”

(because they don’t know it ever succeeded)
Our Next User Study

- Gave project to an undergraduate class of students
  - had them build Gamma given its derivation
- Then asked them to compare this with “big bang” approach which directly implements the spaghetti diagram (final design)
- 28 third-year students Fall 2012
Comprehension

- Do you think the structured way that DxT imposes gives you a deeper understanding of Gamma's design than you would get by not using it and doing it your own way?

Over 80% said DxT gives a deeper understanding.
Modification

- Do you think it would be easier or more difficult to modify Gamma with DxT compared to a big-bang approach?

Over 90% said DxT would make it easier

considerably easier

considerably harder
Recommendation

- Would you recommend your fellow students implementing Gamma using DxE or in a Big-Bang manner?

Almost 90% recommended DxE
Best Part: Comments

I have learned the most from this project than any other CS project I have ever done.

I even made my OS group do a DxT implementation on the last 2 projects due to my experience implementing Gamma.

Honestly, I don't believe that software engineers ever have a source (to provide a DxT explanation) in real life. If there was such a thing we would lose our jobs, because there is an explanation which even a monkey can implement.

It's so much easier to implement (using DxT). The big-bang makes it easy to make so many errors, because you can't test each section separately. DxT might take a bit longer, but saves you so much time debugging, and is a more natural way to build things. You won't get lost in your design trying to do too many things at once.
In Retrospect, Not Surprising

- October 2003 NSF Science of Design Workshop

- Fred Brooks (1999 Turing Award) summarized conclusions of his working group:

  We don’t know what we’re doing and we don’t know what we’ve done!

- Paraphrasing Edsger Dijkstra (1972 Turing Award):

  Have you noticed that there are child prodigies in mathematics, music, and gymnastics, but none in human surgery?
Bottom Line

“it takes effort”

Domain knowledge, experience and understanding how to codify knowledge in a machine-processable way is everything to automated software design.
CONCLUSIONS
Recap

• We are geniuses at making the simplest things look complicated; finding the underlying simplicity is the challenge

• We are also geniuses at making critical design knowledge “dark”; whitening knowledge is another separate challenge

• It takes effort to study legacy applications and domain of such applications to mine out its fundamental “identities” or rewrite rules
  • key to automation
  • key to correct-by-construction
  • key to moving undergraduate education on software design from hacking to modernity
Software Language Engineering

• Is only a tiny part of what it will become

• Languages will be foundation for automated software development
  • based on graph-grammars, not Chomsky String grammars
  • sentences define complex programs
  • programs can be optimized automatically
  • removing the burden of rote, tedious, difficult, and error-prone activities
  • scales domain expertise from a few people to everyone
  • ultimately modernize undergraduate curriculums in software design

• Given you a glimpse of the future and how your work will contribute to its foundation
Software Language Engineering

• Is only a tiny part of what it will become

• Languages will be foundation for automated software development
  • based on graph-grammars, not Chomsky String grammars
  • sentences define complex programs
  • programs can be optimized automatically
  • removing the burden of rote, tedious, difficult, and error-prone activities
  • scales domain expertise from a few people to everyone
  • ultimately modernize undergraduate curriculums in software design
Thank You!

Questions?
Index

- Intro
- Upright
- Who Cares
- How DxT Works
- Best Part of DxT