Higher-Order Transformation and the Distributed Data Problem

Victor Winter
Outline

1. Historic Origins
2. Strategic Programming
   Observing the Application of Strategies to Terms
3. The Distributed Data Problem
   Solutions to the DDP
4. HATS
   TL - A Language for Expressing Higher-Order Transformations
The Word Problem

Definition

Given

- A term algebra $\mathcal{T}(\mathcal{F}, \mathcal{X})$.
- A set $E$ whose elements are (equational) identities of the form $s \approx t$ where $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.
- Two (arbitrary) words $w_1$ and $w_2$ belonging to $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Decide whether $E \vdash w_1 \approx w_2$ using only the rules of equational logic.
The Rules of Equational Logic

\[(s \approx t) \in E \]
\[\frac{}{E \vdash s \approx t} \]

\[
\begin{align*}
E \vdash t \approx s & \quad E \vdash s \approx t & \quad E \vdash t \approx u \\
\hline
E \vdash t \approx t & \quad E \vdash s \approx t & \quad E \vdash s \approx u \\
E \vdash s \approx t & \quad E \vdash t \approx u \\
\hline
E \vdash \sigma(s) \approx \sigma(t) & \quad E \vdash s_1 \approx t_1 \ldots E \vdash s_n \approx t_n \\
\hline
E \vdash f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) & 
\end{align*}
\]
The Axioms of a Boolean Algebra

Commutativity
\[ \text{or}(x, y) \approx \text{or}(y, x) \]
\[ \text{and}(x, y) \approx \text{and}(y, x) \]

Distributivity
\[ \text{or}(x, \text{and}(y, z)) \approx \text{and}(\text{or}(x, y), \text{or}(x, z)) \]
\[ \text{and}(x, \text{or}(y, z)) \approx \text{or}(\text{and}(x, y), \text{and}(x, z)) \]

Identity
\[ \text{or}(x, \text{false}) \approx x \]
\[ \text{and}(x, \text{true}) \approx x \]

Complement
\[ \text{or}(x, \text{not}(x)) \approx x \]
\[ \text{and}(x, \text{not}(x)) \approx x \]
An Example of the Word Problem

Use equational logic together with the axioms from boolean algebra $E_{\text{bool}}$ to show that: $E_{\text{bool}} \vdash \text{or}(a, a) \approx a$.

\[
\begin{array}{ccc}
\text{a} & \xleftarrow{\text{Ident}} & \text{or}(a, \text{false}) & \text{Identity} \\
& \xleftarrow{\text{Comp}} & \text{or}(a, \text{and}(a, \text{not}(a))) & \text{Compliment} \\
& \xrightarrow{\text{Dist}} & \text{and}(\text{or}(a, a), \text{or}(a, \text{not}(a))) & \text{Distributivity} \\
& \xrightarrow{\text{Comp}} & \text{and}(\text{or}(a, a), \text{true}) & \text{Compliment} \\
& \xrightarrow{\text{Ident}} & \text{or}(a, a) & \text{Identity}
\end{array}
\]
The Undecidability of the Word Problem

Definition

Combinatory logic is a formalism, developed in the early 1900’s, in which all computable functions can be expressed. In this framework, the following equivalences hold:

\[ E_{CL} \overset{\text{def}}{=} \{ ((S \cdot x) \cdot y) \cdot z \approx (x \cdot z) \cdot (y \cdot z), \ (K \cdot x) \cdot y \approx x \} \]

The question \( E_{CL} \vdash f_1 \approx f_2 \) is undecidable for arbitrary \( f_1 \) and \( f_2 \).
Term Rewriting Systems

Rewriting is a computational framework restricting the computational alternatives present in equational logic. In particular, identities in $E$ are oriented as follows:

Conversion

$$ (s ≈ t ∈ E) ⇒ (s → t ∈ TRS) $$

provided $Var(s) ⊇ Var(t)$. Here $s → t$ is called a rewrite rule and the resulting set of rewrite rules $TRS$ is called a term rewriting system.
A Simple Term Rewriting System

\[
\text{TRS} \overset{\text{def}}{=} \\
\{ \\
\text{mult}(x, \text{add}(y, z)) \rightarrow \text{add(} \text{mult}(x, y, \text{mult}(x, z))\text{)}, \\
\text{mult(} \text{add}(y, z), x \rightarrow \text{add(} \text{mult}(x, y, \text{mult}(x, z))\text{)}, \\
\}
\]
Explicit Control via \( d1 \)

Another Simple Term Rewriting System

\[
\begin{align*}
\text{TRS}' & \overset{\text{def}}{=} \\
\{ & \quad d1(mult(x, add(y, z))) \rightarrow add(d1(mult(x, y)), d1(mult(x, z))), \\
& \quad d1(mult(add(y, z), x)) \rightarrow add(d1(mult(x, y)), d1(mult(x, z))), \\
& \quad d1(integer) \rightarrow integer
\}
\end{align*}
\]
Implementation of TRS' in a Functional Programming Language

datatype math = mult of (math * math)
| add of (math * math)
| num of int;

fun d1( num(x) ) = num(x)
| d1( mult(x,add(y,z)) ) = add( d1(mult(x,y)), d1(mult(x,z)) )
| d1( mult(add(x,y),z) ) = add( d1(mult(x,y)), d1(mult(x,z)) )

(* These Additional Cases Needed for Control *)
| d1( mult(x,y) ) = mult( d1(x), d1(y) )
| d1( add(x,y) ) = add( d1(x), d1(y) );
Abstraction of Control

Term Traversal: Covers all Constructors

\[
\begin{align*}
\text{fun} & \quad \text{BU TRS} (\ num(x) ) & = \text{TRS}(\text{num}(x)) \\
| & \quad \text{BU TRS} (\ mult(x,y) ) & = \text{TRS}(\text{mult}\ (\text{BU TRS } x, \text{BU TRS } y )) \\
| & \quad \text{BU TRS} (\ add(x,y) ) & = \text{TRS}(\text{add}\ (\text{BU TRS } x, \text{BU TRS } y ))
\end{align*}
\]

Term Rewriting System

\[
\begin{align*}
\text{fun} & \quad \text{rules}(\ mult(x,\text{add}(y,z)) ) & = \text{add}(\text{mult}(x,y),\text{mult}(x,z)) \\
| & \quad \text{rules}(\ mult(\text{add}(x,y),z) ) & = \text{add}(\text{mult}(x,y),\text{mult}(x,z))
\end{align*}
\]

(* This Case Needed for Control *)

| rules term & = term;

\[
\text{fun dist2 term = BU rules term}
\]
The Fixed Point Application of Rules

fun FIX ApplyTRS term =
  let
    val term1 = ApplyTRS term
  in
    if term1 = term then term else FIX ApplyTRS term1
  end

fun distribute term = FIX (BU rules) term
A More Complex Example

Problem

Construct a TRS that will rewrite an arbitrary logical formula into its equivalent in conjunctive normal form (CNF).
Normal Forms

In a pure rewriting system, the semantics of exhaustive application is slightly different than the fixed point application given previously. In particular:

• The application of rules belonging to a TRS continues until rules no longer apply – which is different from applying rules until the term no longer changes.

  • Example: A rule like $a \rightarrow a$ will result in nontermination when applied to a term containing $a$.

  • When rules no longer apply one says that a normal form has been reached with respect to the TRS.
A Classical Rewriting Framework

- Is motivated by the desire to mechanize equational reasoning.
- Rewrite rules represent directed equalities.
- The application of rules to terms is *implicit*, exhaustive, and universal.
- And, the user can control rewriting at the term-level by inhibiting matching/unification.

In this framework:
- the existence of unique normal forms is critical (*confluence*), and
- the ability to always reach normal forms (when they exist) is also critical (*termination*).
Observation

- When constructed in a classical fashion, many term rewriting systems that model real world problems are neither confluent nor terminating.

- This motivates the need to introduce more sophisticated control abstractions for specifying the application of rules within a TRS.
In a Classical Strategic Framework

- The application of rules (a.k.a. strategies) to terms is explicit.

- The application of strategies to a single (subject) term can be controlled by:
  - matching - at the term-level
  - conditions - at the rule-level
  - combinators - at the strategy-level

- The application of strategies to term sequences is controlled by iterators.
  - Traversals: top-down left-to-right, bottom-up left-to-right
  - Iterators: FIX, Repeat
What is a Strategy?

**Basis:** A conditional rewrite rule is a strategy.

\[ lhs \rightarrow rhs \quad \text{if condition} \]

**Induction:** An expression composed of strategies, combinators, and iterators is a strategy.

**Remark**

*One could also consider a term to be a strategy much in the same way that a constant is represented, in a term algebra, as a nullary function.*
## Strategic Combinators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_1 + s_2)</td>
<td>The nondeterministic application of (s_1) or (s_2).</td>
</tr>
<tr>
<td>(s_1 \leftrightarrow s_2)</td>
<td>Left-biased conditional application of (s_1) or (s_2).</td>
</tr>
<tr>
<td>(s_1 \rightarrow s_2)</td>
<td>Right-biased conditional application of (s_2) or (s_1).</td>
</tr>
<tr>
<td>(s_1 &lt;; s_2)</td>
<td>Left-to-right sequential application of (s_1) and (s_2).</td>
</tr>
</tbody>
</table>
## Strategic Constants

<table>
<thead>
<tr>
<th>ID</th>
<th>The strategy that can be successfully applied to any term and returns the term unchanged (quite useful in a failure-based system). Note that the application of the composite strategy $s \leftarrow ID$ will always be successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIP</td>
<td>The strategic version of a no-op. This constant is unique to $TL$.</td>
</tr>
</tbody>
</table>
Some Generic First-Order Iterators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TDL{s}</strong></td>
<td>Applies s to a term using a top-down left-to-right traversal.</td>
</tr>
<tr>
<td><strong>BUL{s}</strong></td>
<td>Applies s to a term using a bottom-up left-to-right traversal.</td>
</tr>
<tr>
<td><strong>FIX{s}</strong></td>
<td>The exhaustive application of s.</td>
</tr>
</tbody>
</table>
A First-Order Traversal Primitive

| mapL(s) | Applies $s$, in a left-to-right fashion, to the immediate subterms (i.e., the children) of the term to which it is applied. |

Definition

$$TDL \ s \ = \ s \ <; \ mapL( \ TDL\{s\} )$$

$$BUL \ s \ = \ mapL( \ BUL\{s\} ) \ <; \ s$$
Examples of Strategy Application

<table>
<thead>
<tr>
<th>Rule Label</th>
<th>Rewrite Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$:</td>
<td>$a \rightarrow a$</td>
</tr>
<tr>
<td>$r_1$:</td>
<td>$a \rightarrow b$</td>
</tr>
<tr>
<td>$r_2$:</td>
<td>$b \rightarrow c$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$a$</td>
</tr>
<tr>
<td>$r_1 \leftarrow r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 \leftarrow r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_1 \prec; r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$c$</td>
</tr>
</tbody>
</table>
Definition

Let $s$ denote a first-order strategy and let $t$ denote a term. We write $s(t)$ to denote the first-order application of $s$ to $t$. 
Successful/Unsuccessful Application

- In a strategic framework, standard combinators such as left-biased choice ($<+$) exercise control over rewriting based on an *abstract view* of strategy application. In particular, the application of a strategy to a term is either successful or unsuccessful.

- This perspective assumes the ability to *observe* the successful/unsuccessful nature of strategy application.

- A fundamental question concerns itself with how this observation is made.
### Examples of Successful and Unsuccessful Application

#### Strategies and Terms

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 \leftarrow r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 \leftarrow r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>?</td>
</tr>
<tr>
<td>$r_2 &lt;; r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>?</td>
</tr>
</tbody>
</table>

#### Examples

- $r_0$: $a \rightarrow a$
- $r_1$: $a \rightarrow b$
- $r_2$: $b \rightarrow c$
The unsuccessful application of a strategy $s$ to a term $t$ causes the term $t$ to be rewritten to the term $FAIL$ – which is a constant denoting unsuccessful application. In this case, unsuccessful application can be observed at the term level.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 \leftrightarrow r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 \leftrightarrow r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$FAIL$</td>
</tr>
<tr>
<td>$r_2 \lessdot r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$FAIL$</td>
</tr>
</tbody>
</table>
Identity-based Solution

The unsuccessful application of a strategy $s$ to a term $t$ yields the term $t$. In this case, unsuccessful application cannot be observed at the term level.

$$r_0: a \rightarrow a$$
$$r_1: a \rightarrow b$$
$$r_2: b \rightarrow c$$

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 &lt;+ r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2 &lt;+ r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$a$</td>
</tr>
<tr>
<td>$r_2 &lt;; r_1$</td>
<td>$(a)$</td>
<td>$\Rightarrow$</td>
<td>$b$</td>
</tr>
</tbody>
</table>
Identity versus Failure

The difference between an identity-based system and a failure-based system is highlighted by the following strategy:

\[ TDL\{s1\} \leftrightarrow s2 \]

In general, this strategy is meaningful only in an identity-based system.
Aside: Syntax

- The syntax of terms is typically defined using:
  - A term algebra whose elements are inductively defined.
  - An abstract syntax.
  - A context-free grammar.

- The main point is that the structure of a term is formally defined.

- In the context of this talk, a term and a tree mean the same thing. You can think of a term as a symbolic description of a structure and a tree as a graphical description of that structure.
A Simple Term Algebra

Base Cases

\[
\begin{align*}
i & \in \text{Integer} & c & \in \text{Symbol} & x & \in \text{Var} \\
i & \in \mathcal{L} & c & \in \mathcal{L} & x & \in \mathcal{L}
\end{align*}
\]

General Case

\[
\begin{align*}
t_1 & \in \mathcal{L} & t_2 & \in \mathcal{L} \\
\text{add}(t_1, t_2) & \in \mathcal{L}
\end{align*}
\]

\[
\mathcal{L} = \{ \text{add}(1, 2), \text{add}(1, \text{add}(2, c)), \text{add}(\text{add}(1, b), 3), \ldots \}
\]
Examples

\[ \mathcal{L} = \{ add(1, 2), add(1, add(b, 3)), add(add(1, c), b), \ldots \} \]

\[
\begin{align*}
  r1: & \quad add(1, add(add(1, x), 1)) \rightarrow add(2, add(add(2, x), 2)) \\
  r2: & \quad add(1, x) \rightarrow add(2, x) \\
  r3: & \quad 1 \rightarrow 2
\end{align*}
\]

Let \( t = add(1, add(add(1, b), 1)) \)

<table>
<thead>
<tr>
<th>Application</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1(t) )</td>
<td>( add(2, add(add(2, b), 2)) )</td>
</tr>
<tr>
<td>( r2(t) )</td>
<td>( add(2, add(add(1, b), 1)) )</td>
</tr>
<tr>
<td>( TDL{r2}(t) )</td>
<td>( add(2, add(add(2, b), 1)) )</td>
</tr>
<tr>
<td>( TDL{r3}(t) )</td>
<td>( add(2, add(add(2, b), 2)) )</td>
</tr>
</tbody>
</table>
The Distributed Data Problem

- Given the simple term language (defined previously) consisting of sums involving integers and symbolic constants.

- Consider the following specification:

  **Definition**

  \[
  \text{Swap}(t) = \text{exchange the first two integers in a term } t \text{ without otherwise disturbing the structure of } t.
  \]
Examples of Swap

\[ add(1, 2) \rightarrow add(2, 1) \]
\[ add(1, add(2, 3)) \rightarrow add(2, add(1, 3)) \]
\[ add(1, add(add(2, 3), 4)) \rightarrow add(2, add(add(1, 3), 4)) \]
\[ add(1, add(add(b, c), 2)) \rightarrow add(2, add(add(b, c), 1)) \]
\[ add(add(a, 1), add(2, b)) \rightarrow add(add(a, 2), add(1, b)) \]
A Graphical Perspective
Real World Examples of the DDP

- Variable definition and use (use/def)
  - type checking
  - data-flow analysis
  - variable renaming
  - program slicing

- Optimization:
  - Function inlining
  - Partial evaluation
  - Common subexpression elimination

- Misc: Symbolic resolution in Java class files
The Essence of the DDP

Definition

The DDP arises when there exists semantic relationships between terms that lie beyond the (syntactic) reach of matching or unification.

What language mechanisms can be used to address the issues raised by the DDP?
A standard solution to the DDP makes use of the following:

- **Auxiliary data structures** (e.g., lists) are provided to store information corresponding to a term using some “internal” representation (e.g., integers and tuples).

- Create, access, and update operations are provided for these auxiliary data structures.

- Translational capabilities are provided to enable the specification of translations between term representations and internal representations.

- Strategies (e.g., traversals and rewrite rules) are **parameterizable** on these auxiliary data structures.
A Different Approach

From the perspective of a strategic framework, an approach that is conceptually more elegant is to use existing strategic primitives to store terms of interest. In particular:

- The **strategy** itself is the data structure used to hold terms.

- **Higher-order** strategies can be used to simulate create and update operations.

- **Strategy application** is used to simulate the access operations. That is, these strategies “know how to apply” the information they contain.

- Some **additional combinators** may be useful to permit the specification of more refined control.
The Transient

- Consider a combinator called `transient` that restricts a strategy so that it may only be applied once.

- Operationally, a transient strategy is a strategy that will reduce to the strategy `SKIP` after its first successful application to a term.

- Conceptually, the transient combinator provides a mechanism for strategic reduction.
The Behavior of \textit{SKIP}

In an identity-based system the following equivalences hold:

\begin{align*}
\text{SKIP} & \leftrightarrow s \equiv s \\
 s & \leftrightarrow \text{SKIP} \equiv s \\
\text{SKIP} & <; s \equiv s \\
 s & <; \text{SKIP} \equiv s
\end{align*}
Example 1

r2: 1 → 2

Let \( t = add(1, add(1, 1)) \)

\[
TDL\{r2\}(t) \quad \Rightarrow \quad add(2, add(2, 2))
\]

\[
TDL\{\text{transient}(r2)\}(t) \quad \Rightarrow \quad add(2, add(1, 1))
\]
Example II

\[
\begin{align*}
\text{r1:} & \quad 2 \rightarrow 1 \\
\text{r2:} & \quad 1 \rightarrow 2 \\
\text{swap:} & \quad TDL\{ \text{transient}(r1) \leftrightarrow \text{transient}(r2) \}
\end{align*}
\]

\[
\begin{align*}
\text{swap( add(add(1, b), add(c, 2)) )} &= \Rightarrow \\
\text{add(add(2, b), add(c, 1))}
\end{align*}
\]
Hard-coded Strategies

- For given term it is always possible to hard-code a problem specific transient strategy that will swap the first two integers.

- Hard-coded strategies

\[ \begin{align*}
\text{r1:} & \quad 3 \rightarrow 7 \\
\text{r2:} & \quad 7 \rightarrow 3 \\
\text{swap:} & \quad TDL\{ \text{transient(r1)} \implies \text{transient(r2)} \} \\
\end{align*} \]

\[
\begin{aligned}
\text{swap( add(add(7, b), add(3, 3)) )} & \Rightarrow \\
\text{add(add(3, b), add(7, 3))}
\end{aligned}
\]
Higher-Order Strategies

- The goal of a second-order strategy is to dynamically create a first-order strategy that can then be applied to a term.

- Consider the following second-order rewrite rule:

\[ int_1 \rightarrow int_2 \rightarrow int_1 \]

- Assume that, in the above rule, the variables \( int_1 \) and \( int_2 \) are typed in the sense that they can only be matched with integer values.
Definition

Let $s$ denote a higher-order strategy and let $t$ denote a term. We write $s[t]$ to denote the higher-order application of $s$ to $t$. 
Example

Given the following:

\[ r : \text{int}_1 \rightarrow \text{int}_2 \rightarrow \text{int}_1 \]

Let \( t = \text{add}(2, \text{add}(b, 2)) \)

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Term</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>[1]</td>
<td>( \text{int}_2 \rightarrow 1 )</td>
</tr>
<tr>
<td>( \text{TDL}{ r[1] } )</td>
<td>(t)</td>
<td>( \text{add}(1, \text{add}(b, 1)) )</td>
</tr>
<tr>
<td>( \text{TDL}{ \text{transient}(r[1]) } )</td>
<td>(t)</td>
<td>( \text{add}(1, \text{add}(b, 2)) )</td>
</tr>
</tbody>
</table>
Higher-Order Traversal

- Consider the following strategy definitions:
  
  \[ r_1 : \text{int}_1 \rightarrow \text{transient} (\text{int}_2 \rightarrow \text{int}_1) \]
  
  \[ r_2 : \text{transient}(r_1) \leftrightarrow \text{transient}(r_1) \]

- What is the meaning of the following strategic expression:

  \[ tdl\{ r_2 \} \]

- We would like \( tdl\{ r_2 \}[ \text{add}(1, \text{add}(b, 2)) ] \) to produce the following result:

  \[ \text{transient}(\text{int}_2 \rightarrow 1) \rightarrow \text{transient}(\text{int}_2 \rightarrow 2) \]

- To accomplish this we need to lift the concept of traversal to the higher-order.
Higher-Order Traversal

- Let us consider the strategic expression $tdl\{r\}[t]$ where $r$ is a higher-order strategy.
- Let $n$ denote the number of terms encountered during the $tdl$ traversal of the term $t$.
- Let $s_i$ denote the result of applying $r$ to the $i^{th}$ term in $t$ as encountered by the $tdl$ traversal.
- Let $\oplus$ denote a binary strategic combinator (e.g., $<+\rangle$).
- Under these assumptions an abstract interpretation for the result of $tdl\{r\}[t]$ would be:

$$S_1 \oplus S_2 \oplus \cdots \oplus S_n$$
Higher-Order Traversal

This suggests the following kinds of traversals.

<table>
<thead>
<tr>
<th>Traversal</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rcond_tdl</code></td>
<td>A TDL traversal where $\oplus \stackrel{\text{def}}{=} \rightarrow$</td>
</tr>
<tr>
<td><code>lcond_tdl</code></td>
<td>A TDL traversal where $\oplus \stackrel{\text{def}}{=} \leftarrow$</td>
</tr>
<tr>
<td><code>lseq_tdl</code></td>
<td>A TDL traversal where $\oplus \stackrel{\text{def}}{=} &lt;$</td>
</tr>
</tbody>
</table>

Remark

The language TL provides a library of “standard” higher-order traversals including those listed above. TL also provides higher-order traversal primitives enabling user-level specification of special-purpose higher-order traversals.
Consider the strategy $rcond_{tdl}\{\text{int}_1 \rightarrow \text{int}_2 \rightarrow \text{int}_1\}$

Result

$\text{SKIP} \leftrightarrow (\text{int}_2 \rightarrow 1) \leftrightarrow (\text{int}_2 \rightarrow 2)$
More Examples

- Input term: \( add(1, 2) \)
- Term sequence resulting from a top-down left-to-right traversal:
  \( \langle add(1, 2), 1, 2 \rangle \)
- Strategic Expression
  \[ rcond_{tdl}\{int_1 \rightarrow int_2 \rightarrow int_1\}[add(1, 2)] \]
- Evaluation Result
  \[ SKIP \mapsto (int_2 \rightarrow 1) \mapsto (int_2 \rightarrow 2) \]
• **Strategic Expression**
  
  \[ rcond_{tdl}\{int_1 \rightarrow transient(int_2 \rightarrow int_1)\}[add(1, 2)] \]

• **Evaluation Result**
  
  \[ SKIP \mapsto transient(int_2 \rightarrow 1) \mapsto transient(int_2 \rightarrow 2) \]

• **Strategic Expression**
  
  \[ rcond_{tdl}\{transient(int_1 \rightarrow int_2 \rightarrow int_1)\}[add(1, 2)] \]

• **Evaluation Result**
  
  \[ SKIP \mapsto (int_2 \rightarrow 1) \mapsto SKIP \]
Swap Revisited

- Consider the following:
  
  \[
  r1 : \quad int_1 \rightarrow transient(int_2 \rightarrow int_1)
  \]
  
  \[
  r2 : \quad transient(r1) \rightarrow transient(r1)
  \]

  \[
  t = add(1, 2)
  \]

- The evaluation of \( rcond_{tdl}\{r2\}[t] \) will yield:

  \[
  transient(int_2 \rightarrow 1) \rightarrow transient(int_2 \rightarrow 2)
  \]

- Thus the evaluation of \( TDL\{rcond_{tdl}\{r2\}[t]\}(t) \) will yield:

  \[
  add(2, 1)
  \]
Goals

- Develop a tool that encourages and facilitates exploration in transformation-based design and development

- Provide an agile transformation system wherein it is possible to add new language abstractions with relatively little effort
The Components of HATS

Parser Generator
Source Parser
TL Parser
Transformation Engine
PrettyPrinter
Tree Viewer
Trace Viewer
The Architecture of HATS from a Dataflow Perspective

Parsing Phase

- .bnf
- .spec
- .tgt
- .tlp
- .parsed
- UserDefinedFunctions.sml
- .sty

Feedback

- output.stat
- .transformed
- rules.xml
- trace.xml
- Various trace-related information

Text

Tree

Metrics

UserDefinedFunctions.sml
A Screenshot of the Editor
A Screenshot of the Tree Viewer
A Screenshot of the Trace Viewer
TL Supports:

- **Rewrite Rules:**
  - First-order labelled conditional rewrite rules
  - Higher-order labelled conditional rewrite rules

- **Iterators:**
  - **Traversals**
    - First-order user-definable - both generic and specific
    - Higher-order user-definable - both generic and specific
  - **FIX**
TL Supports:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;+</code></td>
<td>left-biased choice</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>right-biased choice</td>
</tr>
<tr>
<td><code>&lt;;</code></td>
<td>left-to-right sequential composition</td>
</tr>
<tr>
<td><code>&gt;;</code></td>
<td>right-to-left sequential composition</td>
</tr>
<tr>
<td><code>&lt;*</code></td>
<td>left-to-right conjunctive composition</td>
</tr>
<tr>
<td><code>&gt;*</code></td>
<td>right-to-left conjunctive composition</td>
</tr>
<tr>
<td>transient</td>
<td>a unary combinator</td>
</tr>
<tr>
<td>opaque</td>
<td>a unary combinator</td>
</tr>
<tr>
<td>raise</td>
<td>a unary combinator</td>
</tr>
<tr>
<td>hide</td>
<td>a unary combinator</td>
</tr>
<tr>
<td>lift</td>
<td>a unary combinator</td>
</tr>
</tbody>
</table>
TL Supports:

- **mapL**: Left-to-right one-layer traversal
- **mapR**: Right-to-left one-layer traversal
- **mapB**: Parallel one-layer traversal
- **fold**: Strategic composition
• The distributed data problem arises from a discord between the semantic association of terms within a specification and the structural association of terms resulting from the term language definition (e.g., the swap example).

• Higher-order strategies provide an abstraction in which instances of the distributed data problem can be understood.
• In order to realize the full potential of dynamic strategy creation within a higher-order strategic framework, **traversals** must be lifted to the higher-order.

• The **transient** combinator is useful in controlling the application of dynamically created rules and strategies.

• **TL** is a strategic programming language supporting these abstractions.
For Further Reading I

F. Baader and T. Nipkow.  
*Term Rewriting and All That.*  
Cambridge University Press.

V. L. Winter.  
Program Transformation: What, How, Why?  
*Encyclopedia of Computer Science and Engineering,*  
*Wiley and Sons.*

V. L. Winter and M. Subramaniam.  
Dynamic Strategies, Transient Strategies, and the Distributed Data Problem.  
V. Winter and J. Beranek.
Program Transformation Using HATS 1.84.

V. Winter.
Stack-based Strategic Control.
*In Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming, June 2007.*
The End