Verilog Synthesis

Summer School on Generative and Transformational Techniques in Software Engineering, 2005

Outline

The Signature of a Rewrite Rule

Definition

From the perspective of type, a labelled rewrite rule has the form:

$$id : pattern \longrightarrow s^n \ [\ if\ Boolean\ ]$$

- the type **id** is the set of identifiers
- the type **pattern** is the set of **parse expressions** over a given grammar
- the type s^n is the set of **strategies** of order n
- the Boolean condition is optional

The Signature of a Rewrite Rule

Definition

From the perspective of type, a labelled rewrite rule has the form:

$$id : pattern \longrightarrow s^n \ [\ if\ Boolean\ ]$$

- the type **id** is the set of identifiers
- the type **pattern** is the set of **parse expressions** over a given grammar
- the type s^n is the set of **strategies** of order n
- the Boolean condition is optional

Parse Expression : Pattern

Definition

A **parse expression** is a notation for describing parse trees (concrete syntax trees). Parse expressions are of type **pattern**.

- Let $G = (N, T, P, S)$ denote a context-free grammar.
  - A_{id} is a parse expression if $A \in N$. In the context of matching, the parse expression A_{id} is a variable quantified over the set $\{\alpha \mid A \underset{G}{\overset{*}{\Rightarrow}} \alpha \wedge \alpha \in T^*\}$
  - $A[\![\alpha']\!]$ is a parse expression if $A \underset{G}{\overset{+}{\Rightarrow}} \alpha$ The parse expression $A[\![\alpha']\!]$ is quantified over the set $\{\beta \mid A \underset{G}{\overset{+}{\Rightarrow}} \alpha \underset{G}{\overset{*}{\Rightarrow}} \beta \wedge \beta \in T^*\}$

Parse Expression : Pattern

Definition

A **parse expression** is a notation for describing parse trees (concrete syntax trees). Parse expressions are of type **pattern**.

- Let $G = (N, T, P, S)$ denote a context-free grammar.
  - A_{id} is a parse expression if $A \in N$. In the context of matching, the parse expression A_{id} is a variable quantified over the set $\{\alpha \mid A \underset{G}{\overset{*}{\Rightarrow}} \alpha \wedge \alpha \in T^*\}$
  - $A[\![\alpha']\!]$ is a parse expression if $A \underset{G}{\overset{+}{\Rightarrow}} \alpha$ The parse expression $A[\![\alpha']\!]$ is quantified over the set $\{\beta \mid A \underset{G}{\overset{+}{\Rightarrow}} \alpha \underset{G}{\overset{*}{\Rightarrow}} \beta \wedge \beta \in T^*\}$

Parse Expression Examples

stmtS	::=	stmt stmtS | ()
stmt	::=	blocking_assign ";" | par_block | ...
par_block	::=	"fork" stmtS "join"
blocking_assign	::=	lvalue "=" E
lvalue	::=	...
E	::=	id | ...
...		

$$stmtS_1$$
$$stmtS[\![stmt_1\ stmtS_1]\!]$$
$$stmtS[\![blocking\_assign_1;\ stmtS_1]\!]$$
$$stmtS[\![lvalue_1 = E_1;\ stmtS_1]\!]$$
$$stmtS[\![lvalue_1 = E_1;]\!]$$
$$stmtS[\![stmt_1\ stmt_2\ stmt_3\ stmt_4]\!]$$

Strategic Expression: Strategy

Definition

A **strategic expression** is an expression whose evaluation yields a **strategy**. A strategic expression of order n has type s^n.

- $s^0 =$ a pattern
- $s^{n+1} = lhs \rightarrow s^n$
- s^1 is a **first-order strategy**
- s^2 is a **second-order strategy**
- s^n where $n > 1$ is a **higher-order strategy**
- the result of applying a strategy of type s^{n+1} to a tree t is a strategy of type s^n

Strategic Expression: Strategy

Definition

A **strategic expression** is an expression whose evaluation yields a **strategy**. A strategic expression of order n has type s^n.

- $s^0 =$ a pattern
- $s^{n+1} = lhs \rightarrow s^n$
- s^1 is a **first-order strategy**
- s^2 is a **second-order strategy**
- s^n where $n > 1$ is a **higher-order strategy**
- the result of applying a strategy of type s^{n+1} to a tree t is a strategy of type s^n

An Example of a First-Order Strategy

wrap: stmt⟦ blocking_assign$_1$; ⟧ → stmt⟦ fork blocking_assign$_1$; join ⟧

stmtS	::=	stmt stmtS | ()
stmt	::=	blocking_assign ";" | par_block | ...
par_block	::=	"fork" stmtS "join"
blocking_assign	::=	lvalue "=" E
lvalue	::=	...
E	::=	id | ...
...		

An Example of a Second-Order Strategy

propagate: blocking_assign$[\![\mathrm{id}_1 = E_1]\!] \rightarrow E[\![\mathrm{id}_1]\!] \rightarrow E_1$

stmtS	::=	stmt stmtS | ()
stmt	::=	blocking_assign ";" | par_block | ...
par_block	::=	"fork" stmtS "join"
blocking_assign	::=	lvalue "=" E
lvalue	::=	...
E	::=	id | ...
...		

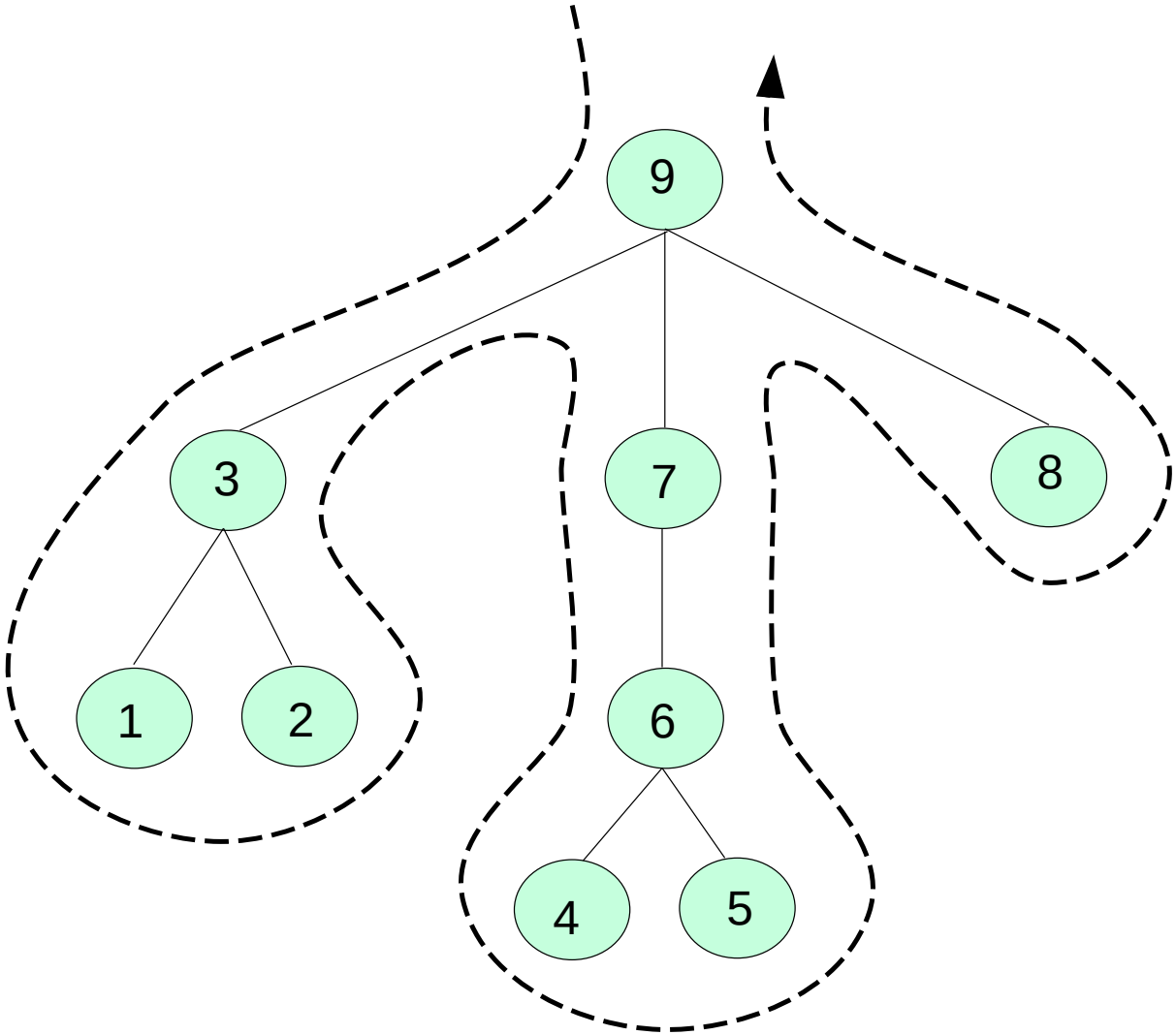
Constructing Strategies

Definition

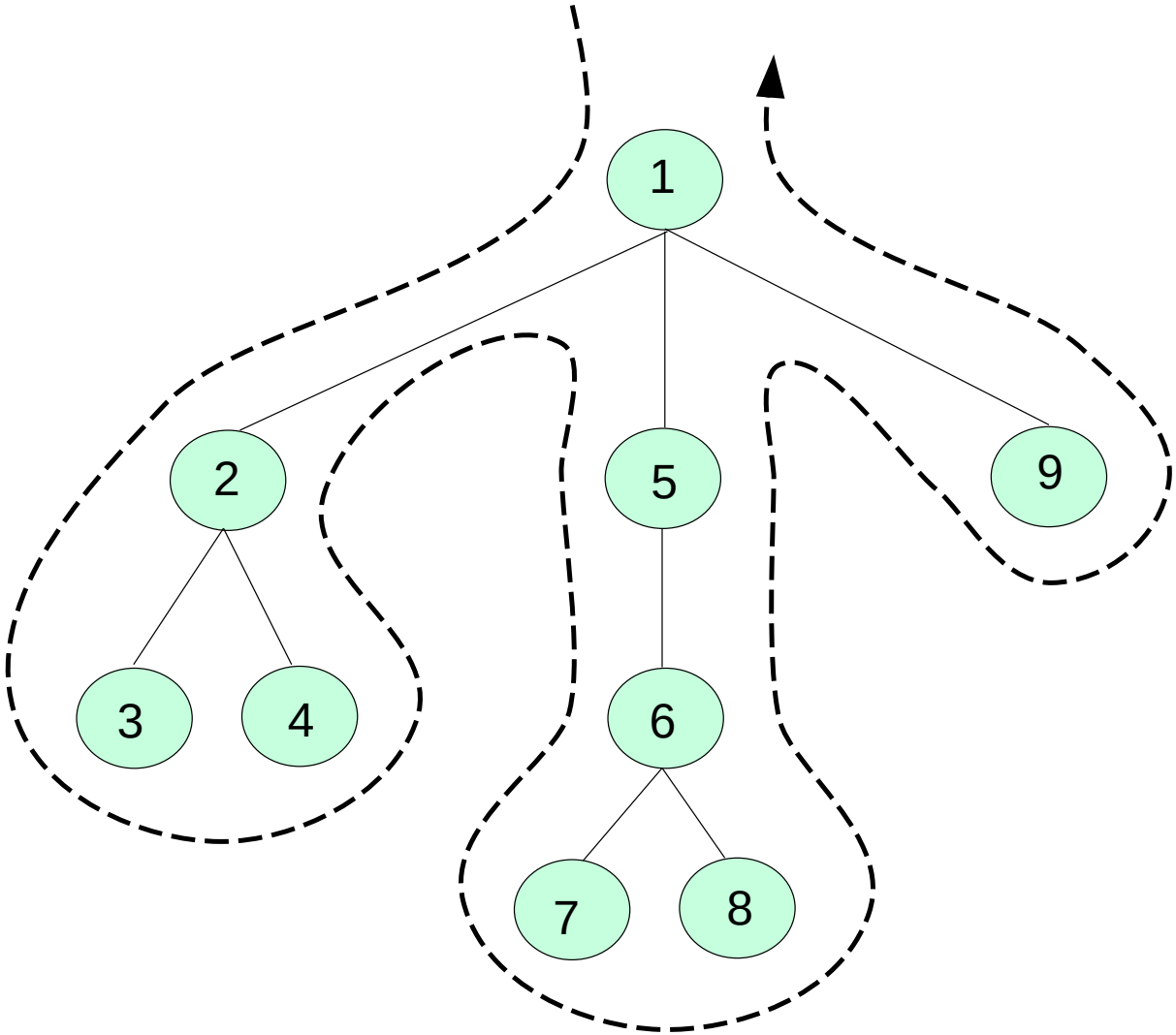
Combinators are operators that can be used to construct strategies

Symbol	Description	Example
$<+$	left-biased choice	$s_1 <+ s_2$
$+>$	right-biased choice	$s_1 +> s_2$
$<;$	left-to-right sequential composition	$s_1 <; s_2$
$;>$	right-to-left sequential composition	$s_1 ;> s_2$
transient	a unary combinator	$transient(s)$
hide	a unary combinator	$hide(s)$

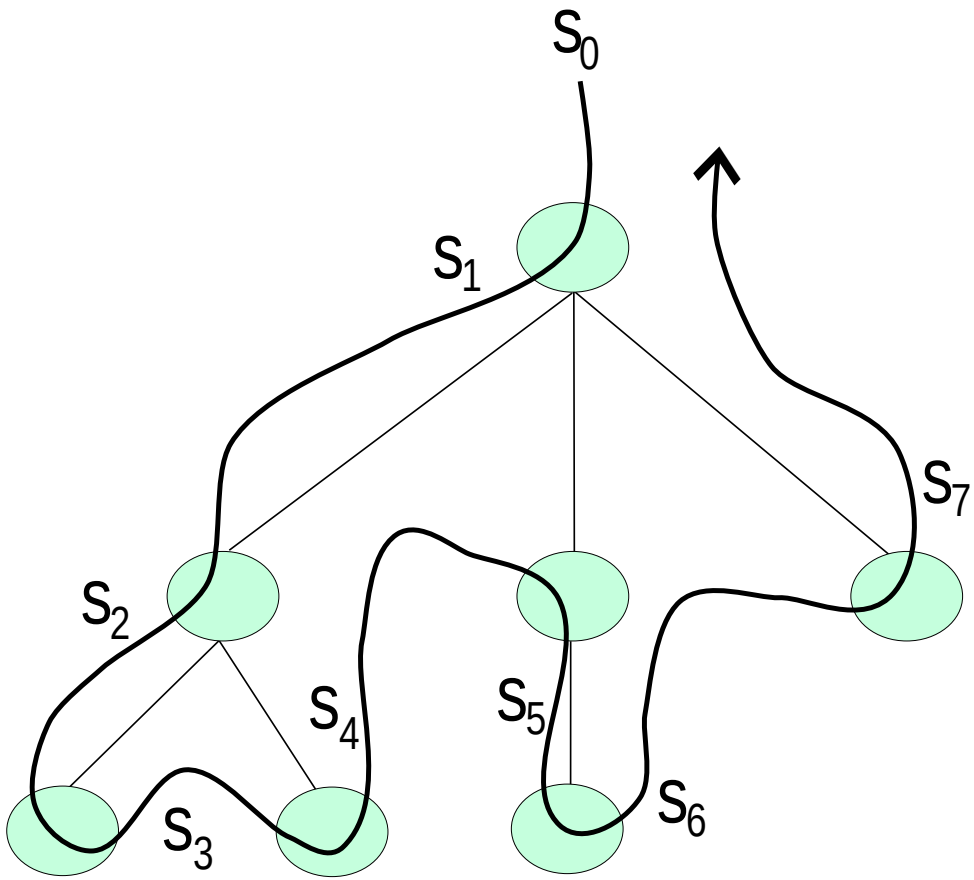
A Bottom-up Left-to-right(BUL) Generic Traversal



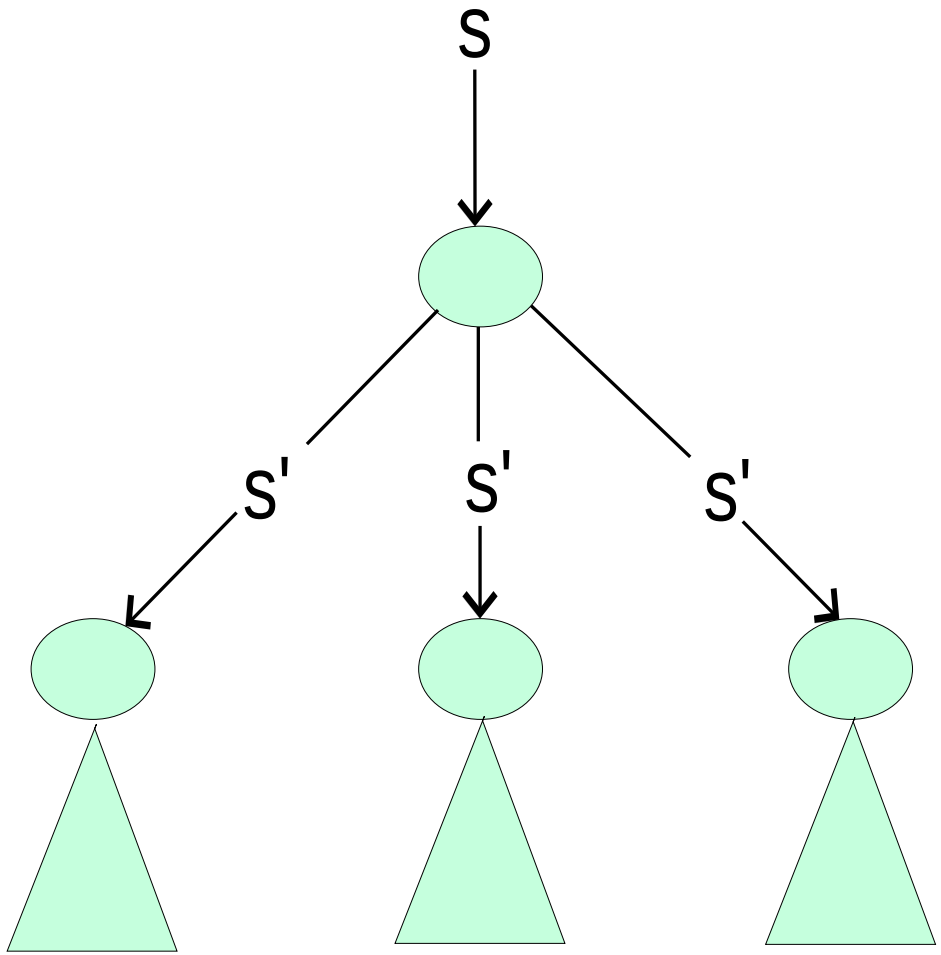
A Top-down Left-to-right(TDL) Generic Traversal



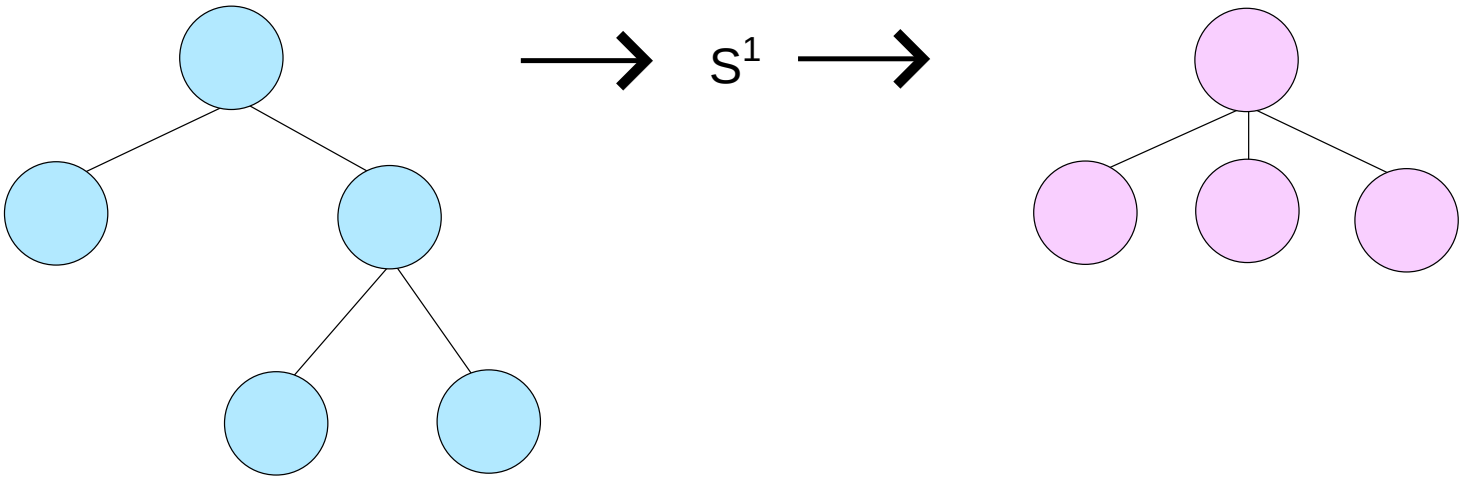
TDL Traversal from a Strategic Perspective



TD Traversal from a Strategic Perspective



First-Order Strategy Application



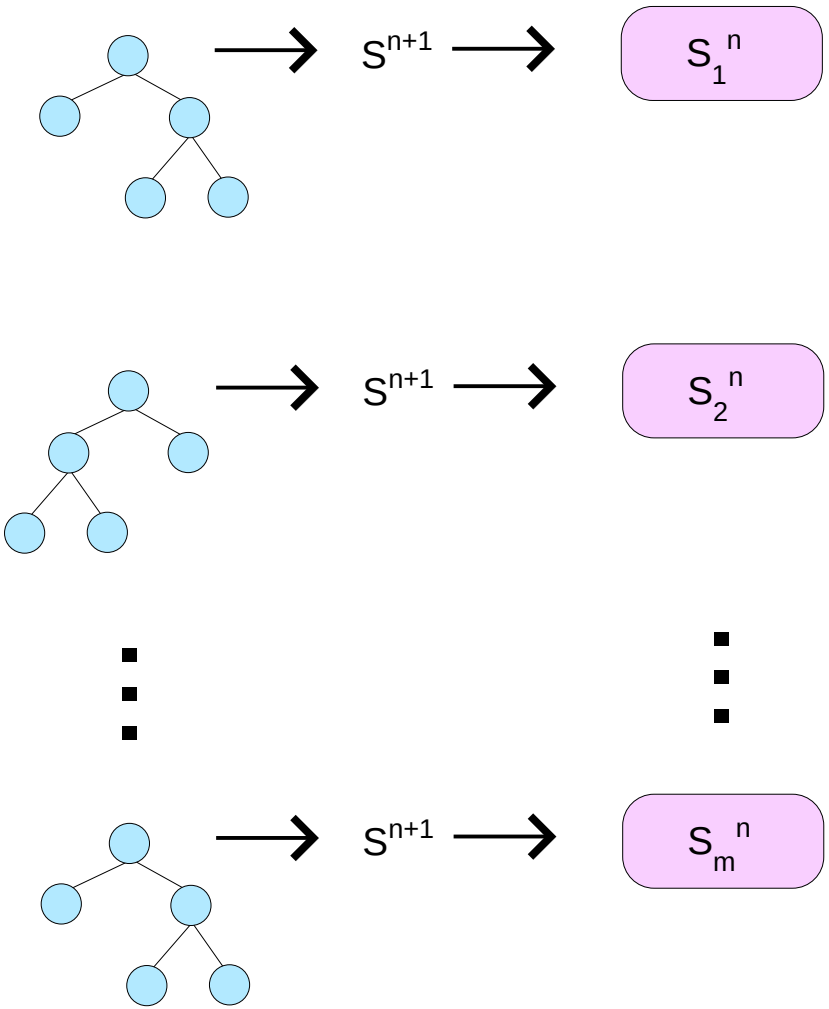
Definitions of Some First-Order Traversals

def BUL s = all_thread_left(BUL{s}) <; s

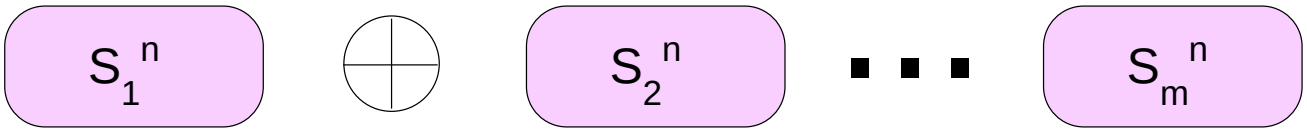
def TDL s = s <; all_thread_left(TDL{s})

def Special_TD s = (par_block$_1$ $\rightarrow$ TDL{s}(par_block$_1$))
<+
all_broadcast(Special_TD{s})

Higher-Order Strategy Application



Higher-Order Strategy Composition



A Taxonomy of Some Generic Higher-Order Traversals

Traversal	bottom-up	top-down	left-to-right	right-to-left	$\oplus$
rcond_tdl		√	√		$+\!\!>$
rcond_tdr		√		√	$+\!\!>$
lcond_tdl		√	√		$<\!\!+$
lcond_tdr		√		√	$<\!\!+$
rcond_bul	√		√		$+\!\!>$
rcond_bur	√			√	$+\!\!>$
lcond_bul	√		√		$<\!\!+$
lcond_bur	√			√	$<\!\!+$
lseq_tdl		√	√		$<;$
lseq_tdr		√		√	$<;$
lseq_bul	√		√		$<;$
lseq_bur	√			√	$<;$

Verilog Synthesis

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

An Overview of Verilog

- Verilog is a hardware hardware description language (HDL)
- Verilog has a C-like syntax
- Verilog has constructs to describe parallel computation and sequential computation
  - The items in a module execute in parallel (e.g., *continuous assignment* statements and *always* statements)
  - Blocks of the form ⟦*begin* ... *end*⟧ execute the statements in their bodies in sequential order
  - Blocks of the form ⟦*fork* ... *join*⟧ execute the statements in their bodies in parallel

Synthesis Goals

Goal

Develop a transformation-based synthesis system that removes sequential computation from Verilog programs

Goal

Construct a transformation whose manipulations are guided by correctness-preserving algebraic laws

Synthesis Goals

Goal

Develop a transformation-based synthesis system that removes sequential computation from Verilog programs

Goal

Construct a transformation whose manipulations are guided by correctness-preserving algebraic laws

Synthesis Example: Source

```
module example (out1, out2, in, cs );
input in;
output out1, out2;

always@(*) begin
    out1 = !cs;
    ns = out1;
    out2 = !out1 || c2;
    if (cs ==0) out2 = !out1;
    else ns = 0;

end

endmodule
```

Intermediate Form

```
module example(out1, out2, in, cs);
input in;
output out1, out2;
always@(*) begin
    fork out1 = !cs; ns = ns; out2 = out2; join
    fork ns = out1; out1 = out1; out2 = out2; join
    fork out2 = !out1 || c2; out1 = out1; ns = ns; join
    if ( cs == 0 ) fork out2 = !out1; out1 = out1; ns = ns; join
    else fork ns = 0; out1 = out1; out2 = out2; join

end
endmodule
```

Target

```
module example(out1, out2, in, cs);
input in;
output out1 , out2;
always@(*) begin
    fork
        ns = (cs == 0) ? !cs : 0;
        out1 = (cs == 0) ? !cs : !cs;
        out2 = (cs == 0) ? !!cs : !!cs || c2;
    join

end
endmodule
```

Law

Parallel assignment completion.

$$(x, y, \ldots := e, f, \ldots) = (x, y, \ldots, z := e, f, \ldots, z)$$

Law

Parallel assignment reordering.

$$(x, \ldots, y, z, \ldots := e, \ldots, f, g, \ldots) = (x, \ldots, z, y, \ldots := e, \ldots, g, f, \ldots)$$

Law

Parallel assignment constant propagation.
$(\overrightarrow{v} := g; \overrightarrow{v} := h(\overrightarrow{v})) = (\overrightarrow{v} := h(g))$ *where* $\overrightarrow{v}$ *is an assignment state.*

Law

Conditional constructor elimination.
$((\vec{v} := g) \lhd c \rhd (\vec{v} := h)) = (\vec{v} := (g \lhd c \rhd h))$

A Verilog Grammar Fragment

modulee	::=	module module_id “;” m_item_0orMore endmodule
m_item_0orMore	::=	module_item m_item_0orMore | ()
module_item	::=	continuous_assign | always_stmt | ...
continuous_assign	::=	“assign” lvalue “=” E “;”
always_stmt	::=	“always” stmt
stmtS	::=	stmt stmtS | ()
stmt	::=	blocking_assign “;” | seq_block | par_block | ...
seq_block	::=	“begin” stmtS “end”
par_block	::=	“fork” stmtS “join”
blocking_assign	::=	lvalue “=” E
...		

Transformations yielding Assignment Normal Form

synthesize: BUL{ wrap <; Law1 } <; BUL{ Law3 <; Law4 }

wrap: stmt⟦ blocking_assign$_1$; ⟧
$\longrightarrow$
stmt⟦ fork blocking_assign$_1$; join ⟧

Law1: modulee_0
$\rightarrow$
$\text{Special\_TD}\{\ \text{lseq\_bul}\{\ \text{make\_total}\ \}[\text{modulee}_0]\ \}(\text{modulee}_0)$

make_total: $\text{blocking\_assign}[\![\text{id}_1 = \text{E}_1]\!] \rightarrow \text{transient}(\text{check}[\text{id}_1] <\!+ \text{add}[\text{id}_1])$

check: $\text{id}_1 \rightarrow \text{stmtS}[\![\ \text{id}_1 = \text{E}_2\ ;\ \text{stmtS}_3\ ]\!] \rightarrow \text{stmtS}[\![\ \text{id}_1 = \text{E}_2\ ;\ \text{stmtS}_3\ ]\!]$

add: $\text{id}_1 \rightarrow \text{stmtS}[\![\ ]\!] \rightarrow \text{stmtS}[\![\ \text{id}_1 = \text{id}_1;\ ]\!]$

Law3: $\text{stmtS}[\![\ \text{par\_block}_1\ \text{par\_block}_2\ ]\!]$
$\longrightarrow$
$\text{BUL}\{\text{lseq\_tdl}\{\text{propagate}\}[\text{par\_block}_1]\ \}(\text{stmtS}[\![\ \text{par\_block}_2\ ]\!]\ )$

propagate: $\text{blocking\_assign}[\![\ \text{id}_1 = \text{E}_1\ ]\!] \longrightarrow \text{E}[\![\ \text{id}_1\ ]\!] \longrightarrow \text{E}_1$

Law4: $stmt[\![\ if\ (\ E_1\ )\ stmt_1\ else\ stmt_2\ ]\!]$
$\longrightarrow$
$BUL\{\ lseq\_bul\{\ convert[E_1]\ \}[stmt_1]\ \}(stmt_2)$

convert E_0
$blocking\_assign[\![\ id_1 = E_1\ ]\!]$
:
$blocking\_assign[\![\ id_1 = E_2\ ]\!]$
$\longrightarrow$
$blocking\_assign[\![\ id_1 = (\ E_0\ )\ ?\ E_1\ :\ E_2\ ]\!]$

Result Summary

```
out1 = !cs;
ns = out1;
out2 = !out1 || c2;
if (cs ==0) out2 = !out1;
else ns = 0;
```

$\Rightarrow$

```
fork
    ns = (cs == 0) ? !cs : 0;
    out1 = (cs == 0) ? !cs : !cs;
    out2 = (cs == 0) ? !!cs : !!cs || c2;
join
```

For Further Reading I

J. Kyoda and H. Jifeng
Towards an Algebraic Synthesis of Verilog
Technical Report, UNU/IIST Report No. 218, 2001.

V. L. Winter and M. Subramaniam
Dynamic Strategies, Transient Strategies, and the Distributed Data Problem.
Science of Computer Programming (Special Issue on Program Transformation), 52:165–212, Elsevier, 2004.

V. L. Winter
Strategy Construction in the Higher-Order Framework of TL.
Electronic Notes in Theoretical Computer Science (ENTCS), 124(1), 2004