Model Checking Higher-Order Computation: II

Luke Ong

Computing Laboratory, University of Oxford

Marktoberdorf Summer School, 4-15 August 2009
Outline of Part 2

1. A Typing System Characterising MSO Theories of Recursion Schemes
   - Preliminaries: Mu-Calculus, APT and Parity Games
   - An Intersection Type System
   - Two Relatively Cheap Fragments

2. Application: A New Approach to Verifying Functional Programs
   - Verification by Reduction to Model Checking Recursion Schemes
   - Resource Usage Problem: A Case Study
   - Experimentation: Preliminary Results and Demo:

Lecture slides and references will be viewable on my homepage
users.comlab.ox.ac.uk/luke.ong
Outline of Part 2

1 A Typing System Characterising MSO Theories of Recursion Schemes
   - Preliminaries: Mu-Calculus, APT and Parity Games
   - An Intersection Type System
   - Two Relatively Cheap Fragments

2 Application: A New Approach to Verifying Functional Programs
   - Verification by Reduction to Model Checking Recursion Schemes
   - Resource Usage Problem: A Case Study
   - Experimentation: Preliminary Results and Demo:

Lecture slides and references will be viewable on my homepage
users.comlab.ox.ac.uk/luke.ong
1. A Typing System Characterising MSO Theories of Recursion Schemes
   - Preliminaries: Mu-Calculus, APT and Parity Games
   - An Intersection Type System
   - Two Relatively Cheap Fragments

2. Application: A New Approach to Verifying Functional Programs
   - Verification by Reduction to Model Checking Recursion Schemes
   - Resource Usage Problem: A Case Study
   - Experimentation: Preliminary Results and Demo:
Alternating parity tree automaton (APT) ≡ modal mu-calculi

Theorem (Equivalence, Emerson+Jutla 91)

Let $A, \varphi, T$ range over APT, modal mu-formulas, ranked trees resp.

1. For each $A$, there exists $\varphi$ such that $A$ accepts $T$ iff $T$ satisfies $\varphi$.
2. For each $\varphi$, there exists $A$ such that $A$ accepts $T$ iff $T$ satisfies $\varphi$.

Positive Boolean formulas over $X$: $B^+(X) \ni \theta ::= t \mid f \mid x \mid \theta \land \theta \mid \theta \lor \theta$

$Y \subseteq X$ satisfies $\theta$ just if assigning true to elements in $Y$ and false to others makes $\theta$ true.

An APT over $\Sigma$-labelled trees is a tuple $A = (\Sigma, Q, \delta, q_I, \Omega)$ where

- $\Sigma$ is a ranked alphabet; $m$ is the largest arity of terminals
- $q_I \in Q$ is the initial state
- $\delta : Q \times \Sigma \longrightarrow B^+(\{1, \ldots, m\} \times Q)$ is the transition function
- $\Omega : Q \longrightarrow \{0, \cdots, M - 1\}$ is the priority function.
Alternating parity tree automaton (APT) ≡ modal mu-calculus

Theorem (Equivalence, Emerson+Jutla 91)

Let $A, \varphi, T$ range over APT, modal mu-formulas, ranked trees resp.

1. For each $A$, there exists $\varphi$ such that $A$ accepts $T$ iff $T$ satisfies $\varphi$.
2. For each $\varphi$, there exists $A$ such that $A$ accepts $T$ iff $T$ satisfies $\varphi$.

Positive Boolean formulas over $X$: $B^+(X) \ni \theta ::= t \mid f \mid x \mid \theta \land \theta \mid \theta \lor \theta$

$Y \subseteq X$ satisfies $\theta$ just if assigning true to elements in $Y$ and false to others makes $\theta$ true.

An APT over $\Sigma$-labelled trees is a tuple $A = (\Sigma, Q, \delta, q_I, \Omega)$ where

- $\Sigma$ is a ranked alphabet; $m$ is the largest arity of terminals
- $q_I \in Q$ is the initial state
- $\delta : Q \times \Sigma \rightarrow B^+(\{1, \ldots, m\} \times Q)$ is the transition function
- $\Omega : Q \rightarrow \{0, \cdots, M - 1\}$ is the priority function.
A run-tree of an APT is just a set of maximal state-annotated paths in the tree that respect the transition relation.

A tree is accepted by an APT just if there is a run-tree such that every infinite path \( \pi \) in it satisfies the parity condition.

Let \( \pi = \pi_1 \pi_2 \cdots \) be an infinite path in \( r \); for each \( i \geq 0 \), let the state label of the node \( \pi_1 \cdots \pi_i \) be \( q_{n_i} \) where \( q_{n_0} \), the state label of \( \epsilon \), is \( q_I \). We say that

\( \pi \) satisfies the parity condition

The largest priority that occurs infinitely often in \( \Omega(q_{n_0}) \Omega(q_{n_1}) \Omega(q_{n_2}) \cdots \) is even.
Example

Let $\Sigma = \{ a : 2, b : 1, c : 0 \}$.

Let $A$ be the APT $(\Sigma, \{ q_0, q_1 \}, \delta, q_0, \Omega)$, where (let $q \in \{ q_0, q_1 \}$)

$$
\delta : \begin{cases}
(q, a) &\mapsto (1, q_1) \land (2, q) \\
(q, b) &\mapsto (1, q) \\
(q, c) &\mapsto \text{true}
\end{cases}
$$

$$
\Omega : \begin{cases}
q_0 &\mapsto 2 \\
q_1 &\mapsto 1
\end{cases}
$$

$A$ accepts a $\Sigma$-tree $t$ just if for every path of $t$, if the path ever takes the left branch of a node labeled by $a$, then the path contains $c$.

For a tree rejected by $A$, consider the full binary tree with nodes labelled by $a$. 
A parity game is a tuple \((V_R, V_V, v_0, E, \Omega)\) such that

- \(E \subseteq V \times V\) is the edge relation of a directed graph whose node-set \(V := V_R + V_V\); \(v_0 \in V\) is the start node
- \(\Omega : V \rightarrow \{0, \ldots, M - 1\}\) assigns a priority to each node.

Playing a parity game

A play consists in the players, \(R\) (Refuter) and \(V\) (Verifier), taking turns to move a token along the edges of the graph. At a given stage of the play, suppose the token is on an \(R\)-node \(v\) (respectively \(V\)-node), then \(R\) (respectively \(V\)) chooses an edge \((v, v')\) and moves the token onto \(v'\). At the start of a play, the token is placed on \(v_0\).

Thus a play is a finite or infinite path \(\pi = v_0 v_{n_1} v_{n_2} \cdots\) in the graph that starts from \(v_0\).
A **parity game** is a tuple \((V_R, V_V, v_0, E, \Omega)\) such that

- \(E \subseteq V \times V\) is the edge relation of a directed graph whose node-set \(V := V_R + V_V\); \(v_0 \in V\) is the start node
- \(\Omega : V \rightarrow \{0, \cdots, M - 1\}\) assigns a priority to each node.

**Playing a parity game**

A **play** consists in the players, \(R\) (**Refuter**) and \(V\) (**Verifier**), taking turns to move a token along the edges of the graph. At a given stage of the play, suppose the token is on an \(R\)-node \(v\) (respectively \(V\)-node), then \(R\) (respectively \(V\)) chooses an edge \((v, v')\) and moves the token onto \(v'\). At the start of a play, the token is placed on \(v_0\).

Thus a **play** is a finite or infinite path \(\pi = v_0 \ v_{n_1} \ v_{n_2} \cdots\) in the graph that starts from \(v_0\).
Winning condition and strategy

Who wins a maximal play?

Let $\pi$ be a maximal play.

- If $\pi$ is finite, and ends in a $V$-node (respectively $R$-node), then $R$ (respectively $V$) wins.
- If $\pi$ is infinite, $V$ wins iff $\pi$ satisfies the parity condition.

Definitions

A $V$-strategy $\mathcal{W}$ is a map from plays ending in a $V$-node to a node extending the play.

$\mathcal{W}$ is winning if $V$ wins every (maximal) play $\pi$ that conforms with the strategy.
An Example: Who has a winning strategy?

\[
\begin{array}{c}
\text{R, 3} \\
\downarrow \\
\text{V, 3} \\
\downarrow \\
\text{V, 2} \\
\downarrow \\
\text{V, 2} \\
\downarrow \\
\text{R, 0} \\
\downarrow \\
\text{R, 1} \\
\end{array}
\]

\[
\begin{array}{c}
\text{V, 3} \\
\downarrow \\
\text{V, 1} \\
\downarrow \\
\text{R, 2} \\
\downarrow \\
\text{R, 3} \\
\downarrow \\
\text{R, 1} \\
\end{array}
\]

\[
\begin{array}{c}
\text{R, 0} \\
\downarrow \\
\text{V, 0} \\
\end{array}
\]

\[
\begin{array}{c}
\text{V, 0} \\
\end{array}
\]

\[R\] always chooses the right child - winning strategy.
An Example: Who has a winning strategy?

\[ R, 3 \]
\[ V, 3 \]
\[ V, 2 \]
\[ V, 2 \]
\[ R, 0 \]
\[ R, 1 \]
\[ V, 3 \]
\[ V, 1 \]
\[ R, 2 \]
\[ R, 3 \]
\[ R, 1 \]
\[ R, 0 \]
\[ V, 0 \]

*R* always chooses the right child - winning strategy.
A Game Reading of the Fundamental Semantic Theorem

**Theorem (Emerson+Street 89)**

Given a labelled transition system $\mathcal{L}$, a start state $s_0$, and a modal $\mu$-formula $\varphi$, there is a parity game $G(\mathcal{L}, s_0, \varphi)$ such that $\mathcal{L}, s_0 \models \varphi$ iff Verifier has a winning strategy for $G(\mathcal{L}, s_0, \varphi)$.

**Example** $G(\mathcal{L}, 0, \mu Y.\nu Z.[a](\langle b \rangle t \lor Y) \land Z)$ where

$$
\mathcal{L} = \begin{array}{c}
0 \\
1 \\
2
\end{array} \xrightarrow{a} \begin{array}{c}
\circ \\
\circ \\
\circ
\end{array} \xrightarrow{b}$$

(For the game graph, see next slide.)
Game graph of $G(T, \mu Y.\nu Z.[a]((\langle b\rangle t \lor Y) \land Z))$

(0, $\mu Y.\nu Z.[a]((\langle b\rangle t \lor Y) \land Z))$

↓

(0, Y)

↓

(0, $\nu Z.[a]((\langle b\rangle t \lor Y) \land Z))$

↓

(0, Z)

↓

(0, $[a]((\langle b\rangle t \lor Y) \land Z))$ R

↓

(1, $((\langle b\rangle t \lor Y) \land Z))$ R

V (1, $\langle b\rangle t \lor Y$)

↓

V (1, $\langle b\rangle t$)

↓

(2, t)

V (1, $\langle b\rangle t$)

↓

(1, Y)

↓

(1, $[a]((\langle b\rangle t \lor Y) \land Z))$ R

↓

(1, Z)

↓

(1, $((\langle b\rangle t \lor Y) \land Z)$)

↓

(1, $((\langle b\rangle t \lor Y) \land Z)$)

↓

V (1, $\langle b\rangle t \lor Y$)

↓

V (0, $\langle b\rangle t$)

↓

V (0, $\langle b\rangle t$)
Theorem (**Characterisation.** Kobayashi + O. LiCS 2009)

Given a property $\varphi$ (APT / mu-calculus) there is a typing system $\mathcal{K}_\varphi$ such that for every recursion scheme $G$, the tree $\llbracket G \rrbracket$ satisfies $\varphi$ iff $G$ is $\mathcal{K}_\varphi$-typable.

Theorem (**Parameterised Complexity.** Kobayashi + O. LiCS 2009)

There is a type-inference algorithm polytime in size of recursion scheme, assuming the other parameters are fixed.

The runtime is

$$O(p^{1+\lfloor m/2 \rfloor} \exp_n((a | Q| M)^{1+\epsilon}))$$

where $p$ is the number of rewrite rules of the scheme, $a$ is largest arity of the types, $M$ the number of priorities and $|Q|$ the number of states.
**Intersection types**: Long history. First used to construct filter models for untyped $\lambda$-calculus (Dezani, Barendregt, et al. early 80s).

Fix an APT $A = (\Sigma, Q, \delta, q_I, \Omega)$.

**Idea**: Refine intersection types with APT states $q$ and priorities $m_i$ of APT.

$$\begin{align*}
Types \quad \theta & ::= \quad q \mid \tau \to \theta \\
\tau & ::= \quad \land \{ (\theta_1, m_1), \ldots, (\theta_k, m_k) \}
\end{align*}$$

**Intuition**: A tree function described by $(q_1, m_1) \land (q_2, m_2) \to q$.

The largest priority in this path (including the root and $q_1$) is $m_1$.

The largest priority in this path (including the root and $q_2$) is $m_2$. 
Intersection types: Long history. First used to construct filter models for untyped \( \lambda \)-calculus (Dezani, Barendregt, et al. early 80s).

Fix an APT \( \mathcal{A} = (\Sigma, Q, \delta, q_I, \Omega) \).

**Idea:** Refine intersection types with APT states \( q \) and priorities \( m_i \) of APT.

\[
\begin{align*}
\text{Types} & \quad \theta \ ::= \quad q \mid \tau \rightarrow \theta \\
\tau & \ ::= \quad \bigwedge \{ (\theta_1, m_1), \cdots, (\theta_k, m_k) \}
\end{align*}
\]

**Intuition.** A tree function described by \( (q_1, m_1) \land (q_2, m_2) \rightarrow q \).
A type-checking approach

Typing judgement

\[ \Gamma \vdash t : \theta \]

where the environment \( \Gamma \) is a finite set of bindings \( x : (\theta, m)^b \) with \( b \in \{ t, f \} \).

- \( x : (\theta, m)^t \in \Gamma \) means \( x \) can be used only before visiting a state with priority larger than \( m \).
- \( x : (\theta, m)^f \in \Gamma \) means it is additionally required that \( x \) can be used after visiting a state with priority \( m \).

E.g. Suppose \( \Omega(q) = 0 \). Then \( \{ x : (q, 1)^t \} \vdash x : q \) is valid.
Type-checking infinite trees with parity condition

**Typing rules** are simple: only four rules - one per term-constructor.

**Definition of typability.** We say that $G$ is **typable** just if Verifier has a winning strategy in a **parity game** determined by the APT $(Q, \delta, q_I, \Omega)$.

**Intuition of the parity game:** A way to construct an infinite tree of type derivations, suitable for parity condition reasoning.
Underlying graph is bipartite; two kinds of vertices “$F : (\theta, m)$” and “$\Gamma$”.
Verifier tries to prove that scheme is typable; Refuter tries to disprove it.

**Start vertex:** $S : (q_I, \Omega(q_I))$.

**Verifier:** Given $F : (\theta, m)$, choose $\Gamma$ such that $\Gamma \vdash \text{rhs}(F) : \theta$ is valid.

**Refuter:** Given $\Gamma$, choose $F : (\theta, m) \in \Gamma$ (and ask Verifier to prove why $F$ has type $\theta$).

**Proof** “Standard” methods (e.g. type soundness via type preservation) apply, except reasoning about priorities, which is novel and of independent interest.
Typing rules are simple: only four rules - one per term-constructor.

**Definition of typability.** We say that $G$ is **typable** just if Verifier has a winning strategy in a **parity game** determined by the APT ($Q, \delta, q_I, \Omega$).

**Intuition of the parity game:** A way to construct an infinite tree of type derivations, suitable for parity condition reasoning. Underlying graph is bipartite; two kinds of vertices “$F : (\theta, m)$” and “$\Gamma$”. Verifier tries to prove that scheme is typable; Refuter tries to disprove it.

**Start vertex:** $S : (q_I, \Omega(q_I))$.

**Verifier:** Given $F : (\theta, m)$, choose $\Gamma$ such that $\Gamma \vdash \text{rhs}(F) : \theta$ is valid.

**Refuter:** Given $\Gamma$, choose $F : (\theta, m) \in \Gamma$ (and ask Verifier to prove why $F$ has type $\theta$).

**Proof** “Standard” methods (e.g. type soundness via type preservation) apply, except reasoning about priorities, which is novel and of independent interest.
Typing rules are simple: only four rules - one per term-constructor.

Definition of typability. We say that $G$ is typable just if Verifier has a winning strategy in a parity game determined by the APT $(Q, \delta, q_I, \Omega)$.

Intuition of the parity game: A way to construct an infinite tree of type derivations, suitable for parity condition reasoning. Underlying graph is bipartite; two kinds of vertices “$F : (\theta, m)$” and “$\Gamma$”. Verifier tries to prove that scheme is typable; Refuter tries to disprove it.

Start vertex: $S : (q_I, \Omega(q_I))$.

Verifier: Given $F : (\theta, m)$, choose $\Gamma$ such that $\Gamma \vdash rhs(F) : \theta$ is valid.

Refuter: Given $\Gamma$, choose $F : (\theta, m) \in \Gamma$ (and ask Verifier to prove why $F$ has type $\theta$).

Proof “Standard” methods (e.g. type soundness via type preservation) apply, except reasoning about priorities, which is novel and of independent interest.
\[(\theta, m)^b \uparrow \Omega(\theta) = (\theta, m)^t\]

\[x : (\theta, m)^b \vdash x : \theta\]  
\[\text{(T-Var)}\]

\[\{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a)\]

\[\emptyset \vdash a : \land_{j=1}^{k_1}(q_{1j}, m_{1j}) \rightarrow \cdots \rightarrow \land_{j=1}^{k_n}(q_{nj}, m_{nj}) \rightarrow q\]

\[\text{where } m_{ij} = \max(\Omega(q_{ij}), \Omega(q))\]  
\[\text{(T-Const)}\]

\[\Gamma_0 \vdash t_0 : (\theta_1, m_1) \land \cdots \land (\theta_k, m_k) \rightarrow \theta\]

\[\Gamma_i \uparrow m_i \vdash t_1 : \theta_i \text{ for each } i \in \{1, \ldots, k\}\]

\[\Gamma_0 \cup \Gamma_1 \cup \cdots \cup \Gamma_k \vdash t_0 \ t_1 : \theta\]  
\[\text{(T-App)}\]

\[\Gamma, x : \land_{i \in I}(\theta_i, m_i)^f \vdash t : \theta\quad I \subseteq J\]

\[\Gamma \vdash \lambda x. t : \land_{i \in J}(\theta_i, m_i) \rightarrow \theta\]  
\[\text{(T-Abs)}\]
Safety Fragment of Mu-Calculus / Trivial APT

Trivial APT are APT with a single priority of 0. [Aehlig, LMCS 2007]
Trivial acceptance condition: A tree is accepted just if there is a run-tree (i.e. state-annotation of nodes respecting the transition relation).
Equi-expressive with the “safety fragment” of mu-calculus:

\[ \phi, \psi ::= P_f | Z | \phi \lor \psi | \phi \land \psi | \langle i \rangle \phi | \nu Z . \phi. \]

But surprisingly

Theorem (Kobayashi + O., ICALP 2009)
The Trivial APT Acceptance Problem for order-\(n\) recursion schemes is still \(n\)-EXPTIME complete.

[\(n\)-EXPTIME hardness by reduction from word acceptance problem of order-\(n\) alternating PDA which is \(n\)-EXPTIME complete [Engelfriet 91].]
Trivial APT are APT with a single priority of 0. [Aehlig, LMCS 2007]

**Trivial acceptance condition:** A tree is accepted just if there is a run-tree (i.e. state-annotation of nodes respecting the transition relation).

Equi-expressive with the “safety fragment” of mu-calculus:

\[
\varphi, \psi ::= P_f \mid Z \mid \varphi \lor \psi \mid \varphi \land \psi \mid \langle i \rangle \varphi \mid \nu Z . \varphi.
\]

But surprisingly

**Theorem (Kobayashi + O., ICALP 2009)**

The Trivial APT Acceptance Problem for order-\(n\) recursion schemes is still \(n\)-EXPTIME complete.

\([n\text{-EXPTIME} \text{ hardness by reduction from word acceptance problem of order-}\(n\) alternating PDA which is } n\text{-EXPTIME complete} \text{ [Engelfriet 91].}\]
Trivial APT are APT with a single priority of 0. [Aehlig, LMCS 2007]

Trivial acceptance condition: A tree is accepted just if there is a run-tree (i.e. state-annotation of nodes respecting the transition relation).

Equi-expressive with the “safety fragment” of mu-calculus:

\[ \varphi, \psi ::= P_f \mid Z \mid \varphi \lor \psi \mid \varphi \land \psi \mid \langle i \rangle \varphi \mid \nu Z . \varphi. \]

But surprisingly

**Theorem (Kobayashi + O., ICALP 2009)**

*The Trivial APT Acceptance Problem for order-n recursion schemes is still \( n \)-EXPTIME complete.*

\( [n\text{-EXPTIME} \text{ hardness by reduction from word acceptance problem of order-} n \text{ alternating PDA which is } n\text{-EXPTIME} \text{ complete [Engelfriet 91].}] \)
Disjunctive APT are APT whose transition function maps each state-symbol pair to a purely disjunctive positive boolean formula. Disjunctive APT capture path / linear-time properties; equi-expressive with “disjunctive fragment” of mu-calculus:

\[ \varphi, \psi ::= P_f \land \varphi \mid Z \mid \varphi \lor \psi \mid \langle i \rangle \varphi \mid \nu Z. \varphi \mid \mu Z. \varphi \]

**Theorem (Kobayashi + O., ICALP 2009)**

The Disjunctive APT Acceptance Problem for order-\( n \) recursion schemes is \((n - 1)\)-EXPTIME complete.

\((n - 1)\)-EXPTIME decidable: For order-1 APT-types \( \land S_1 \rightarrow \cdots \rightarrow \land S_k \rightarrow q \), we may assume at most one \( S_i \)'s is nonempty (and is singleton). Hence only \( k \times |Q|^2 \times m \) many such types (N.B. exponential for general APT).

\((n - 1)\)-EXPTIME hardness: by reduction from emptiness problem of order-\( n \) deterministic PDA [Engelfriet 91].
Disjunctive APT are APT whose transition function maps each state-symbol pair to a purely disjunctive positive boolean formula. Disjunctive APT capture path / linear-time properties; equi-expressive with “disjunctive fragment” of mu-calculus:

\[ \varphi, \psi ::= P_f \land \varphi | Z | \varphi \lor \psi | \langle i \rangle \varphi | \nu Z.\varphi | \mu Z.\varphi \]

**Theorem (Kobayashi + O., ICALP 2009)**

The Disjunctive APT Acceptance Problem for order-\(n\) recursion schemes is \((n - 1)\)-EXPTIME complete.

\((n - 1)\)-EXPTIME decidable: For order-1 APT-types \(\land S_1 \to \cdots \to \land S_k \to q\), we may assume at most one \(S_i\)’s is nonempty (and is singleton). Hence only \(k \times |Q|^2 \times m\) many such types (N.B. exponential for general APT).

\((n - 1)\)-EXPTIME hardness: by reduction from emptiness problem of order-\(n\) deterministic PDA [Engelfriet 91].
Disjunctive APT are APT whose transition function maps each state-symbol pair to a purely disjunctive positive boolean formula. Disjunctive APT capture path / linear-time properties; equi-expressive with “disjunctive fragment” of mu-calculus:

$$\varphi, \psi ::= P_f \land \varphi \mid Z \mid \varphi \lor \psi \mid \langle i \rangle \varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

Theorem (Kobayashi + O., ICALP 2009)

The Disjunctive APT Acceptance Problem for order-n recursion schemes is \((n - 1)\)-EXPTIME complete.

\((n - 1)\)-EXPTIME decidable: For order-1 APT-types \(\bigwedge S_1 \rightarrow \cdots \rightarrow \bigwedge S_k \rightarrow q\), we may assume at most one \(S_i\)’s is nonempty (and is singleton). Hence only \(k \times |Q|^2 \times m\) many such types (N.B. exponential for general APT).

\((n - 1)\)-EXPTIME hardness: by reduction from emptiness problem of order-\(n\) deterministic PDA [Engelfriet 91].
Corollary

The following problems are \((n - 1)\)-EXPTIME complete: assume \(G\) is an order-\(n\) recursion scheme

1. **Reachability:** “Does \([G]\) have a node labelled by a given symbol?”
2. **LTL Model-Checking:** “Does every path in \([G]\) satisfy a given \(\varphi\)?”
3. **Resource Usage Problem**
Outline

1. **A Typing System Characterising MSO Theories of Recursion Schemes**
   - Preliminaries: Mu-Calculus, APT and Parity Games
   - An Intersection Type System
   - Two Relatively Cheap Fragments

2. **Application: A New Approach to Verifying Functional Programs**
   - Verification by Reduction to Model Checking Recursion Schemes
   - Resource Usage Problem: A Case Study
   - Experimentation: Preliminary Results and Demo:
Verification Problem: “Does $P$ satisfy $\varphi$?”

- The functional program $P$ is transformed to a recursion scheme $\tilde{P}$ that generates a tree representing all possible event sequences in $P$.
- $[\tilde{P}]$ is then model checked against (transformed) property $\tilde{\varphi}$, so that $P \models \varphi$ iff $[\tilde{P}] \models \tilde{\varphi}$.

This method is fully automatic, sound and complete.

<table>
<thead>
<tr>
<th>Program Classes</th>
<th>Models of Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>imperative programs + iteration</td>
<td>finite-state automata</td>
</tr>
<tr>
<td>imperative programs + recursion</td>
<td>PDA / boolean programs</td>
</tr>
<tr>
<td>order-$n$ functional programs</td>
<td>order-$n$ recursion schemes</td>
</tr>
</tbody>
</table>
Verification Problem: “Does $P$ satisfy $\varphi$?”

- The functional program $P$ is transformed to a recursion scheme $\tilde{P}$ that generates a tree representing all possible event sequences in $P$.
- $\llbracket \tilde{P} \rrbracket$ is then model checked against (transformed) property $\tilde{\varphi}$, so that $P \models \varphi$ iff $\llbracket \tilde{P} \rrbracket \models \tilde{\varphi}$.

This method is fully automatic, sound and complete.

<table>
<thead>
<tr>
<th>Program Classes</th>
<th>Models of Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>imperative programs + iteration</td>
<td>finite-state automata</td>
</tr>
<tr>
<td>imperative programs + recursion</td>
<td>PDA / boolean programs</td>
</tr>
<tr>
<td>order-$n$ functional programs</td>
<td>order-$n$ recursion schemes</td>
</tr>
</tbody>
</table>
**Scenario.** Higher-order (recursive) functional programs generated from booleans with dynamic resource creation and access primitives.

**Question.** Does program $P$ access each resource $\rho$ according to the given resource specification $\rho^L$, where $L$ is a regular language over the alphabet of resource access primitives.

**Example.** A simple resource specification: “An opened file is eventually closed, and after which it is not read”. So $L = r^*c$.

```ml
let rec g x = if b then close(x)
              else read(x) ; g(x) in
let r = open_in "foo" in g(r)
```

Does the program access the resource foo in accord with $L$?
**Scenario.** Higher-order (recursive) functional programs generated from booleans with dynamic resource creation and access primitives.

**Question.** Does program $P$ access each resource $\rho$ according to the given resource specification $\rho^L$, where $L$ is a regular language over the alphabet of resource access primitives.

**Example.** A simple resource specification: “An opened file is eventually closed, and after which it is not read”. So $L = r^*c$.

```plaintext
let rec g x = if b then close(x)
  else read(x); g(x) in
let r = open_in "foo" in g(r)
```

Does the program access the resource foo in accord with $L$?
1. Transform source program to rec. scheme

\[
\begin{align*}
S & \rightarrow \rho^{r^*c} (G \downarrow d) \\
G \times k & \rightarrow br(c k)(r(G \times k))
\end{align*}
\]

that generates an infinite tree, each of whose path (from root) corresponds to a possible access sequence to resource \( \rho \).

2. Reduce resource usage problem to model checking the scheme against a transformed property given by a trivial automaton.

3. Further reduce model checking problem to a type inference problem.
Resource Usage Verification Problem

**Instance:** A functional program $P$ using resources ($\lambda \rightarrow$ + recursion + booleans + resource creation / access primitives), and specification $\varphi$ (regular expression).

**Question:** Does $P$ use resources in accord with $\varphi$?

**Theorem (Kobayashi + O., ICALP 2009)**

For an order-$n$ source program, the Resource Usage Problem is $(n - 1)$-EXPTIME complete.
Many verification problems reducible to Resource Usage Problem

- **Program Reachability**: “Given a program (closed term of ground type), does its computation reach a special construct fail?”
- Assertion-based verification problems; safety properties
- **Flow Analysis**: “Given a program and its subterms $s$ and $t$, does the value of $s$ flow to the value of $t$?”

An interesting exception!

What is reachability in higher-order functional programs?

### Contextual Reachability

“Given a term $P$ and its (coloured) subterm $N^\alpha$, is there a program context $C[\ ]$ such that evaluating $C[P]$ cause control to flow to $N^\alpha$?”

Many versions of the problem. Connexions with Stirling’s dependency tree automata.

Many verification problems reducible to Resource Usage Problem

- **Program Reachability**: “Given a program (closed term of ground type), does its computation reach a special construct fail?”
- Assertion-based verification problems; safety properties
- **Flow Analysis**: “Given a program and its subterms s and t, does the value of s flow to the value of t?”

An interesting exception!

What is reachability in higher-order functional programs?

**Contextual Reachability**

“*Given a term P and its (coloured) subterm N^α, is there a program context C[] such that evaluating C[P] cause control to flow to N^α?*

Many versions of the problem. Connexions with Stirling’s dependency tree automata.

(See O. + Tzevelekos, “Functional Reachability”, In *Proc. LiCS, 2009*).
Two useful fragments of the modal mu-calculus / APT:

1. Trivial APT ("Safety Fragment"): APT with a singleton priority of 0.
2. Disjunctive APT: APT whose transition function maps each state to a positive boolean formula that is purely disjunctive.

Theorem (Kobayashi + O., ICALP 2009)

1. The Trivial APT Acceptance Problem for order-n recursion schemes is still n-EXPTIME complete.
2. The Disjunctive APT Acceptance Problem for order-n recursion schemes is \((n - 1)\)-EXPTIME complete.

Useful Corollary: The following problems (for order-\(n\) schemes) are \((n - 1)\)-EXPTIME complete:

1. Resource Usage Problem
2. Reachability: “Does \([G]\) have a node labelled by a given symbol?”
Two useful fragments of the modal mu-calculus / APT:

(1) Trivial APT ("Safety Fragment"): APT with a singleton priority of 0.
(2) Disjunctive APT: APT whose transition function maps each state to a positive boolean formula that is purely disjunctive.

Theorem (Kobayashi + O., ICALP 2009)

1. The Trivial APT Acceptance Problem for order-n recursion schemes is still $n$-EXPTIME complete.
2. The Disjunctive APT Acceptance Problem for order-n recursion schemes is $(n-1)$-EXPTIME complete.

Useful Corollary: The following problems (for order-$n$ schemes) are $(n-1)$-EXPTIME complete:

1. Resource Usage Problem
2. Reachability: "Does $[G]$ have a node labelled by a given symbol?"
Two useful fragments of the modal mu-calculus / APT:

1. Trivial APT ("Safety Fragment"): APT with a singleton priority of 0.
2. Disjunctive APT: APT whose transition function maps each state to a positive boolean formula that is purely disjunctive.

Theorem (Kobayashi + O., ICALP 2009)

1. The Trivial APT Acceptance Problem for order-\(n\) recursion schemes is still \(n\)-EXPTIME complete.
2. The Disjunctive APT Acceptance Problem for order-\(n\) recursion schemes is \((n - 1)\)-EXPTIME complete.

Useful Corollary: The following problems (for order-\(n\) schemes) are \((n - 1)\)-EXPTIME complete:

1. Resource Usage Problem
2. Reachability: “Does \([G]\) have a node labelled by a given symbol?”
Preliminary experiments with TRecS (Kobayashi, PPDP 09)

<table>
<thead>
<tr>
<th>Order</th>
<th>Types</th>
<th># Intersection Types (assume 2 states)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(o \rightarrow o)</td>
<td>(2^2 \times 2 = 8)</td>
</tr>
<tr>
<td>2</td>
<td>((o \rightarrow o) \rightarrow o)</td>
<td>(2^8 \times 2 = 512)</td>
</tr>
<tr>
<td>3</td>
<td>(((o \rightarrow o) \rightarrow o) \rightarrow o)</td>
<td>(2^{512} \times 2 = 2^{513} \approx 10^{154})</td>
</tr>
</tbody>
</table>

Example. amscomp/compileenv.ml (40 loc) in OCaml compiler 3.11.0

```ocaml
let read_sect () =
  let fp = open "foo" in
  {readc = fun x -> read fp;
   closec = fun x -> close fp}
let main () =
  let s = read_sect () in s.readc ();
  s.closec ()
```

Result: An order-4 recursion scheme is obtained after “slicing” the source program and CPS transform; # rules = 23, # APT states = 4. Thanks to ingenious optimisation techniques, time to infer types = ? msec.

Preliminary experiments with TRecS (Kobayashi, PPDP 09)

<table>
<thead>
<tr>
<th>Order</th>
<th>Types</th>
<th># Intersection Types (assume 2 states)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>o → o</td>
<td>$2^2 \times 2 = 8$</td>
</tr>
<tr>
<td>2</td>
<td>(o → o) → o</td>
<td>$2^8 \times 2 = 512$</td>
</tr>
<tr>
<td>3</td>
<td>((o → o) → o) → o</td>
<td>$2^{512} \times 2 = 2^{513} \approx 10^{154}$</td>
</tr>
</tbody>
</table>

**Example.** amscomp/compileenv.ml (40 loc) in OCaml compiler 3.11.0

```ml
let read_sect() =
  let fp = open "foo" in
  {
    readc = fun x -> read fp;
    closec = fun x -> close fp
  }
let main() =
  let s = read_sect() in s.readc() ;
  s.closec()
```

**Result:** An order-4 recursion scheme is obtained after “slicing” the source program and CPS transform; # rules = 23, # APT states = 4. Thanks to ingenious optimisation techniques, time to infer types = ? msec.

**Demo.** [http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/](http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/)
**Preliminary experiments with TRecS (Kobayashi, PPDP 09)**

<table>
<thead>
<tr>
<th>Order</th>
<th>Types</th>
<th># Intersection Types (assume 2 states)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( o \rightarrow o )</td>
<td>( 2^2 \times 2 = 8 )</td>
</tr>
<tr>
<td>2</td>
<td>((o \rightarrow o) \rightarrow o)</td>
<td>( 2^8 \times 2 = 512 )</td>
</tr>
<tr>
<td>3</td>
<td>(((o \rightarrow o) \rightarrow o) \rightarrow o)</td>
<td>( 2^{512} \times 2 = 2^{513} \approx 10^{154} )</td>
</tr>
</tbody>
</table>

**Example.** amscomp/compileenv.ml (40 loc) in OCaml compiler 3.11.0

```ocaml
let read_sect() =
  let fp = open "foo" in
  let readc = fun x -> read fp;
  let closec = fun x -> close fp
  in
  let main() =
    let s = read_sect() in s.readc();
    s.closec()
  in main()
```

**Result:** An order-4 recursion scheme is obtained after “slicing” the source program and CPS transform; # rules = 23, # APT states = 4. Thanks to ingenious optimisation techniques, time to infer types = ? msec.

**Demo.** [http://www.kb.ecei.tohoku.ac.jp/~koba/treces/]
An abstract model checking framework (Kobayashi, POPL 2009)

**Input:** (i) Functional program with ground-type values (e.g. \texttt{int}), and dynamic resource creation and access. (ii) Access specification \textit{Spec}.

- **Step 1:** CPS conversion + lambda-lifting
- **Step 2:** Predicate abstraction
- **Step 3:** Conversion to recursion schemes
- **Step 4:** Model checking against \textit{Spec}
- **Step 5:** Real counter-example?
- **Step 6:** Abstraction refinement

- Use model-checking techniques (CEGAR) to abstract info. about \textit{data} (or base values).
- Use type-based techniques to abstract info. about \textit{control} (or function).

Program is safe: property satisfied

Program is unsafe


Conclusions

- Verification of higher-order programs is challenging and worthwhile.
- Recursion schemes are a robust and highly expressive language for infinite structures. Their algorithmic model theory is very rich.
- Recent progress in the theory has been made possible by semantic methods; and new (and highly complex) algorithms extracted.
- Verification of functional programs can be reduced to model checking recursion schemes. The approach is automatic, sound and complete.

Further directions:

1. Is safety a genuine constraint on expressiveness? Equivalently, are order-\(n\) CPDA more expressive than order-\(n\) PDA?
2. Extend verification techniques to call-by-value, polymorphism, pattern matching and recursive data types.
3. Major case study: Develop a fully-fledged model checker for Haskell / OCaml.
**Conclusions**

- Verification of higher-order programs is challenging and worthwhile.
- Recursion schemes are a robust and highly expressive language for infinite structures. Their algorithmic model theory is very rich.
- Recent progress in the theory has been made possible by *semantic methods*; and new (and highly complex) algorithms extracted.
- Verification of functional programs can be reduced to model checking recursion schemes. The approach is automatic, sound and complete.

**Further directions:**

1. **Is safety a genuine constraint on expressiveness?** Equivalently, are order-$n$ CPDA more expressive than order-$n$ PDA?
2. **Extend verification techniques** to call-by-value, polymorphism, pattern matching and recursive data types.
3. **Major case study:** Develop a fully-fledged model checker for Haskell / OCaml.