The Safe $\lambda$-Calculus

William Blum

Joint work with C.-H. Luke Ong

Oxford University Computing Laboratory

BCTCS, 2–5 April 2007
Overview

- **Safety**: a restriction for higher-order grammars.
- Transposed to the $\lambda$-calculus, it gives rise to the **Safe $\lambda$-calculus**.
- Safety has nice algorithmic properties, automata-theoretic and game-semantic characterisations.
What is the Safety Restriction?

First appeared under the name “restriction of derived types” in “IO and OI Hierarchies” by W. Damm, TCS 1982

It is a syntactic restriction for higher-order grammars that constrains the occurrences of the variables in the grammar equations according to their orders.

Theorem (Knapik, Niwiński and Urzyczyn (2001,2002))

1. The Monadic Second Order (MSO) model checking problem for trees generated by safe higher-order grammars of any order is decidable.

2. Automata-theoretic characterisation: Safe grammars of order n are as expressive as pushdown automata of order n.

Aehlig, de Miranda, Ong (2004) introduced the Safe \( \lambda \)-calculus.
What is the Safety Restriction?

- First appeared under the name “restriction of derived types” in “IO and OI Hierarchies” by W. Damm, TCS 1982
- It is a **syntactic restriction** for higher-order grammars that constrains the occurrences of the variables in the grammar equations according to their orders.

**Theorem (Knapik, Niwiński and Urzyczyn (2001,2002))**

1. *The Monadic Second Order (MSO) model checking problem for trees generated by safe higher-order grammars of any order is decidable.*
2. **Automata-theoretic characterisation:** Safe grammars of order $n$ are as expressive as pushdown automata of order $n$.

- Aehlig, de Miranda, Ong (2004) introduced the Safe $\lambda$-calculus.
What is the Safety Restriction?

- First appeared under the name “restriction of derived types” in “IO and OI Hierarchies” by W. Damm, TCS 1982
- It is a syntactic restriction for higher-order grammars that constrains the occurrences of the variables in the grammar equations according to their orders.

Theorem (Knapik, Niwiński and Urzyczyn (2001,2002))

1. The Monadic Second Order (MSO) model checking problem for trees generated by safe higher-order grammars of any order is decidable.

2. Automata-theoretic characterisation: Safe grammars of order n are as expressive as pushdown automata of order n.

- Aehlig, de Miranda, Ong (2004) introduced the Safe λ-calculus.
Simply Typed $\lambda$-Calculus

- **Simple types** $A := o | A \to A$.
- The order of a type is given by $\text{order}(o) = 0$, $\text{order}(A \to B) = \max(\text{order}(A) + 1, \text{order}(B))$.
- Judgements of the form $\Gamma \vdash M : T$ where $\Gamma$ is the context, $M$ is the term and $T$ is the type:

  $$(\text{var}) \quad \frac{x : A \vdash x : A}{\Gamma \vdash x : A}$$

  $$(\text{wk}) \quad \frac{\Gamma \vdash M : A}{\Delta \vdash M : A} \quad \Gamma \subset \Delta$$

  $$(\text{app}) \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

  $$(\text{abs}) \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B}$$

- Example: $f : o \to o \to o, x : o \vdash (\lambda \varphi^o \to o x^o.\varphi\ x)(f\ x)$
- A single rule: $\beta$-reduction. e.g. $(\lambda x^A.M)N \rightarrow_\beta M[N/x]$
Simply Typed $\lambda$-Calculus

- **Simple types** $A := o \mid A \rightarrow A$.
- The **order** of a type is given by $\text{order}(o) = 0$, $\text{order}(A \rightarrow B) = \max(\text{order}(A) + 1, \text{order}(B))$.
- Judgements of the form $\Gamma \vdash M : T$ where $\Gamma$ is the context, $M$ is the term and $T$ is the type:

  \[
  \begin{array}{ccc}
  \text{(var)} & \frac{x : A \vdash x : A}{x : A \vdash x : A} & \text{(wk)} & \frac{\Gamma \vdash M : A}{\Delta \vdash M : A} & \Gamma \subset \Delta \\
  \text{(app)} & \frac{\Gamma \vdash M : A \rightarrow B & \Gamma \vdash N : A}{\Gamma \vdash MN : B} & \text{(abs)} & \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B}
  \end{array}
  \]

- Example: $f : o \rightarrow o \rightarrow o$, $x : o \vdash (\lambda x^o \rightarrow o. x^o. \varphi)(f \ x)$
- A single rule: $\beta$-reduction. e.g. $(\lambda x. M)N \rightarrow^\beta M[N/x]$
Simply Typed $\lambda$-Calculus

- **Simple types** $A := o \mid A \to A$.
- **The order** of a type is given by $\text{order}(o) = 0$, $\text{order}(A \to B) = \max(\text{order}(A) + 1, \text{order}(B))$.
- Judgements of the form $\Gamma \vdash M : T$ where $\Gamma$ is the context, $M$ is the term and $T$ is the type:

\[
\begin{align*}
\text{(var)} & \quad \frac{x : A \vdash x : A}{\text{(wk)}} \quad \frac{\Gamma \vdash M : A}{\Delta \vdash M : A} \quad \frac{\Gamma \subset \Delta}{\text{(app)}} \quad \frac{\Gamma \vdash M : A \to B}{\Gamma \vdash N : A} \quad \frac{\Gamma \vdash MN : B}{\text{(abs)}} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \to B}
\end{align*}
\]

- Example: $f : o \to o \to o, x : o \vdash (\lambda \varphi^o \to o^o. x^o. \varphi^o) (f^o x)$
- A single rule: $\beta$-reduction. e.g. $(\lambda x. M)N \to_\beta M[N/x]$
Simply Typed $\lambda$-Calculus

- **Simple types** $A := o \mid A \rightarrow A$.

- The **order** of a type is given by $\text{order}(o) = 0$, $\text{order}(A \rightarrow B) = \max(\text{order}(A) + 1, \text{order}(B))$.

- Judgements of the form $\Gamma \vdash M : T$ where $\Gamma$ is the context, $M$ is the term and $T$ is the type:

  
  $\frac{}{x : A \vdash x : A}$  \hspace{1cm} (var)

  $\frac{\Gamma \vdash M : A \quad \Delta \vdash M : A}{\Gamma \subset \Delta}$  \hspace{1cm} (wk)

  $\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$  \hspace{1cm} (app)

  $\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B}$  \hspace{1cm} (abs)

- Example: $f : o \rightarrow o \rightarrow o, x : o \vdash (\lambda \varphi^{o\rightarrow o}.x^o.\varphi \ x)(f \ x)$

- A single rule: $\beta$-reduction. e.g. $(\lambda x^A. M)N \rightarrow_\beta M[N/x]$
Simply Typed $\lambda$-Calculus

- **Simple types** $A := o \mid A \rightarrow A$.
- The **order** of a type is given by $\text{order}(o) = 0$, $\text{order}(A \rightarrow B) = \max(\text{order}(A) + 1, \text{order}(B))$.
- Judgments of the form $\Gamma \vdash M : T$ where $\Gamma$ is the context, $M$ is the term and $T$ is the type:

\[
\begin{align*}
\text{var} & : \quad x : A \vdash x : A \\
\text{wk} & : \quad \Delta \vdash M : A \quad \Gamma \subset \Delta
\end{align*}
\]

\[
\begin{align*}
\text{app} & : \quad \Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A \\
& \quad \Gamma \vdash MN : B
\end{align*}
\]

\[
\begin{align*}
\text{abs} & : \quad \Gamma, x : A \vdash M : B \\
& \quad \Gamma \vdash \lambda x^A.M : A \rightarrow B
\end{align*}
\]

- Example: $f : o \rightarrow o \rightarrow o, x : o \vdash (\lambda \varphi^o \rightarrow o^o x^o.\varphi \ x)(f \ x)$

- A single rule: $\beta$-reduction. e.g. $(\lambda x.M)N \rightarrow_\beta M[N/x]$
Variable Capture

The usual “problem” in $\lambda$-calculus: avoid variable capture when performing substitution: $(\lambda x. (\lambda y. x)) y \rightarrow_\beta (\lambda y. x)[y/x] \neq \lambda y. y$

1. Standard solution: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. $(\lambda x. (\lambda y. x)) y$ becomes $(\lambda x. (\lambda z. x)) y$ which reduces to $(\lambda z. x)[y/x] = \lambda z. y$
   Drawback: requires to have access to an unbounded supply of names to perform a given sequence of $\beta$-reductions.

2. Another solution: switch to the $\lambda$-calculus à la de Brujin where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.
   Drawback: the conversion to nameless de Brujin $\lambda$-terms requires an unbounded supply of indices.
   Safety avoids the need for variable renaming!
Variable Capture

The usual “problem” in $\lambda$-calculus: avoid variable capture when performing substitution: $(\lambda x. (\lambda y. x)) y \rightarrow_\beta (\lambda y. x)[y/x] \neq \lambda y. y$

1. **Standard solution**: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. $(\lambda x. (\lambda y. x)) y$ becomes $(\lambda x. (\lambda z. x)) y$ which reduces to $(\lambda z. x)[y/x] = \lambda z. y$

   **Drawback**: requires to have access to an unbounded supply of names to perform a given sequence of $\beta$-reductions.

2. **Another solution**: switch to the $\lambda$-calculus à la de Brujin where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.

   **Drawback**: the conversion to nameless de Brujin $\lambda$-terms requires an unbounded supply of indices.

   Safety avoids the need for variable renaming!
Variable Capture

The usual “problem” in $\lambda$-calculus: avoid variable capture when performing substitution: $(\lambda x. (\lambda y.x))y \rightarrow^\beta (\lambda y.x)[y/x] \neq \lambda y.y$

1. **Standard solution**: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. $(\lambda x. (\lambda y.x))y$ becomes $(\lambda x. (\lambda z.x))y$ which reduces to $(\lambda z.x)[y/x] = \lambda z.y$
   **Drawback**: requires to have access to an unbounded supply of names to perform a given sequence of $\beta$-reductions.

2. **Another solution**: switch to the $\lambda$-calculus à la de Bruijn where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.
   **Drawback**: the conversion to nameless de Bruijn $\lambda$-terms requires an unbounded supply of indices.

Safety avoids the need for variable renaming!
Variable Capture

The usual “problem” in $\lambda$-calculus: avoid variable capture when performing substitution: $(\lambda x.(\lambda y.x))y \rightarrow_\beta (\lambda y.x)[y/x] \neq \lambda y.y$

1. **Standard solution**: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. $(\lambda x.(\lambda y.x))y$ becomes $(\lambda x.(\lambda z.x))y$ which reduces to $(\lambda z.x)[y/x] = \lambda z.y$

   **Drawback**: requires to have access to an unbounded supply of names to perform a given sequence of $\beta$-reductions.

2. **Another solution**: switch to the $\lambda$-calculus à la de Brujin where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.

   **Drawback**: the conversion to nameless de Brujin $\lambda$-terms requires an unbounded supply of indices.

Safety avoids the need for variable renaming!
Variable Capture

The usual “problem” in $\lambda$-calculus: avoid variable capture when performing substitution: $(\lambda x.(\lambda y.x))y \rightarrow_\beta (\lambda y.x)[y/x] \neq \lambda y.y$

1. **Standard solution**: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. $(\lambda x.(\lambda y.x))y$ becomes $(\lambda x.(\lambda z.x))y$ which reduces to $(\lambda z.x)[y/x] = \lambda z.y$

   **Drawback**: requires to have access to an unbounded supply of names to perform a given sequence of $\beta$-reductions.

2. **Another solution**: switch to the $\lambda$-calculus à la de Brujin where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.

   **Drawback**: the conversion to nameless de Brujin $\lambda$-terms requires an unbounded supply of indices.

Safety avoids the need for variable renaming!
Variable Capture

The usual “problem” in \( \lambda \)-calculus: avoid variable capture when performing substitution: \((\lambda x.(\lambda y.x))y \rightarrow_{\beta} (\lambda \_x.\_[y/x]) \neq \lambda y.y\)

1. **Standard solution**: Barendregt’s convention. Variables are renamed so that free variables and bound variables have different names. Eg. \((\lambda x.(\lambda y.x))y\) becomes \((\lambda x.(\lambda z.x))y\) which reduces to \((\lambda z.x)[y/x] = \lambda z.y\)

**Drawback**: requires to have access to an unbounded supply of names to perform a given sequence of \( \beta \)-reductions.

2. **Another solution**: switch to the \( \lambda \)-calculus à la de Brujin where variable binding is specified by an index instead of a name. Variable renaming then becomes unnecessary.

**Drawback**: the conversion to nameless de Brujin \( \lambda \)-terms requires an unbounded supply of indices.

Safety avoids the need for variable renaming!
The Safe $\lambda$-Calculus

The formation rules

\[
\begin{align*}
\text{(var)} & \quad \frac{x : A \vdash s x : A}{\Gamma} \\
\text{(wk)} & \quad \frac{\Gamma \vdash s M : A \quad \Delta \subset \Delta}{\Delta \vdash s M : A}
\end{align*}
\]

\[
\begin{align*}
\text{(app)} & \quad \frac{\Gamma \vdash M : (A_1, \ldots, A_l, B) \quad \Gamma \vdash s N_1 : A_1 \quad \cdots \quad \Gamma \vdash s N_l : A_l}{\Gamma \vdash s MN_1 \ldots N_l : B}
\end{align*}
\]

with the side-condition $\forall y \in \Gamma : \text{ord}(y) \geq \text{ord}(B)$

\[
\begin{align*}
\text{(abs)} & \quad \frac{\Gamma, x_1 : A_1 \ldots x_n : A_n \vdash s M : B}{\Gamma \vdash s \lambda x_1 : A_1 \ldots x_n : A_n. M : A_1 \to \ldots \to A_n \to B}
\end{align*}
\]

with the side-condition $\forall y \in \Gamma : \text{ord}(y) \geq \text{ord}(A_1 \to \ldots \to A_n \to B)$

Property

In the Safe $\lambda$-calculus there is no need to rename variables when performing substitution.
The Safe \( \lambda \)-Calculus

The formation rules

\[
\begin{align*}
\text{(var)} & \quad \frac{x : A \vdash_s x : A}{\text{ }} \\
\text{(wk)} & \quad \frac{\Gamma \vdash_s M : A \quad \Delta \vdash_s M : A}{\Gamma \subset \Delta} \\
\text{(app)} & \quad \frac{\Gamma \vdash M : (A, \ldots, A_l, B) \quad \Gamma \vdash_s N_1 : A_1 \quad \cdots \quad \Gamma \vdash_s N_l : A_l}{\Gamma \vdash_s MN_1 \cdots N_l : B} \\
& \quad \text{with the side-condition } \forall y \in \Gamma : \text{ord}(y) \geq \text{ord}(B)
\end{align*}
\]

\[
\begin{align*}
\text{(abs)} & \quad \frac{\Gamma, x_1 : A_1 \ldots x_n : A_n \vdash_s M : B}{\Gamma \vdash_s \lambda x_1 : A_1 \ldots x_n : A_n. M : A_1 \to \cdots \to A_n \to B} \\
& \quad \text{with the side-condition } \forall y \in \Gamma : \text{ord}(y) \geq \text{ord}(A_1 \to \cdots \to A_n \to B)
\end{align*}
\]

Property

In the Safe \( \lambda \)-calculus there is no need to rename variables when performing substitution.
Examples

- Contracting the $\beta$-redex in the following term

$$f : o \rightarrow o \rightarrow o, x : o \vdash (\lambda x^o \rightarrow x^o. \varphi x)(f \ x)$$

leads to variable capture:

$$(\lambda x. \varphi x)(f \ x) \not\rightarrow^\beta (\lambda x. (f \ x)x).$$

Hence the term is unsafe.

Indeed, $\text{ord}(x) = 0 \leq 1 = \text{ord}(f \ x)$.

- The term $(\lambda x^o \rightarrow x^o. \varphi x)(\lambda y^o. y)$ is safe.
Examples

- Contracting the $\beta$-redex in the following term

$$f : o \rightarrow o \rightarrow o, x : o \vdash (\lambda \varphi^o \rightarrow^o o^o. \varphi \ x)(f \ x)$$

leads to variable capture:

$$(\lambda \varphi x. \varphi \ x)(f \ x) \not\rightarrow_{\beta} (\lambda x. (f \ x) x).$$

Hence the term is unsafe.
Indeed, $\text{ord}(x) = 0 \leq 1 = \text{ord}(f \ x)$.

- The term $(\lambda \varphi^o \rightarrow^o o^o. \varphi \ x)(\lambda y^o. y)$ is safe.
Examples

- Contracting the $\beta$-redex in the following term

$$f : o \rightarrow o \rightarrow o, x : o \vdash (\lambda \varphi^{o \rightarrow o} x^o. \varphi \ x)(f \ x)$$

leads to variable capture:

$$(\lambda \varphi x. \varphi \ x)(f \ x) \not\rightarrow^{\beta} (\lambda x. (f \ x) x).$$

Hence the term is **unsafe**.
Indeed, $\text{ord}(x) = 0 \leq 1 = \text{ord}(f \ x)$. 

- The term $(\lambda \varphi^{o \rightarrow o} x^o. \varphi \ x)(\lambda y^o. y)$ is safe.
Numerical functions

Church Encoding: for \( n \in \mathbb{N} \), \( n = \lambda sz.s^nz \) of type \( I = (o \to o) \to o \to o \).

Theorem (Schwichtenberg 1976)

The numeric function representable by simply-typed terms of type \( I \to \ldots \to I \) are exactly the multivariate polynomials extended with the conditional function:

\[
\text{cond}(t, x, y) = \begin{cases} 
  x, & \text{if } t = 0 \\
  y, & \text{if } t = n + 1 
\end{cases}
\]

\text{cond} can be represented by the unsafe term 
\( \lambda FGH\alpha x.H(\lambda y.G\alpha x)(F\alpha x) \).

In fact \text{cond} is not representable in the Safe \( \lambda \)-calculus:

Theorem

Functions representable by safe \( \lambda \)-expressions of type \( I \to \ldots \to I \) are exactly the multivariate polynomials.
Numerical functions

Church Encoding: for \( n \in \mathbb{N} \), \( \bar{n} = \lambda sz.s^n z \) of type \( I = (o \to o) \to o \to o \).

**Theorem (Schwichtenberg 1976)**

The numeric function representable by simply-typed terms of type \( I \to \ldots \to I \) are exactly the multivariate polynomials extended with the conditional function:

\[
cond(t, x, y) = \begin{cases} 
  x, & \text{if } t = 0 \\
  y, & \text{if } t = n + 1
\end{cases}
\]

\( cond \) can be represented by the *unsafe* term

\[
\lambda FGH\alpha x. H(\lambda y. G\alpha x)(F\alpha x).
\]

In fact \( cond \) is not representable in the Safe \( \lambda \)-calculus:

**Theorem**

Functions representable by safe \( \lambda \)-expressions of type \( I \to \ldots \to I \) are exactly the multivariate polynomials.
Game Semantics

Let $M : T$ be a pure simply typed term.

- **Game-semantics** provides a model of $\lambda$-calculus. $M$ is denoted by a strategy $[M]$ on a game induced by $T$.
- A **strategy** is represented by a set of sequences of moves together with **links**: each move points to a preceding move.
- Computation tree = canonical tree representation of a term.
- Traversals $Trav(M) = \text{sequences of nodes with links respecting some formation rules.}$

The Correspondence Theorem

The game semantics of a term can be represented on the computation tree:

$Trav(M) \cong \langle \langle M \rangle \rangle$

$Reduction(Trav(M)) \cong [M]$ 

where $\langle \langle M \rangle \rangle$ is the revealed game-semantic denotation (i.e. internal moves are uncovered).
Game Semantics

Let $M : T$ be a pure simply typed term.

- **Game-semantics** provides a model of $\lambda$-calculus. $M$ is denoted by a strategy $\llbracket M \rrbracket$ on a game induced by $T$.
- A **strategy** is represented by a set of sequences of moves together with **links**: each move points to a preceding move.
- **Computation tree** = canonical tree representation of a term.
- **Traversals** $Trav(M) = \text{sequences of nodes with links respecting some formation rules.}$

The Correspondence Theorem

The game semantics of a term can be represented on the computation tree:

$$Trav(M) \cong \llbracket M \rrbracket$$

$$Reduction(Trav(M)) \cong \llbracket M \rrbracket$$

where $\llbracket M \rrbracket$ is the revealed game-semantic denotion (i.e. internal moves are uncovered).
Game Semantics

Let $M : T$ be a pure simply typed term.

- Game-semantics provides a model of $\lambda$-calculus. $M$ is denoted by a strategy $[M]$ on a game induced by $T$.
- A strategy is represented by a set of sequences of moves together with links: each move points to a preceding move.
- Computation tree $= $ canonical tree representation of a term.
- Traversals $T r a v(M) = $ sequences of nodes with links respecting some formation rules.

The Correspondence Theorem

The game semantics of a term can be represented on the computation tree:

$$T r a v(M) \cong \langle \langle M \rangle \rangle$$

$$Reduction(T r a v(M)) \cong [M]$$

where $\langle \langle M \rangle \rangle$ is the revealed game-semantic denotation (i.e. internal moves are uncovered).
Game-semantic Characterisation of Safety

- Computation tree of safe terms are \textit{incrementally-bound}: each variable $x$ is bound by the first $\lambda$-node occurring in the path to the root with order $\gt \text{ord}(x)$.

- Using the Correspondence Theorem we can show:

\begin{itemize}
  \item \textbf{Proposition} \\
  Safe terms are denoted by \textit{P-incrementally justified strategies}: each P-move $m$ points to the last O-move in the P-view with order $\gt \text{ord}(m)$.
\end{itemize}

\begin{itemize}
  \item \textbf{Corollary} \\
  Justification pointers attached to P-moves are redundant in the game-semantics of safe terms.
\end{itemize}
Computation tree of safe terms are incrementally-bound: each variable \( x \) is bound by the first \( \lambda \)-node occurring in the path to the root with order \( > \text{ord}(x) \).

Using the Correspondence Theorem we can show:

**Proposition**

Safe terms are denoted by **P-incrementally justified strategies**: each P-move \( m \) points to the last O-move in the P-view with order \( > \text{ord}(m) \).

**Corollary**

Justification pointers attached to P-moves are redundant in the game-semantics of safe terms.
Game-semantic Characterisation of Safety

▶ Computation tree of safe terms are incrementally-bound: each variable $x$ is bound by the first $\lambda$-node occurring in the path to the root with order $> \text{ord}(x)$.

▶ Using the Correspondence Theorem we can show:

**Proposition**
Safe terms are denoted by $\text{P}$-incrementally justified strategies: each $\text{P}$-move $m$ points to the last $\text{O}$-move in the $\text{P}$-view with order $> \text{ord}(m)$.

**Corollary**
Justification pointers attached to $\text{P}$-moves are redundant in the game-semantics of safe terms.
Conclusion and Future Works

Conclusion:
Safety is a syntactic constraint with nice algorithmic and game-semantic properties.

Future works:
- A categorical model of Safe PCF.
- Complexity classes characterised with the Safe λ-calculus?
- Safe Idealized Algol: is contextual equivalence decidable?

Related works:
- Jolie G. de Miranda’s thesis on unsafe grammars.
- Ong introduced computation trees in LICS2006 to prove decidability of MSO theory on infinite trees generated by higher-order grammars (whether safe or not).