Introduction to higher-order verification I
Recursion schemes and terms

Charles Grellois

PPS & LIALA — Université Paris 7

GdT Sémantique et Vérification – November 27th, 2014

Overview

Model-checking higher-order programs

A well-known approach in verification: model-checking.

- Construct a model of a program
- Specify a property in an appropriate logic
- Make them interact in order to determine whether the program satisfies the property.

Interaction is often realized by translating the formula into an equivalent automaton, which then runs over the model.

Need to balance expressivity vs. complexity in the choice of the model and of the logic.

Model-checking higher-order programs

Lunctional languages (such as C++, Haskell, OCaML, Javascript, Python, or Scala) allow and encourage the use of higher-order functions.

Informally, these are functions which can take a function as input:

$$\texttt{compose}\ \phi\ x\ =\ \phi(\phi(x))$$

$\texttt{map}\ f\ l$ applies the function f to every element of the list l

It is a real challenge for verification, as it needs models with recursion of higher-order.

Higher-order recursion schemes (HORS) allow to abstract such programs and precisely model their higher-order behaviour.

Model-checking higher-order programs

Lunctional languages (such as C++, Haskell, OCaML, Javascript, Python, or Scala) allow and encourage the use of higher-order functions.

Informally, these are functions which can take a function as input:

$$\texttt{compose}\ \phi\ x\ =\ \phi(\phi(x))$$

$\texttt{map}\ f\ l$ applies the function f to every element of the list l

It is a real challenge for verification, as it needs models with recursion of higher-order.

Higher-order recursion schemes (HORS) allow to abstract such programs and precisely model their higher-order behaviour.

Model-checking higher-order programs

Verification met semantics with Ong's decidability result (2006):

"It is decidable whether a given MSO formula holds
at the root of the value tree of a higher-order recursion scheme"

which relies on a semantic analysis of the model

A first motivation of this group: in two weeks, you will understand this statement !

Model-checking higher-order programs

Verification met semantics with Ong's decidability result (2006):

"It is decidable whether a given MSO formula holds
at the root of the value tree of a higher-order recursion scheme"

which relies on a semantic analysis of the model

A first motivation of this group: in two weeks, you will understand this statement !

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

More generally, this result – and its proof – were the starting point of many semantic investigations of higher-order and its relations with automata theory:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics, higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz (interpretation in finite models)
- Grellois-Melliès (interpretation in models of linear logic, and in associated type systems)

A motivation of this group is to understand these approaches and how they relate.

Of course, we are more generally interested in every potential meeting point for our communities.

Model-checking higher-order programs

The theoretical study of this problem also lead to the design of model-checkers:

- TRecS and GTRecS (Kobayashi et al.)
- C-SHORe (Broadbent, Carayol, Hague, Serre)
- Preface (Ramsay, Neatherway, Ong)
- others ??

It would be nice to have some talks about practical aspects too. In particular, how do we abstract a program into a recursion scheme in practice ? How helpful is the theoretical understanding of the problem in the implementation of a model-checker ?

Note that none of these model-checkers (to my knowledge) checks the whole MSO logic.

Model-checking higher-order programs

The theoretical study of this problem also lead to the design of model-checkers:

- TRecS and GTRecS (Kobayashi et al.)
- C-SHORe (Broadbent, Carayol, Hague, Serre)
- Preface (Ramsay, Neatherway, Ong)
- others ??

It would be nice to have some talks about practical aspects too. In particular, how do we abstract a program into a recursion scheme in practice ? How helpful is the theoretical understanding of the problem in the implementation of a model-checker ?

Note that none of these model-checkers (to my knowledge) checks the whole MSO logic.

Model-checking higher-order programs

The theoretical study of this problem also lead to the design of model-checkers:

- TRecS and GTRecS (Kobayashi et al.)
- C-SHORe (Broadbent, Carayol, Hague, Serre)
- Preface (Ramsay, Neatherway, Ong)
- others ??

It would be nice to have some talks about practical aspects too. In particular, how do we abstract a program into a recursion scheme in practice ? How helpful is the theoretical understanding of the problem in the implementation of a model-checker ?

Note that none of these model-checkers (to my knowledge) checks the whole MSO logic.

Model-checking higher-order programs

The theoretical study of this problem also lead to the design of model-checkers:

- TRecS and GTRecS (Kobayashi et al.)
- C-SHORe (Broadbent, Carayol, Hague, Serre)
- Preface (Ramsay, Neatherway, Ong)
- others ??

It would be nice to have some talks about practical aspects too. In particular, how do we abstract a program into a recursion scheme in practice ? How helpful is the theoretical understanding of the problem in the implementation of a model-checker ?

Note that none of these model-checkers (to my knowledge) checks the whole MSO logic.

Practical aspects

- A meeting every two weeks or so
- On Thursdays at 16:00, in room 3052
- There will soon be a website (for slides, dates,. . .) and a mailing list

Practical aspects

- A meeting every two weeks or so
- On Thursdays at 16:00, in room 3052
- There will soon be a website (for slides, dates,. . .) and a mailing list

Practical aspects

- To start: two introductory talks — their aim is to fix a common set of concepts and definitions which will not need to be recasted at every talk
- The goal is that everyone understands this common basis of knowledge, so please interrupt this talk everytime you need it !
- Then, from January, "normal" talks will start.
- This group is intended for discussion: interrupting talks with questions will be encouraged in general
- Please contact us if you want to talk ! We are opened to a wide variety of subjects, as soon as they are a meeting point for our communities.

Practical aspects

- To start: two introductory talks — their aim is to fix a common set of concepts and definitions which will not need to be recasted at every talk
- The goal is that everyone understands this common basis of knowledge, so please interrupt this talk everytime you need it !
- Then, from January, "normal" talks will start.
- This group is intended for discussion: interrupting talks with questions will be encouraged in general
- Please contact us if you want to talk ! We are opened to a wide variety of subjects, as soon as they are a meeting point for our communities.

Practical aspects

- To start: two introductory talks — their aim is to fix a common set of concepts and definitions which will not need to be recasted at every talk
- The goal is that everyone understands this common basis of knowledge, so please interrupt this talk everytime you need it !
- Then, from January, "normal" talks will start.
- This group is intended for discussion: interrupting talks with questions will be encouraged in general
- Please contact us if you want to talk ! We are opened to a wide variety of subjects, as soon as they are a meeting point for our communities.

Overview

Higher-order recursion schemes

Idea: it is a kind of grammar whose parameters may be functions and which generates trees.

It is a model which does not interpret conditionals, but generates a tree of all possible behaviours of a program.

A very simple functional program

At first, a recursion scheme looks like a grammar:

$$\begin{array}{lcl} \mathtt{S} & = & \mathtt{L}\ \mathtt{Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ ) \end{array}$$

It produces a tree by substitution, starting from the axiom S.

A very simple functional program

At first, a recursion scheme looks like a grammar:

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L}\ x & = & \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )) \end{array}$$

It produces a tree by substitution, starting from the axiom S.

Value tree of a recursion scheme

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L } x & = & \texttt{if } x \, (\texttt{L } (\texttt{data } x\,) \end{array}$$

generates:

S

Value tree of a recursion scheme

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L}\ x & = & \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ ) \end{array}$$

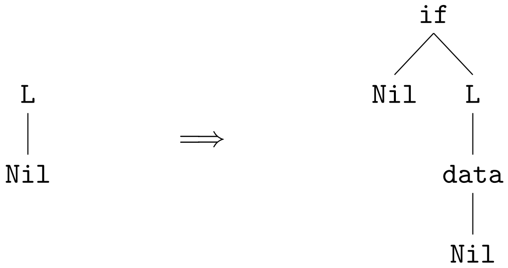
generates:

$$\texttt{S} \quad \Longrightarrow$$

Value tree of a recursion scheme

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )) \end{array}$$

generates:

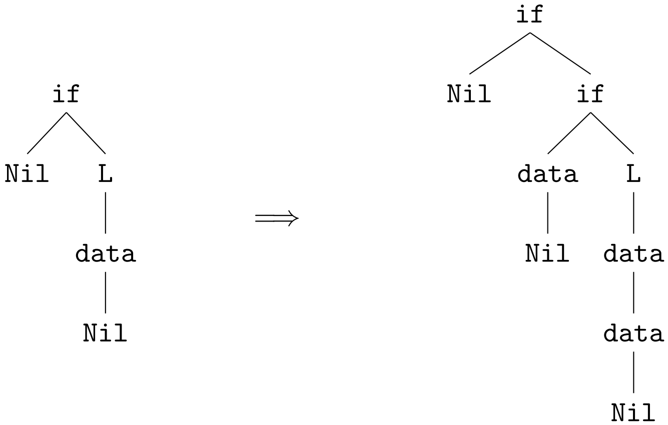


Notice that substitution and expansion occur in one same step.

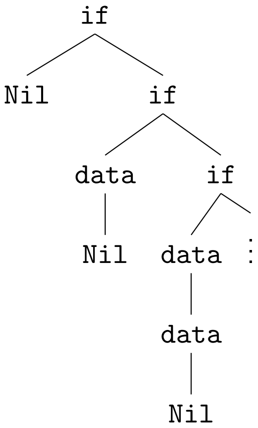
Value tree of a recursion scheme

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ ) \end{array}$$

generates:

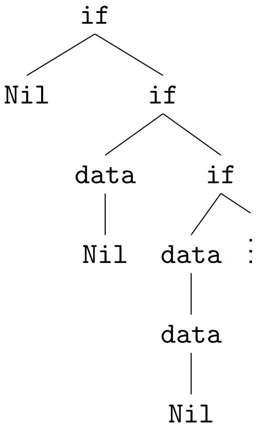


Value tree of a recursion scheme



Important remark: this scheme is very simple, yet it produces a tree which is not regular (it does not have a finite number of subtrees).

Value tree of a recursion scheme



Important remark: this scheme is very simple, yet it produces a tree which is not regular (it does not have a finite number of subtrees).

Recursion schemes: formal definition

We shall define:

- types (as they constraint rules and trees),
- trees (as they are produced by schemes),
- terms (as they appear in the rewriting rules),
- recursion schemes,
- the rewriting relation induced by a recursion scheme,
- and the value tree of a recursion scheme.

Simple types (or kinds)

Kinds are generated by the grammar

$$\kappa \ ::= \ \bot \mid \kappa \to \kappa.$$

By convention, the arrow associates to the right, so every kind may be written

$$\kappa \ = \ \kappa_1 \to \cdots \to \kappa_n \to \bot$$

with n called the arity of κ.

The order $order(\kappa)$ of κ is defined as 0 if $n = 0$ and as $1 + \max(order(\kappa_1), \ldots, order(\kappa_n))$ otherwise.

The set of all kinds is denoted $\mathcal{K}$.

Simple types (or kinds) — terminology

The word kind was proposed by Kobayashi and Ong in their 2009 article on intersection types for verification.

Its purpose is to easily distinguish from types, which they intend as intersection types.

In the sequel, we will try to use the term kind, but will probably sometimes say simple type as well.

(these types and this approach will be the subject of another talk)

Also, $\perp$ is often denoted by o. We will understand this “notation” when we come to linear logic.

Simple types (or kinds) — terminology

The word kind was proposed by Kobayashi and Ong in their 2009 article on intersection types for verification.

Its purpose is to easily distinguish from types, which they intend as intersection types.

In the sequel, we will try to use the term kind, but will probably sometimes say simple type as well.

(these types and this approach will be the subject of another talk)

Also, $\perp$ is often denoted by o. We will understand this "notation" when we come to linear logic.

Simple types (or kinds) — examples

The kind

$$\bot \rightarrow (\bot \rightarrow (\bot \rightarrow \bot))$$

(as formed by the grammar) is also denoted

$$\bot \rightarrow \bot \rightarrow \bot \rightarrow \bot$$

by associativity to the right.

So, its arity is 3.

What about its order ?

Simple types (or kinds) — examples

The kind

$$\perp \rightarrow (\perp \rightarrow (\perp \rightarrow \perp))$$

(as formed by the grammar) is also denoted

$$\perp \rightarrow \perp \rightarrow \perp \rightarrow \perp$$

by associativity to the right.

So, its arity is 3.

What about its order ?

Simple types (or kinds) — examples

The kind

$$\bot \rightarrow (\bot \rightarrow (\bot \rightarrow \bot))$$

(as formed by the grammar) is also denoted

$$\bot \rightarrow \bot \rightarrow \bot \rightarrow \bot$$

by associativity to the right.

So, its arity is 3.

What about its order ?

Simple types (or kinds) — examples

$$\perp \to \perp \to \perp \to \perp$$

Recall the definition of the order of a kind $\kappa = \kappa_1 \to \cdots \to \kappa_n \to \perp$:

The order *order*(κ) of κ is defined as 0 if $n = 0$ and as $1 + \max(\mathit{order}(\kappa_1), \ldots, \mathit{order}(\kappa_n))$ otherwise.

So its order is

$$1 + \max(\mathit{order}(\perp), \mathit{order}(\perp), \mathit{order}(\perp))$$

$$= 1 + \max(0, 0, 0)$$

$$= 1$$

Simple types (or kinds) — examples

$$\perp \to \perp \to \perp \to \perp$$

Recall the definition of the order of a kind $\kappa = \kappa_1 \to \cdots \to \kappa_n \to \perp$:

The order $order(\kappa)$ of κ is defined as 0 if $n = 0$ and as $1 + \max(order(\kappa_1), \ldots, order(\kappa_n))$ otherwise.

So its order is

$$1 + \max(order(\perp), order(\perp), order(\perp))$$

$$= 1 + \max(0, 0, 0)$$

$$= 1$$

Simple types (or kinds) — examples

$$\bot \to \bot \to \bot \to \bot$$

Recall the definition of the order of a kind $\kappa = \kappa_1 \to \cdots \to \kappa_n \to \bot$:

The order $order(\kappa)$ of κ is defined as 0 if $n = 0$ and as $1 + \max(order(\kappa_1), \ldots, order(\kappa_n))$ otherwise.

So its order is

$$1 + \max(order(\bot), order(\bot), order(\bot))$$

$$= 1 + \max(0, 0, 0)$$

$$= 1$$

Simple types (or kinds) — examples

The kind

$$\kappa \ = \ (\ (\ \bot \rightarrow (\ \bot \rightarrow \bot\ ) \rightarrow \bot$$

can not be "simplified" by associativity to the right.

So, its arity is 1.

Its order is

$$\begin{array}{rcl} order(\kappa) & = & 1 + order\,(\ (\ \bot \rightarrow \bot\ ) \rightarrow \bot\ ) \\ & = & 1 + 1 + order(\bot \rightarrow \bot) \\ & = & 1 + 1 + 1 + order(\bot) \\ & = & 1 + 1 + 1 + 0 \\ & = & 3 \end{array}$$

Informally, the order measures the nesting of a type.

Simple types (or kinds) — examples

The kind

$$\kappa \;=\; (\,(\,\perp \rightarrow(\;\perp \rightarrow \perp\,)\rightarrow \perp$$

can not be "simplified" by associativity to the right.

So, its arity is 1.

Its order is

$$\begin{aligned} order(\kappa) &= 1 + order\,(\,(\,\perp \rightarrow \perp\,) \rightarrow \perp\,) \\ &= 1 + 1 + order(\perp \rightarrow \perp) \\ &= 1 + 1 + 1 + order(\perp) \\ &= 1 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

Informally, the order measures the nesting of a type.

Simple types (or kinds) — examples

The kind

$$\kappa \;=\; (\;(\;\perp\,\rightarrow(\;\perp\rightarrow\perp\;)\rightarrow\perp$$

can not be "simplified" by associativity to the right.

So, its arity is 1.

Its order is

$$\begin{array}{rcl} order(\kappa) & = & 1 + order\,(\;(\;\perp\rightarrow\perp\;)\rightarrow\perp\;) \\ & = & 1+1+order(\perp\rightarrow\perp) \\ & = & 1+1+1+order(\perp) \\ & = & 1+1+1+0 \\ & = & 3 \end{array}$$

Informally, the order measures the nesting of a type.

Simple types (or kinds) — examples

The kind

$$\kappa \;=\; (\,(\,\bot \rightarrow (\;\bot \rightarrow \bot\;)\rightarrow \bot$$

can not be "simplified" by associativity to the right.

So, its arity is 1.

Its order is

$$\begin{aligned} order(\kappa) &= 1 + order\,(\,(\,\bot \rightarrow \bot\,) \rightarrow \bot\,) \\ &= 1 + 1 + order(\bot \rightarrow \bot) \\ &= 1 + 1 + 1 + order(\bot) \\ &= 1 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

Informally, the order measures the nesting of a type.

Trees: signatures

We call signature or ranked alphabet a set Σ of constructors together with a function $ar : \Sigma \to \mathbb{N}$ defining the arity of the constructors of the signature.

The arity of a constructor $f \in \Sigma$ may be seen as a kind $\mathrm{kind}(f)$ defined as the kind

$$\bot \to \cdots \to \bot \to \bot$$

of arity $ar(f)$.

Trees

Recursion schemes produce labelled ranked trees.

A Σ-labelled (ranked) tree is defined as a function $t : Dom(t) \rightarrow \Sigma$ with $Dom(t) \subseteq \mathbb{N}^*$ a prefix-closed set of finite words on natural numbers, satisfying the following property:

$$\forall \alpha \in Dom(t),\ \{i \mid \alpha \cdot i \in Dom(t)\} = \{1, \ldots, ar(t(\alpha))\}$$

When this last condition is relaxed, the tree is called unranked – the run-trees of alternating automata will be unranked, as we will see during the next talk.

A branch $b = i_0 \cdots i_n \cdots$ of a tree t is a finite or countable sequence of integers whose prefixes $i_0 \cdots i_n$ are all in $Dom(t)$, and which, if finite, ends on a nullary node: $ar(t(i_0 \cdots i_n)) = 0$.

Trees

Recursion schemes produce labelled ranked trees.

A Σ-labelled (ranked) tree is defined as a function $t : Dom(t) \rightarrow \Sigma$ with $Dom(t) \subseteq \mathbb{N}^*$ a prefix-closed set of finite words on natural numbers, satisfying the following property:

$$\forall \alpha \in Dom(t),\ \{i \mid \alpha \cdot i \in Dom(t)\} = \{1, \ldots, ar(t(\alpha))\}$$

When this last condition is relaxed, the tree is called unranked – the run-trees of alternating automata will be unranked, as we will see during the next talk.

A branch $b = i_0 \cdots i_n \cdots$ of a tree t is a finite or countable sequence of integers whose prefixes $i_0 \cdots i_n$ are all in $Dom(t)$, and which, if finite, ends on a nullary node: $ar(t(i_0 \cdots i_n)) = 0$.

Trees

Recursion schemes produce labelled ranked trees.

A Σ-labelled (ranked) tree is defined as a function $t : Dom(t) \rightarrow \Sigma$ with $Dom(t) \subseteq \mathbb{N}^*$ a prefix-closed set of finite words on natural numbers, satisfying the following property:

$$\forall \alpha \in Dom(t),\ \{i \mid \alpha \cdot i \in Dom(t)\} = \{1, \ldots, ar(t(\alpha))\}$$

When this last condition is relaxed, the tree is called unranked – the run-trees of alternating automata will be unranked, as we will see during the next talk.

A branch $b = i_0 \cdots i_n \cdots$ of a tree t is a finite or countable sequence of integers whose prefixes $i_0 \cdots i_n$ are all in $Dom(t)$, and which, if finite, ends on a nullary node: $ar(t(i_0 \cdots i_n)) = 0$.

Trees

Recursion schemes produce labelled ranked trees.

A Σ-labelled (ranked) tree is defined as a function $t : Dom(t) \rightarrow \Sigma$ with $Dom(t) \subseteq \mathbb{N}^*$ a prefix-closed set of finite words on natural numbers, satisfying the following property:

$$\forall \alpha \in Dom(t),\ \{i \mid \alpha \cdot i \in Dom(t)\} = \{1, \ldots, ar(t(\alpha))\}$$

When this last condition is relaxed, the tree is called unranked – the run-trees of alternating automata will be unranked, as we will see during the next talk.

A branch $b = i_0 \cdots i_n \cdots$ of a tree t is a finite or countable sequence of integers whose prefixes $i_0 \cdots i_n$ are all in $Dom(t)$, and which, if finite, ends on a nullary node: $ar(t(i_0 \cdots i_n)) = 0$.

Trees

Recursion schemes produce labelled ranked trees.

A Σ-labelled (ranked) tree is defined as a function $t : Dom(t) \rightarrow \Sigma$ with $Dom(t) \subseteq \mathbb{N}^*$ a prefix-closed set of finite words on natural numbers, satisfying the following property:

$$\forall \alpha \in Dom(t),\ \{i \mid \alpha \cdot i \in Dom(t)\} = \{1, \ldots, ar(t(\alpha))\}$$

When this last condition is relaxed, the tree is called unranked – the run-trees of alternating automata will be unranked, as we will see during the next talk.

A branch $b = i_0 \cdots i_n \cdots$ of a tree t is a finite or countable sequence of integers whose prefixes $i_0 \cdots i_n$ are all in $Dom(t)$, and which, if finite, ends on a nullary node: $ar(t(i_0 \cdots i_n)) = 0$.

Terms

Recall the example scheme:

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ ) \end{array}$$

We may currify rules, and obtain equivalently

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L} & = & \lambda x.\,\mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ ) \end{array}$$

where $\lambda x.$ is a notation expressing the fact that x is a variable which will be given as argument to the non-terminal L.

Terms

More generally, a rule

$$\texttt{L}\ f\ x\ y \quad = \quad t$$

will be written as

$$\texttt{L} \quad = \quad \lambda f.\ \lambda x.\ \lambda y.\, t$$

meaning that L takes three arguments, the first one being denoted f in the term t, the second one x and the third one y.

Well-kinded terms

Consider a set of variables $\mathcal{V}$, a set of constants $\mathcal{C}$ and a function kind : $\mathcal{V} \cup \mathcal{C} \rightarrow \mathcal{K}$.

The set of well-kinded terms $\Lambda(\mathcal{V}, \mathcal{C})$ is defined inductively:

- $x \in \mathcal{V}$ is a term of kind $\text{kind}(x)$,
- $c \in \mathcal{C}$ is a term of kind $\text{kind}(c)$,
- if t is a term of kind κ and $x \in \mathcal{V}$, $\lambda x.\, t$ is a term of kind $\text{kind}(x) \rightarrow \kappa$,
- if t_1 is a term of kind $\kappa \rightarrow \kappa'$ and t_2 is a term of kind κ, $t_1\ t_2$ is a term of kind κ'.

We extend the function kind to well-kinded terms accordingly.

Well-kinded terms – example

Is the term

$$\lambda x.\,\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

well-kinded ? If so, what is its kind ?

We need first to give kinds to the variables x and L, and to the constants `if` and `data`.

Well-kinded terms – example

Is the term

$$\lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

well-kinded ? If so, what is its kind ?

We need first to give kinds to the variables x and L, and to the constants `if` and `data`.

Well-kinded terms – example

$$\lambda x.\,\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

- $\text{kind}(x)\ =\ \bot$
- $\text{kind}(L)\ =\ \bot \to \bot$
- $\text{kind}(\texttt{if})\ =\ \bot \to \bot \to \bot$
- $\text{kind}(\texttt{data})\ =\ \bot \to \bot$

So that:

- $\text{kind}(\texttt{data}\ x)\ =\ \bot$
- $\text{kind}(L\ (\texttt{data}\ x))\ =\ \bot$
- $\text{kind}(\texttt{if}\ x\ (L\ (\texttt{data}\ x)))\ =\ \bot$
- $\text{kind}(\lambda x.\,\texttt{if}\ x\ (L\ (\texttt{data}\ x)))\ =\ \bot \to \bot$

Well-kinded terms – example

$$\lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

- $\text{kind}(x) \; = \; \bot$
- $\text{kind}(L) \; = \; \bot \to \bot$
- $\text{kind}(\texttt{if}) \; = \; \bot \to \bot \to \bot$
- $\text{kind}(\texttt{data}) \; = \; \bot \to \bot$

So that:

- $\text{kind}(\texttt{data}\ x) \; = \; \bot$
- $\text{kind}(L\ (\texttt{data}\ x)) \; = \; \bot$
- $\text{kind}(\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \; = \; \bot$
- $\text{kind}(\lambda x.\, \texttt{if}\ x\ (L\ (\texttt{data}\ x))) \; = \; \bot \to \bot$

Well-kinded terms – example

$$\lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

- $\text{kind}(x) \ = \ \bot$
- $\text{kind}(L) \ = \ \bot \to \bot$
- $\text{kind}(\texttt{if}) \ = \ \bot \to \bot \to \bot$
- $\text{kind}(\texttt{data}) \ = \ \bot \to \bot$

So that:

- $\text{kind}(\texttt{data}\ x) \ = \ \bot$
- $\text{kind}(L\ (\texttt{data}\ x)) \ = \ \bot$
- $\text{kind}(\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \bot$
- $\text{kind}(\lambda x.\, \texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \bot \to \bot$

Well-kinded terms – example

$$\lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

- $\text{kind}(x) \ = \ \perp$
- $\text{kind}(L) \ = \ \perp \rightarrow \perp$
- $\text{kind}(\texttt{if}) \ = \ \perp \rightarrow \perp \rightarrow \perp$
- $\text{kind}(\texttt{data}) \ = \ \perp \rightarrow \perp$

So that:

- $\text{kind}(\texttt{data}\ x) \ = \ \perp$
- $\text{kind}(L\ (\texttt{data}\ x)) \ = \ \perp$
- $\text{kind}(\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \perp$
- $\text{kind}(\lambda x.\, \texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \perp \rightarrow \perp$

Well-kinded terms – example

$$\lambda x.\,\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

- $\text{kind}(x) \ = \ \bot$
- $\text{kind}(L) \ = \ \bot \rightarrow \bot$
- $\text{kind}(\texttt{if}) \ = \ \bot \rightarrow \bot \rightarrow \bot$
- $\text{kind}(\texttt{data}) \ = \ \bot \rightarrow \bot$

So that:

- $\text{kind}(\texttt{data}\ x) \ = \ \bot$
- $\text{kind}(L\ (\texttt{data}\ x)) \ = \ \bot$
- $\text{kind}(\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \bot$
- $\text{kind}(\lambda x.\,\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \ = \ \bot \rightarrow \bot$

Well-kinded terms – example

Consider a rule

$$L \;=\; \lambda x.\,\texttt{if}\;\; x\;(\texttt{L}\;\;(\texttt{data}\;\; x\;)$$

Since

$$\text{kind}(\lambda x.\,\texttt{if}\; x\;(L\;(\texttt{data}\; x))) \;=\; \bot \to \bot$$

it is natural to ask that $\text{kind}(L) \;=\; \bot \to \bot$.

(in the example we gave, it was automatic since L itself was playing the role of L).

Moreover, this kind has order 1.
This will be the order of this rewriting rule.

Well-kinded terms – example

Consider a rule

$$L \;=\; \lambda x.\,\texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )$$

Since

$$\mathrm{kind}(\lambda x.\,\texttt{if}\ x\ (L\ (\texttt{data}\ x))) \;=\; \bot \to \bot$$

it is natural to ask that $\mathrm{kind}(L) \;=\; \bot \to \bot$.

(in the example we gave, it was automatic since L itself was playing the role of L).

Moreover, this kind has order 1.
This will be the order of this rewriting rule.

Recursion schemes

Consider a set of variables $\mathcal{V}$ and a function kind : $\mathcal{V} \to \mathcal{K}$. A higher-order recursion scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consists of

- a signature Σ,
- a finite set $\mathcal{N}$ of non-terminals with a function kind : $\mathcal{N} \to \mathcal{K}$,
- a function $\mathcal{R} : \mathcal{N} \to \Lambda(\mathcal{V}, \mathcal{N} \cup \Sigma)$ such that, for every $L \in \mathcal{N}$, $\mathcal{R}(L)$ is of the form $\lambda x_1 \cdots \lambda x_n.\, t$, where t is a term without abstractions of kind $\perp$ and which does not contain S, and such that kind(L) = kind$(\mathcal{R}(L))$,
- and of $S \in \mathcal{N}$ of kind $\perp$ called its axiom.

The order of $\mathcal{G}$ is $\max(\{order(\text{kind}(L)) \mid L \in \mathcal{N}\})$.

Recursion schemes

Consider a set of variables $\mathcal{V}$ and a function kind : $\mathcal{V} \rightarrow \mathcal{K}$. A higher-order recursion scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consists of

- a signature Σ,
- a finite set $\mathcal{N}$ of non-terminals with a function kind : $\mathcal{N} \rightarrow \mathcal{K}$,
- a function $\mathcal{R} : \mathcal{N} \rightarrow \Lambda(\mathcal{V}, \mathcal{N} \cup \Sigma)$ such that, for every $L \in \mathcal{N}$, $\mathcal{R}(L)$ is of the form $\lambda x_1 \cdots \lambda x_n.\, t$, where t is a term without abstractions of kind $\perp$ and which does not contain S, and such that
kind(L) = kind$(\mathcal{R}(L))$,
- and of $S \in \mathcal{N}$ of kind $\perp$ called its axiom.

The order of $\mathcal{G}$ is $\max(\{order(\text{kind}(L)) \mid L \in \mathcal{N}\})$.

Order of a recursion schemes

Considering the signature

$$\Sigma \;=\; \{\texttt{if} : 2, \texttt{data} : 1, \texttt{Nil} : 0\}$$

the following set of rules defines a recursion scheme:

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L} & = & \lambda x.\,\texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ ) \end{array}$$

The order of S is 0, the one of L is 1.

So that this recursion scheme is of order 1.
It is why we said it was a very simple one.

Order can be understood as a good measure of the rewriting complexity of a recursion scheme.

Order of a recursion schemes

Considering the signature

$$\Sigma = \{\texttt{if} : 2, \texttt{data} : 1, \texttt{Nil} : 0\}$$

the following set of rules defines a recursion scheme:

$$\begin{array}{lcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L} & = & \lambda x.\,\texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ ) \end{array}$$

The order of S is 0, the one of L is 1.

So that this recursion scheme is of order 1.
It is why we said it was a very simple one.

Order can be understood as a good measure of the rewriting complexity of a recursion scheme.

Another recursion scheme

An example from Serre et al.:

$$
\begin{array}{ccl}
S & = & M\ \mathtt{Nil} \\
M & = & \lambda x.\,\mathtt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\
A & = & \lambda y.\,\lambda \phi.\,\mathtt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ )
\end{array}
$$

with

$$
\Sigma \ = \ \{\mathtt{Nil} : 0,\ \mathtt{if} : 2,\ commit : 1,\ error : 1,\ end : 0,\ cons : 1\}
$$

Exercise: check that terms are well-kinded and compute the order of the scheme.

The answer is that the order is 2.

Another recursion scheme

An example from Serre et al.:

$$\begin{array}{rcl} S & = & M\ \texttt{Nil} \\ M & = & \lambda x.\, \texttt{if}\ (\ \mathit{commit}\ x\ )\ (\ A\ x\ M\ ) \\ A & = & \lambda y.\, \lambda \phi.\, \texttt{if}\ (\ \phi\ (\ \mathit{error}\ \mathit{end}\ )\ )\ (\ \phi\ (\ \mathit{cons}\ y\ )\ ) \end{array}$$

with

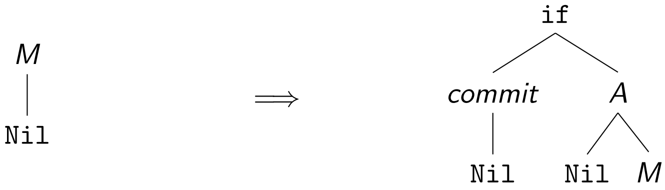
$$\Sigma \ = \ \{\texttt{Nil} : 0,\ \texttt{if} : 2,\ \mathit{commit} : 1,\ \mathit{error} : 1,\ \mathit{end} : 0,\ \mathit{cons} : 1\}$$

Exercise: check that terms are well-kinded and compute the order of the scheme.

The answer is that the order is 2.

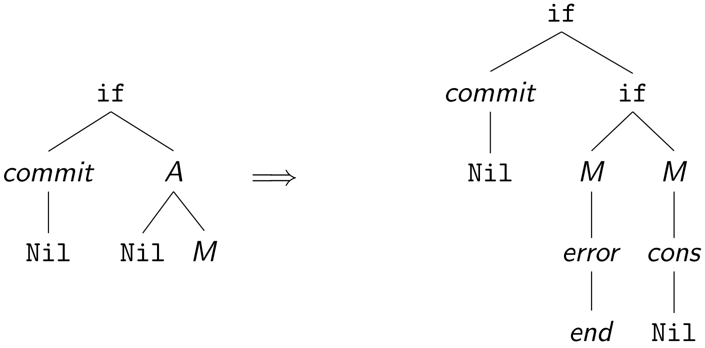
Value tree of a recursion scheme

$$
\begin{array}{rcl}
S & = & M\ \texttt{Nil} \\
M\ x & = & \texttt{if}\ (\ \mathit{commit}\ x\ )\ (\ A\ x\ M\ ) \\
A\ y\ \phi & = & \texttt{if}\ (\ \phi\ (\ \mathit{error}\ \mathit{end}\ )\ )\ (\ \phi\ (\ \mathit{cons}\ y\ )\ )
\end{array}
$$



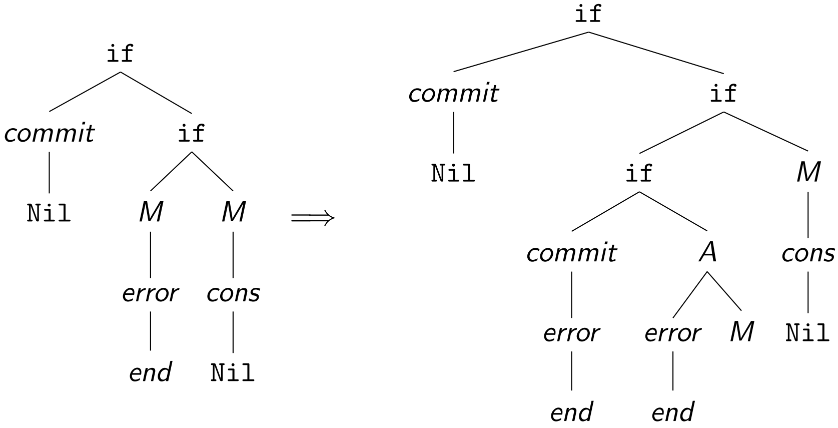
Value tree of a recursion scheme

$$
\begin{aligned}
S &= M\ \texttt{Nil} \\
M\ x &= \texttt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\
A\ y\ \phi &= \texttt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ )
\end{aligned}
$$



Value tree of a recursion scheme

$$
\begin{array}{rcl}
S & = & M\ \texttt{Nil} \\
M\ x & = & \texttt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\
A\ y\ \phi & = & \texttt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ )
\end{array}
$$



Rewriting relation over recursion schemes

To define formally the value tree of a recursion scheme, we need to define how it rewrites.

We define inductively the rewriting relation $\rightarrow_{\mathcal{G}}$ over terms by:

- $L\, t_1 \cdots t_n \;\rightarrow_{\mathcal{G}}\; t[x_i := t_i]$ if $\mathcal{R}(L) = \lambda x_1 \cdots \lambda x_n.\, t$,
- if $s \rightarrow_{\mathcal{G}} t$ then $s\, u \rightarrow_{\mathcal{G}} t\, u$ and $u\, s \rightarrow_{\mathcal{G}} u\, t$.

Informally, recall that a rule

$$L \;=\; \lambda x.\, \lambda y.\, t$$

means that L takes two arguments, that the first one is denoted x in the term t, and the second one is denoted y.

So, in order to evaluate an application $L\ u\ v$, we need to substitute the variable x with the term u, and the variable y with the term v.

Rewriting relation over recursion schemes

To define formally the value tree of a recursion scheme, we need to define how it rewrites.

We define inductively the rewriting relation $\rightarrow_{\mathcal{G}}$ over terms by:

- $L\, t_1 \cdots t_n \;\rightarrow_{\mathcal{G}}\; t[x_i := t_i]$ if $\mathcal{R}(L) = \lambda x_1 \cdots \lambda x_n.\, t$,
- if $s \rightarrow_{\mathcal{G}} t$ then $s\, u \rightarrow_{\mathcal{G}} t\, u$ and $u\, s \rightarrow_{\mathcal{G}} u\, t$.

Informally, recall that a rule

$$L \quad = \quad \lambda x.\, \lambda y.\, t$$

means that L takes two arguments, that the first one is denoted x in the term t, and the second one is denoted y.

So, in order to evaluate an application $L\ u\ v$, we need to substitute the variable x with the term u, and the variable y with the term v.

Rewriting relation over recursion schemes

To define formally the value tree of a recursion scheme, we need to define how it rewrites.

We define inductively the rewriting relation $\rightarrow_{\mathcal{G}}$ over terms by:

- $L\, t_1 \cdots t_n \;\rightarrow_{\mathcal{G}}\; t[x_i := t_i]$ if $\mathcal{R}(L) \;=\; \lambda x_1 \cdots \lambda x_n.\, t$,
- if $s \rightarrow_{\mathcal{G}} t$ then $s\, u \rightarrow_{\mathcal{G}} t\, u$ and $u\, s \rightarrow_{\mathcal{G}} u\, t$.

Informally, recall that a rule

$$L \quad = \quad \lambda x.\, \lambda y.\, t$$

means that L takes two arguments, that the first one is denoted x in the term t, and the second one is denoted y.

So, in order to evaluate an application $L\ u\ v$, we need to substitute the variable x with the term u, and the variable y with the term v.

Divergence

Consider a recursion scheme

$$\begin{array}{lcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S$$

Divergence

Consider a recursion scheme

$$\begin{array}{rcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S$$

Divergence

Consider a recursion scheme

$$\begin{array}{lcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S \rightarrow_{\mathcal{G}} K\ c$$

Divergence

Consider a recursion scheme

$$\begin{array}{rcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S \rightarrow_{\mathcal{G}} K\ c \rightarrow_{\mathcal{G}} K\ (K\ c)$$

Divergence

Consider a recursion scheme

$$\begin{array}{rcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S \rightarrow_{\mathcal{G}} K\ c \rightarrow_{\mathcal{G}} K\ (K\ c) \rightarrow_{\mathcal{G}} K\ (K\ (K\ c))$$

Divergence

Consider a recursion scheme

$$\begin{array}{rcl} S & = & K\ c \\ K & = & \lambda x.\, K\ (K\ x) \end{array}$$

It rewrites as

$$S \rightarrow_{\mathcal{G}} K\ c \rightarrow_{\mathcal{G}} K\ (K\ c) \rightarrow_{\mathcal{G}} K\ (K\ (K\ c)) \rightarrow_{\mathcal{G}} \ldots$$

which never outputs a symbol at its head and thus "never produces anything".

Divergence

Lor this reason, we add a new symbol for divergence.

People from the verification community denote it $\perp$. It is fine for them, as they would give it the kind o.

In semantics, this is the Ω of Böhm trees – in this framework, it always has simple type $\perp$.

In this talk, I will use $\perp$, since it will be clear that it does not represent a kind.

Divergence

Lor this reason, we add a new symbol for divergence.

People from the verification community denote it $\perp$. It is fine for them, as they would give it the kind o.

In semantics, this is the Ω of Böhm trees – in this framework, it always has simple type $\perp$.

In this talk, I will use $\perp$, since it will be clear that it does not represent a kind.

Evaluation policies

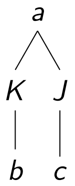
So far, we only considered schemes in which there was only one rewritable non-terminal at the same time.

However, consider a rule

$$L \quad = \quad \lambda x.\,\lambda y.\,a\;(K\;x)\;(J\;y)$$

An example of rewriting:

$$L\;b\;c \qquad \rightarrow_{\mathcal{G}}$$



Which non-terminal should we rewrite first ?

It is not very important in this case, as no rewriting of a non-terminal affects the other.

Evaluation policies

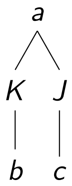
So far, we only considered schemes in which there was only one rewritable non-terminal at the same time.

However, consider a rule

$$L \quad = \quad \lambda x.\, \lambda y.\, a\ (K\ x)\ (J\ y)$$

An example of rewriting:

$L\ b\ c \qquad \rightarrow_{\mathcal{G}}$



Which non-terminal should we rewrite first ?

It is not very important in this case, as no rewriting of a non-terminal affects the other.

Evaluation policies

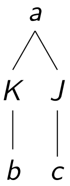
So far, we only considered schemes in which there was only one rewritable non-terminal at the same time.

However, consider a rule

$$L \quad = \quad \lambda x.\, \lambda y.\, a\ (K\ x)\ (J\ y)$$

An example of rewriting:

$L\ b\ c \qquad \rightarrow_{\mathcal{G}}$



Which non-terminal should we rewrite first ?

It is not very important in this case, as no rewriting of a non-terminal affects the other.

Evaluation policies

Consider now the rules

$$\begin{array}{rcl} S & = & J\ (K\ c) \\ K & = & \lambda x.(\,K\ (K\ x)) \\ J & = & \lambda y.\,c \end{array}$$

Which non-terminal should we rewrite first ?

Evaluation policies

Consider now the rules

$$\begin{array}{rcl} S & = & J\ (K\ c) \\ K & = & \lambda x.(\,K\ (K\ x)) \\ J & = & \lambda y.\,c \end{array}$$

Which non-terminal should we rewrite first ?

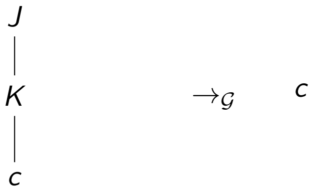
$S \quad \to_{\mathcal{G}}$

Evaluation policies

Consider now the rules

$$\begin{array}{rcl} S & = & J\ (K\ c) \\ K & = & \lambda x.(\,K\ (K\ x)) \\ J & = & \lambda y.\,c \end{array}$$

If we rewrite J:



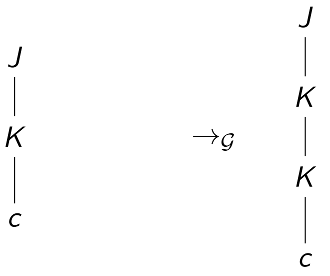
and the evaluation is finished.

Evaluation policies

Consider now the rules

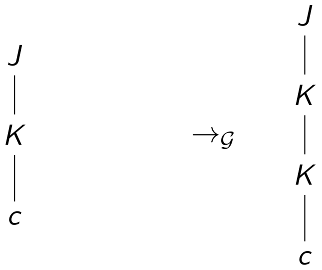
$$\begin{array}{rcl} S & = & J\ (K\ c) \\ K & = & \lambda x.(\ K\ (K\ x)) \\ J & = & \lambda y.\ c \end{array}$$

If we rewrite K:



and we have the same choice again.

Evaluation policies

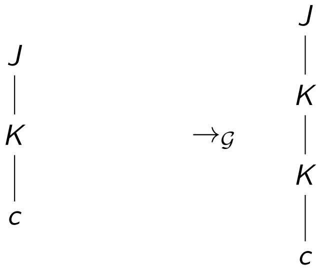


In case we always rewrite K (innermost strategy), the rewriting diverges and produces $\perp$.

Lor more about this, see

Axel Haddad, *IO vs OI in Higher-Order Recursion Schemes*

Evaluation policies



In case we always rewrite K (innermost strategy), the rewriting diverges and produces $\perp$.

Lor more about this, see

Axel Haddad, *IO vs OI in Higher-Order Recursion Schemes*

Value tree of a recursion scheme

We want to define the value tree $[\![\mathcal{G}]\!]$ of the scheme as the one obtained by the "most productive reduction".

The definition is order-theoretical, and hides this notion of reduction.

Value tree of a recursion scheme

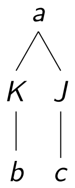
Given a term of $\Lambda(\mathcal{V}, \mathcal{N} \cup \Sigma)$, define $t^\perp$ by induction as follows:

- $a^\perp \;=\; a$ for every $a \in \Sigma$
- $(t_1\ t_2)^\perp \;=\; (t_1)^\perp\ (t_2)^\perp$ if $(t_1)^\perp \;\neq\; \perp$
- in every other case, $t^\perp \;=\; \perp$

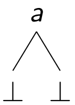
Roughly speaking, seeing t as a tree, $t^\perp$ is obtained from t by replacing every non-terminal by $\perp$ and removing the subtree that was rooted on this non-terminal.

Value tree of a recursion scheme

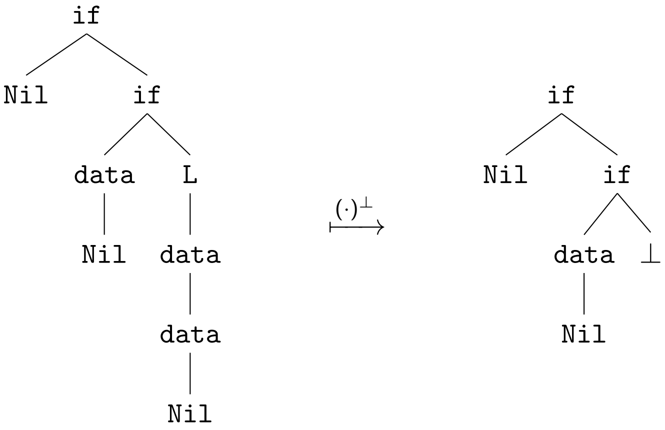
Consider for example



The erasing operation $(\cdot)^{\perp}$ maps it to



Value tree of a recursion scheme



Value tree of a recursion scheme

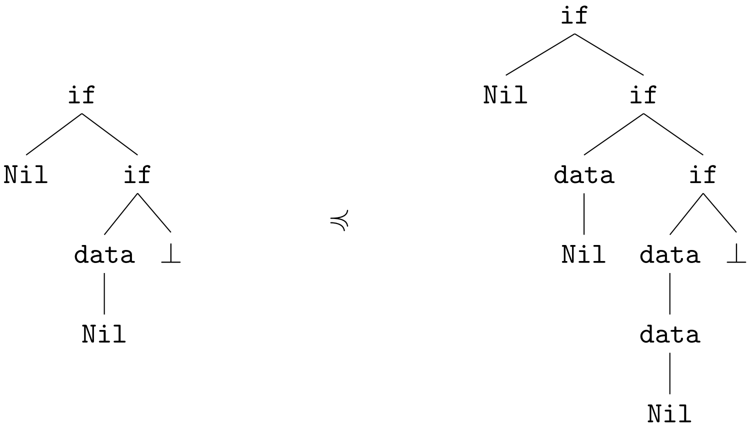
Define the order $\preccurlyeq$ over $\Sigma \uplus \{\perp\}$ by

$$\forall a \in \Sigma \quad \perp \preccurlyeq a$$

and generalize it to $(\Sigma \uplus \{\perp\})$-labelled ranked trees as follows: $t \preccurlyeq t'$ iff $Dom(t) \subseteq Dom(t')$ and

$$\forall w \in Dom(t) \quad t(w) \preccurlyeq t(w')$$

Value tree of a recursion scheme



Value tree of a recursion scheme

The resulting order is a dcpo (directed complete partial order).

Indeed, any non-empty set D of $(\Sigma \uplus \{\bot\})$-labelled ranked trees such that

$$\forall t,\, u \in D \quad \exists v \in D \quad t \preccurlyeq v \text{ and } u \preccurlyeq v$$

has a supremum denoted $\bigvee D$.

Such a set is called a directed set.

Value tree of a recursion scheme

Given a scheme $\mathcal{G}$, its value tree $[\![\mathcal{G}]\!]$ is then defined as

$$[\![\mathcal{G}]\!] \quad = \quad \bigvee \{ t^{\perp} \mid S \rightarrow_{\mathcal{G}}^{*} t \}$$

Exercise: check that this set is directed.

Examples of recursion schemes

Exercise: compute the value tree of the following recursion scheme:

$$\begin{array}{rcl} S & = & L\ c \\ L & = & \lambda x.\, a\ (L\ (b\ x))\ (b\ x) \end{array}$$

Its branch language is $\{a^n b^n c \mid n \geq 1\}$.

Examples of recursion schemes

Exercise: compute the value tree of the following recursion scheme:

$$\begin{array}{rcl} S & = & L\ c \\ L & = & \lambda x.\, a\ (L\ (b\ x))\ (b\ x) \end{array}$$

Its branch language is $\{a^n b^n c \mid n \geq 1\}$.

Examples of recursion schemes

Exercise: compute the value tree of the following recursion scheme:

$$\begin{array}{rcl} S & = & L\ (C\ b\ b) \\ L & = & \lambda\phi.\ a\ (L\ (C\ \phi\ \phi))\ (c\ (\phi\ d)) \\ C & = & \lambda\phi.\ \lambda\psi.\ \lambda x.\ \phi\ (\psi\ x) \end{array}$$

Its branch language is $\{a^n c b^{2^n} d \mid n \geq 1\}$.

Examples of recursion schemes

Exercise: compute the value tree of the following recursion scheme:

$$\begin{array}{rcl} S & = & L\ (C\ b\ b) \\ L & = & \lambda\phi.\, a\ (L\ (C\ \phi\ \phi))\ (c\ (\phi\ d)) \\ C & = & \lambda\phi.\,\lambda\psi.\,\lambda x.\,\phi\ (\psi\ x) \end{array}$$

Its branch language is $\{a^n c b^{2^n} d \mid n \geq 1\}$.

Overview

A quick overview of λY-calculus

Let us first formalize the notion of substitution: in the following situation

$$(\lambda x.\, t)\ u$$

we apply u as argument to the term t, which contains a variable x depicting the argument it requires.

The application of these two terms can be understood as their interaction – which shall result in

$$t[x := u]$$

where in t the occurences of the variable x representing its argument have been replaced by this argument u.

(we do not talk here about free/bounded variables, ... – this part of the talk is more informal)

A quick overview of λY-calculus

Let us first formalize the notion of substitution: in the following situation

$$(\lambda x.\, t)\ u$$

we apply u as argument to the term t, which contains a variable x depicting the argument it requires.

The application of these two terms can be understood as their interaction – which shall result in

$$t[x := u]$$

where in t the occurences of the variable x representing its argument have been replaced by this argument u.

(we do not talk here about free/bounded variables, . . . – this part of the talk is more informal)

A quick overview of λY-calculus

The relation which realizes this interaction is called the β-reduction.
It is defined as:

$$(\lambda x.\, t)\ u \ \rightarrow_\beta\ t[x := u]$$

A quick overview of λY-calculus

Recall that the terms of the set $\Lambda(\mathcal{V}, \mathcal{N} \cup \Sigma)$ are

- well-kinded terms
- where abstractions (λ) are only defined over variables (elements of $\mathcal{V}$).

If we consider instead $\Lambda(\mathcal{V} \cup \mathcal{N}, \Sigma)$, what is the difference ?

We can form terms where non-terminals are abstracted, as

$$\lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

which has kind

$$(\bot \to \bot) \to \bot \to \bot$$

A quick overview of λY-calculus

Recall that the terms of the set $\Lambda(\mathcal{V}, \mathcal{N} \cup \Sigma)$ are

- well-kinded terms
- where abstractions (λ) are only defined over variables (elements of $\mathcal{V}$).

If we consider instead $\Lambda(\mathcal{V} \cup \mathcal{N}, \Sigma)$, what is the difference ?

We can form terms where non-terminals are abstracted, as

$$\lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )$$

which has kind

$$(\perp \to \perp) \to \perp \to \perp$$

A quick overview of λY-calculus

We denote that

$$\lambda L.\, \lambda x.\, \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ::\ (\bot \rightarrow \bot) \rightarrow \bot \rightarrow \bot$$

The notation $t\ ::\ \kappa$ means that $\text{kind}(t) = \kappa$. It was introduced by Kobayashi and Ong, again for mixing intersection types and simple types.

The relation :: can be understood as the simple typing relation.

A quick overview of λY-calculus

Due to the associativity to the right over kinds, the kind

$$(\perp \rightarrow \perp) \rightarrow \perp \rightarrow \perp$$

coincides with the kind

$$(\perp \rightarrow \perp) \rightarrow (\perp \rightarrow \perp)$$

which is of the form $\kappa \rightarrow \kappa$.

A quick overview of λY-calculus

We add to the calculus (to the syntax of terms) a family of operators

$$Y_\kappa \quad :: \quad (\kappa \rightarrow \kappa) \rightarrow \kappa$$

which act as fixpoint. This action is modelled by the relation δ of the λY-calculus:

$$Y\,M \;\rightarrow_\delta\; M\,(Y\,M)$$

A quick overview of λY-calculus

In our example:

$$Y\ (\ \lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )$$

$$\rightarrow_\delta\ \ (\ \lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )\ (Y\ (\ \lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )$$

$$\rightarrow_\beta\ \ \lambda x.\, \texttt{if}\ \ x\ (Y\ (\ \lambda L.\, \lambda x.\, \texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )\ (\texttt{data}\ \ x\ )\ )$$

$$\rightarrow_\delta\ \cdots$$

A quick overview of λY-calculus

In our example:

$$Y \; (\; \lambda L.\, \lambda x.\, \texttt{if} \;\; x \, (\texttt{L} \;\; (\texttt{data} \;\; x\;)\;)\;)$$

$$\rightarrow_\delta \;\; (\; \lambda L.\, \lambda x.\, \texttt{if} \;\; x \, (\texttt{L} \;\; (\texttt{data} \;\; x\;)\;)\;)\; (Y \;\; (\; \lambda L.\, \lambda x.\, \texttt{if} \;\; x \, (\texttt{L} \;\; (\texttt{data} \;\; x\;)\;)\;)$$

$$\rightarrow_\beta \;\; \lambda x.\, \texttt{if} \;\; x \, (Y \;\; (\; \lambda L.\, \lambda x.\, \texttt{if} \;\; x \, (\texttt{L} \;\; (\texttt{data} \;\; x\;)\;)\;)\; (\texttt{data} \;\; x\;)\;)$$

$$\rightarrow_\delta \;\; \cdots$$

A quick overview of λY-calculus

In our example:

$$Y\ (\ \lambda L.\,\lambda x.\,\texttt{if}\ \ x\,(\texttt{L}\ \ (\texttt{data}\ \ x\,)\,)\,)$$

$$\rightarrow_{\delta}\ \ (\ \lambda L.\,\lambda x.\,\texttt{if}\ \ x\,(\texttt{L}\ \ (\texttt{data}\ \ x\,)\,)\,)\,(Y\ \ (\ \lambda L.\,\lambda x.\,\texttt{if}\ \ x\,(\texttt{L}\ \ (\texttt{data}\ \ x\,)\,)\,)$$

$$\rightarrow_{\beta}\ \ \lambda x.\,\texttt{if}\ \ x\,(Y\ \ (\ \lambda L.\,\lambda x.\,\texttt{if}\ \ x\,(\texttt{L}\ \ (\texttt{data}\ \ x\,)\,)\,)\,(\texttt{data}\ \ x\,)\,)$$

$$\rightarrow_{\delta}\ \cdots$$

A quick overview of λY-calculus

In our example:

$$Y\ (\ \lambda L.\lambda x.\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )$$

$$\rightarrow_\delta\ (\ \lambda L.\lambda x.\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )\ (Y\ (\ \lambda L.\lambda x.\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )$$

$$\rightarrow_\beta\ \ \lambda x.\texttt{if}\ \ x\ (Y\ (\ \lambda L.\lambda x.\texttt{if}\ \ x\ (\texttt{L}\ \ (\texttt{data}\ \ x\ )\ )\ )\ (\texttt{data}\ \ x\ )\ )$$

$$\rightarrow_\delta\ \cdots$$

A quick overview of λY-calculus

We obtain a correspondence between recursion schemes and the λ-calculus with a fixpoint operator Y.

In a recursion scheme, the rewriting relation $\rightarrow_{\mathcal{G}}$ corresponds to a particular class of reduction strategies of the λY-calculus where everytime the relation of fixpoint expansion $\rightarrow_{\delta}$, the β-reduction is applied to every position of the term where it can be used (they are called redexes).

This is why in some talks about semantic models we will use a fixpoint operator: in order to give a semantic account of the syntactic recursion given by the rewriting operation of recursion schemes.

Next time. . .

The next session is on December 11th.

We will talk about logic and automata: MSO, modal μ-calculus, alternating parity automata.

Thank you for coming !