From Implicit Complexity to Quantitative Resource Analysis

Complexity Analysis by Program Transformation

Ugo Dal Lago
(Joint work with Martin Avanzini and Georg Moser)

PhD Open, University of Warsaw, June 25-27, 2015
Higher-Order Complexity Analysis

- **The Problem**
  - Given an *higher-order* functional program and a first-order term $M$, evaluate the *asymptotical time complexity* of $M$.
  - We consider the unitary cost model.
  - The problem is *undecidable* in general...
  - ...but decidable approximations exist.

- **Solutions**
  - Type Systems.
  - Cost Functions.
  - Interpretations.

- **Automation**
  - Not so much is known.
  - Tools exits, but most of them are not maintained.
Higher-Order Complexity Analysis

- **The Problem**
  - Given an *higher-order* functional program and a first-order term $M$, evaluate the *asymptotical time complexity* of $M$.
  - We consider the unitary cost model.
  - The problem is *undecidable* in general...
  - ...but decidable approximations exist.

- **Solutions**
  - Type Systems.
  - Cost Functions.
  - Interpretations.

- **Automation**
  - Not so much is known.
  - Tools exits, but most of them are not maintained.
The Problem

- Given a TRS, evaluate its asymptotical time complexity.
- First-order variation on the previous problem

Solutions

- The interpretation Method.
- Path Orders.
- Variations on the dependency pair methods.
- Combinations of the previous methods.

Automation

- Many tools (here dubbed FOPs) exist.
- There is a yearly complexity competition.
The Problem

- Given a TRS, evaluate its asymptotical time complexity.
- First-order variation on the previous problem

Solutions

- The interpretation Method.
- Path Orders.
- Variations on the dependency pair methods.
- Combinations of the previous methods.

Automation

- Many tools (here dubbed FOPs) exist.
- There is a yearly complexity competition.
... Scheme OCaml

complexity certificate
Higher-Order vs. First-Order

- How should we translate higher-order programs into first-order ones?
- We can use program transformations.

\[ \text{\textless P} \text{\textgreater} \rightarrow \{ \cdot \} \rightarrow \text{\textless P} \text{\textgreater} \]

- But under which constraints?
  - They should be correct, i.e., preserve the meaning of programs.
  - They need to be complexity reflecting: \( \langle P \rangle \) is not asymptotically more efficient than \( P \).
  - It would be nice if they were complexity preserving.
- Natural answer: Reynold’s defunctionalization.
let comp f g x = f (g x) ;;

let rev l =
  (* walk :: 'a list → ('a list → 'a list) *)
  let rec walk xs =
    match xs with
    [] → (fun x → x)
  | x :: xs’ →
    comp (walk xs’) (fun ys → x :: ys)
  in walk l [] ;;

let main l = rev l ;;
Example: Plain Defunctionalisation

@((fun_1, x)) → x
@((fun_2(x), ys)) → Cons(x, ys)
@((comp_f_g(f, g), x)) → @((f, @(g, x))
@((comp_f(f), g)) → comp_f_g(f, g)
@((comp, f)) → comp_f(f)
walk_match(Nil) → fun_1
walk_match(Cons(x, xs')) → @((@((comp, @(fix[walk], xs')))), fun_2(x))
@((walk, xs)) → walk_match(xs)
@((fix[walk], xs)) → @(walk, xs)
@((rev, l)) → @((@((fix[walk], l)), Nil))
main(l) → @((rev, l))
Example: Plain Defunctionalisation

@\text{fun}_1(x) \rightarrow x
@\text{fun}_2(x), y \rightarrow \text{Cons}(x, y)
@\text{comp}_f(g)(f,g), x \rightarrow @(f, @(g, x))
@\text{comp}_f(g), x \rightarrow \text{comp}_f(g, f)
@\text{comp}, f \rightarrow \text{comp}_f(f)
\text{walk_match}(\text{Nil}) \rightarrow \text{fun}_1
\text{walk_match}(\text{Cons}(x, x'), x) \rightarrow @(\text{fix[walk]}, \text{xs'}), \text{fun}_2(x)
@\text{walk}, \text{xs} \rightarrow \text{walk_match}(\text{xs})
@\text{fix[walk]}, \text{xs} \rightarrow @\text{walk}, \text{xs}
@\text{rev}, l \rightarrow @@\text{fix[walk]}, l, \text{Nil}
\text{main}(l) \rightarrow @\text{rev}, l
Example: Defunctionalisation + Program Transformations

\[
\begin{align*}
\text{comp}_f \_g_1(f, g, \text{fun}_2(x), y) & \rightarrow \text{Cons}(x, y) \\
\text{comp}_f \_g_1(f, g, \text{fun}_2(x), y) & \rightarrow \\
\text{comp}_f \_g_1(f, g, \text{Cons}(x, y)) & \\
\text{fix}[\text{walk}]_1(\text{Nil}) & \rightarrow \text{fun}_1 \\
\text{fix}[\text{walk}]_1(\text{Cons}(x, xs')) & \rightarrow \\
\text{comp}_f \_g(\text{fix}[\text{walk}]_1(xs'), \text{fun}_2(x)) & \\
\text{main}(\text{Nil}) & \rightarrow \text{Nil} \\
\text{main}(\text{Cons}(x, xs')) & \rightarrow \\
\text{comp}_f \_g_1(\text{fix}[\text{walk}](xs'), \text{fun}_2(x), \text{Nil}) & \\
\end{align*}
\]
Example: Defunctionalisation + Program Transformations

\[
\begin{align*}
\text{comp}_f \circ \text{g}_1(\text{fun}_1, \text{fun}_2(x), ys) & \rightarrow \text{Cons}(x, ys) \\
\text{comp}_f \circ \text{g}_1(\text{comp}_f \circ \text{g}(f, g), \text{fun}_2(x), ys) & \rightarrow \\
& \quad \text{comp}_f \circ \text{g}_1(f, g, \text{Cons}(x, ys)) \\
\text{fix}(\text{walk})_1(\text{Nil}) & \rightarrow \text{fun}_1 \\
\text{fix}(\text{walk})_1(\text{Cons}(x, xs')) & \rightarrow \\
& \quad \text{comp}_f \circ \text{g}(\text{fix}(\text{walk})_1(xs'), \text{fun}_2(x)) \\
\text{main}(\text{Nil}) & \rightarrow \text{Nil} \\
\text{main}(\text{Cons}(x, xs')) & \rightarrow \\
& \quad \text{comp}_f \circ \text{g}_1(\text{fix}(\text{walk})(xs'), \text{fun}_2(x), \text{Nil})
\end{align*}
\]
Our Four Program Transformations

- Inlining
- Dead-Code Elimination
- Instantiation
- Uncurrying
Inlining

@\(\text{fun}_1, x) \rightarrow x\)
@\(\text{fun}_2(x), ys) \rightarrow \text{Cons}(x, ys)\)
@\(\text{comp}_f_g(f, g), x) \rightarrow @f, @g(x)\)
@\(\text{comp}_f(f), g) \rightarrow \text{comp}_f_g(f, g)\)
@\(\text{comp}_f, f) \rightarrow \text{comp}_f(f)\)
\\text{walk}_\text{match}(\text{Nil}) \rightarrow \text{fun}_1\)
\\text{walk}_\text{match}(\text{Cons}(x, xs')) \rightarrow
  @(@\(\text{comp}, @\{\text{fix}[\text{walk}], xs'\}\), \text{fun}_2(x))\)
@\(\text{walk}, xs) \rightarrow \text{walk}_\text{match}(xs)\)
@\(\text{fix}[\text{walk}], xs) \rightarrow @\(\text{walk}, xs\)\)
@\(\text{rev}, l) \rightarrow @(@\{\text{fix}[\text{walk}], l\}, \text{Nil} )\)
\\text{main}(l) \rightarrow @\(\text{rev}, l)\)
Inlining

@\text{fun}_1, x \rightarrow x
@\text{fun}_2(x), ys \rightarrow \text{Cons}(x, ys)
@\text{comp}_f \text{g}(f, g), x \rightarrow @f, @g, x)
@\text{comp}_f(f), g \rightarrow \text{comp}_f \text{g}(f, g)
@\text{comp}, f \rightarrow \text{comp}_f(f)
\text{walk}_\text{match}(\text{Nil}) \rightarrow \text{fun}_1
\text{walk}_\text{match}(\text{Cons}(x, xs')) \rightarrow
  @(@(@\text{comp}, @\text{fix}[\text{walk}], xs'), \text{fun}_2(x))
\underbrace{\text{walk}, xs} \rightarrow \text{walk}_\text{match}(xs)
@\text{fix}[\text{walk}], xs \rightarrow @\text{walk}, xs
@\text{rev}, l \rightarrow @(@\text{fix}[\text{walk}], l), \text{Nil}
\text{main}(l) \rightarrow @\text{rev}, l
Inlining

\( @(\text{fun}_1, x) \rightarrow x \)
\( @(\text{fun}_2(x), ys) \rightarrow \text{Cons}(x, ys) \)
\( @(\text{comp}_f \_g(f, g), x) \rightarrow @(f, @(g, x)) \)
\( @(\text{comp}_f(f), g) \rightarrow \text{comp}_f \_g(f, g) \)
\( @(\text{comp}, f) \rightarrow \text{comp}_f(f) \)

\text{walk\_match}(\text{Nil}) \rightarrow \text{fun}_1
\text{walk\_match}(\text{Cons}(x, xs')) \rightarrow
\begin{align*}
@(@(\text{comp}, @(\text{fix}[\text{walk}], xs'))), \text{fun}_2(x) \end{align*}
\begin{align*}
@(\text{walk}, xs) \rightarrow \text{walk\_match}(xs)
\end{align*}
\begin{align*}
@(\text{fix}[\text{walk}], xs) \rightarrow @(\text{walk}, xs)
\end{align*}
\begin{align*}
@(@(@\text{fix}[\text{walk}], l), \text{Nil})
\end{align*}
\begin{align*}
\text{main}(l) \rightarrow @(\text{rev}, l)
\end{align*}
We need to be careful about guaranteeing the transformation to be **complexity reflecting**.

- We need to avoid “hiding” time complexity under the carpet of inlining.

- We thus restrict to **redex preserving** inlining, i.e. to inlining which does not make the resulting program too efficient compared to the starting one.

- But **where** and **when** should inlining be applied?
  - If done without care, the process can even diverge.

- We apply inlining only if it makes a certain **metric** on programs to decrease.
We need to be careful about guaranteeing the transformation to be complexity reflecting. We need to avoid “hiding” time complexity under the carpet of inlining.

We thus restrict to redex preserving inlining, i.e. to inlining which does not make the resulting program too efficient compared to the starting one.

But where and when should inlining be applied? If done without care, the process can even diverge.

We apply inlining only if it makes a certain metric on programs to decrease.
Inlining

- We need to be careful about guaranteeing the transformation to be complexity reflecting.
  - We need to avoid “hiding” time complexity under the carpet of inlining.
- We thus restrict to redex preserving inlining, i.e. to inlining which does not make the resulting program too efficient compared to the starting one.
- But where and when should inlining be applied?
  - If done without care, the process can even diverge.
- We apply inlining only if it makes a certain metric on programs to decrease.
Inlining

- We need to be careful about guaranteeing the transformation to be complexity reflecting.
  - We need to avoid “hiding” time complexity under the carpet of inlining.
- We thus restrict to redex preserving inlining, i.e. to inlining which does not make the resulting program too efficient compared to the starting one.
- But where and when should inlining be applied?
  - If done without care, the process can even diverge.
- We apply inlining only if it makes a certain metric on programs to decrease.
Dead Code Elimination

fun_1 @ x → x
fun_2(x) @ ys → Cons(x, ys)
comp_f_g(f, g) @ x → f @ (g @ x)
comp_f(f) @ g → comp_f_g(f, g)
comp @ f → comp_f(f)
walk_match(Nil) → fun_1
walk_match(Cons(x, xs')) →
    comp @ (fix[walk] @ xs') @ fun_2(x)
walk @ xs → walk_match(xs)
fix[walk] @ Nil → fun_1
fix[walk] @ Cons(x, xs’) →
    comp_f_g(fix[walk] @ xs’, fun_2(x))
rev @ l → fix[walk] @ l @ Nil
main(l) → fix[walk] @ l @ Nil
Dead Code Elimination

fun_1 @ x → x
fun_2(x) @ ys → Cons(x, ys)
comp_f_g(f, g) @ x → f @ (g @ x)
comp_f(f) @ g → comp_f_g(f, g)
comp @ f → comp_f(f)

walk_match(Nil) → fun_1
walk_match(Cons(x, xs')) →
  comp @ (fix[walk] @ xs') @ fun_2(x)
walk @ xs → walk_match(xs)

fix[walk] @ Nil → fun_1
fix[walk] @ Cons(x, xs') →
  comp_f_g(fix[walk] @ xs', fun_2(x))
rev @ l → fix[walk] @ l @ Nil
main(l) → fix[walk] @ l @ Nil
fun_1 @ x → x
fun_2(x) @ ys → Cons(x, ys)
comp_f_g(f, g) @ x → f @ (g @ x)
fix[walk] @ Nil → fun_1
fix[walk] @ Cons(x, xs’) →
  comp_f_g(fix[walk] @ xs’, fun_2(x))
main(l) → fix[walk] @ l @ Nil
fun_1 @ x → x
fun_2(x) @ ys → Cons(x, ys)
comp_f_g(f, g) @ x → f @ (g @ x)
fix[walk] @ Nil → fun_1
fix[walk] @ Cons(x, xs') →
  comp_f_g(fix[walk] @ xs', fun_2(x))
main(l) → fix[walk] @ l @ Nil
We are interested in the **collecting semantics** of a program.

- For each program point (i.e., for each variable occurring in rewrite rules) which are the terms which can be substituted for it?

Understanding whether a specific term is part of the collecting semantics of (a variable occurrence in) a TRS is **undecidable**.

- It can however be **overapproximated** by a control-flow analysis based on tree automata [Jones2007,KochemsOng2011].
  - ...which, however, needs to be tailored to our needs.
We are interesting in the **collecting semantics** of a program.

- For each program point (i.e., for each variable occurring in rewrite rules) which are the terms which can be substituted for it?

Understanding whether a specific term is part of the collecting semantics of (a variable occurrence in) a TRS is **undecidable**.

- It can however be **overapproximated** by a control-flow analysis based on tree automata [Jones2007,KochemsOng2011].
  - ...which, however, needs to be tailored to our needs.
We are interested in the **collecting semantics** of a program.
- For each program point (i.e., for each variable occurring in rewrite rules) which are the terms which can be substituted for it?
- Understanding whether a specific term is part of the collecting semantics of (a variable occurrence in) a TRS is **undecidable**.
- It can however be **overapproximated** by a control-flow analysis based on tree automata [Jones2007,KochemsOng2011].
  - …which, however, needs to be tailored to our needs.
Uncurrying

\[ \text{fun}_1 \ @ x \to x \]
\[ \text{comp}_f \_g(\text{fun}_1, \text{fun}_2(x')) @ x \to \text{Cons}(x', x) \]
\[ \text{comp}_f \_g(\text{comp}_f \_g(f', g'), \text{fun}_2(x')) @ x \to \]
\[ \quad \text{comp}_f \_g(f', g') @ \text{Cons}(x', x) \]
\[ \text{fix[walk]} @ \text{Nil} \to \text{fun}_1 \]
\[ \text{fix[walk]} @ \text{Cons}(x, xs') \to \]
\[ \quad \text{comp}_f \_g(\text{fix[walk]} @ xs', \text{fun}_2(x)) \]
\[ \text{main}(l) \to \text{fix[walk]} @ l @ \text{Nil} \]
Uncurrying

\[
\begin{align*}
\text{comp}_f g_1 & (\text{fun}_1, \text{fun}_2(x), ys) \rightarrow \text{Cons}(x, ys) \\
\text{comp}_f g_1 & (\text{comp}_f g(f, g), \text{fun}_2(x), ys) \rightarrow \\
& \quad \text{comp}_f g_1(f, g, \text{Cons}(x, ys)) \\
\text{fix}[\text{walk}]_1 & (\text{Nil}) \rightarrow \text{fun}_1 \\
\text{fix}[\text{walk}]_1 & (\text{Cons}(x, xs')) \rightarrow \\
& \quad \text{comp}_f g(\text{fix}[\text{walk}]_1(xs'), \text{fun}_2(x)) \\
\text{main}(\text{Nil}) & \rightarrow \text{Nil} \\
\text{main}(\text{Cons}(x, xs')) & \rightarrow \\
& \quad \text{comp}_f g_1(\text{fix}[\text{walk}](xs'), \text{fun}_2(x), \text{Nil})
\end{align*}
\]
simplify = simpATRS; toTRS; simpTRS  where
simpATRS =
  exhaustive inline(lambda-rewrite);
  exhaustive inline(match);
  exhaustive inline(constructor);
usableRules
toTRS = cfa; uncurry; usableRules
simpTRS = exhaustive ((inline(decreasing); usableRules)
  <> cfaDCE)
Experimental Evaluation

- HoCA has been implemented, and is available online: http://cbr.uibk.ac.at/tools/hoca/
- We built a testbed of around 30 higher-order programs.

<table>
<thead>
<tr>
<th></th>
<th>constant</th>
<th>linear</th>
<th>quadratic</th>
<th>polynomial</th>
<th>terminating</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td># systems</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOP execution time</td>
<td>0.37/1.71/3.05</td>
<td>0.37/4.82/13.85</td>
<td>0.37/4.82/13.85</td>
<td>0.37/4.82/13.85</td>
<td>0.83/1.38/1.87</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td># systems</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HoCA execution time</td>
<td>0.01/2.28/4.56</td>
<td>0.01/0.54/ 4.56</td>
<td>0.01/0.43/ 4.56</td>
<td>0.01/0.42/ 4.56</td>
<td>0.01/0.87/ 6.48</td>
</tr>
<tr>
<td>FOP execution time</td>
<td>0.23/0.51/ 0.79</td>
<td>0.23/2.53/14.00</td>
<td>0.23/6.30/30.12</td>
<td>0.23/10.94/60.10</td>
<td>0.72/1.43/3.43</td>
</tr>
</tbody>
</table>
Some Relevant Cases

- **HO Insertion Sort.**
  - The comparison function is passed as an argument;
  - Correctly dubbed quadratic.

- **HO Merge Sort**
  - A divide and conquer combinator \([\text{Bird}1989]\);
  - FOPs can only prove it terminating.

- **Okasaki’s Parsing Combinators**
  - Really exploits higher-order functions;
  - Hundreds of lines of code;
  - Again, only termination can be proved through FOPs.
Some Relevant Cases

- **HO Insertion Sort.**
  - The comparison function is passed as an argument;
  - Correctly dubbed quadratic.

- **HO Merge Sort**
  - A divide and conquer combinator [Bird1989];
  - FOPs can only prove it terminating.

- **Okasaki’s Parsing Combinators**
  - Really exploits higher-order functions;
  - Hundreds of lines of code;
  - Again, only termination can be proved through FOPs.
Some Relevant Cases

- **HO Insertion Sort**
  - The comparison function is passed as an argument;
  - Correctly dubbed quadratic.

- **HO Merge Sort**
  - A divide and conquer combinator [Bird1989];
  - FOPs can only prove it terminating.

- **Okasaki’s Parsing Combinators**
  - Really exploits higher-order functions;
  - Hundreds of lines of code;
  - Again, only termination can be proved through FOPs.
Conclusions

- We show how to effectively reduce HO analysis to FO analysis.

- An alternative title could be: “The Art of Defunctionalisation for Complexity Analysis”.

- **Pros:**
  - Outperforms, e.g., type systems, in terms of expressivity;
  - Very efficient;
  - Reveals the weaknesses of FOPs.

- **Cons:**
  - Not modular.

- Question: should we drop the “reflection” constraint?
Conclusions

- We show how to effectively reduce HO analysis to FO analysis.
- An alternative title could be: “The Art of Defunctionalisation for Complexity Analysis”.

Pros:
- Outperforms, e.g., type systems, in terms of expressivity;
- Very efficient;
- Reveals the weaknesses of FOPs.

Cons:
- Not modular.

Question: should we drop the “reflection” constraint?
Conclusions

- We show how to effectively reduce HO analysis to FO analysis.
- An alternative title could be: “The Art of Defunctionalisation for Complexity Analysis”.
- **Pros:**
  - Outperforms, e.g., type systems, in terms of expressivity;
  - Very efficient;
  - Reveals the weaknesses of FOPs.
- **Cons:**
  - Not modular.
- **Question:** should we drop the “reflection” constraint?
Conclusions

- We show how to effectively reduce HO analysis to FO analysis.
- An alternative title could be: “The Art of Defunctionalisation for Complexity Analysis”.
- **Pros:**
  - Outperforms, e.g., type systems, in terms of expressivity;
  - Very efficient;
  - Reveals the weaknesses of FOPs.
- **Cons:**
  - Not modular.
- Question: should we drop the “reflection” constraint?
Questions?