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

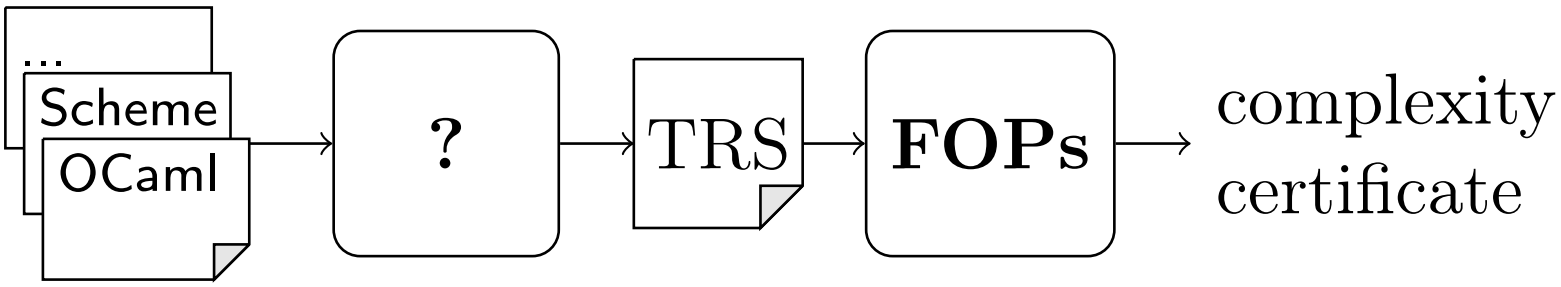
- **The Problem**
  - Given an *higher-order* functional program and a first-order term M, evaluate the *asymptotical time complexity* of M.
  - We consider the unitary cost model.
  - The problem is *undecidable* in general...
  - ...but decidable approximations exist.
- **Solutions**
  - Type Systems.
  - Cost Functions.
  - Interpretations.
- **Automation**
  - Not so much is known.
  - Tools exits, but most of them are not maintained.

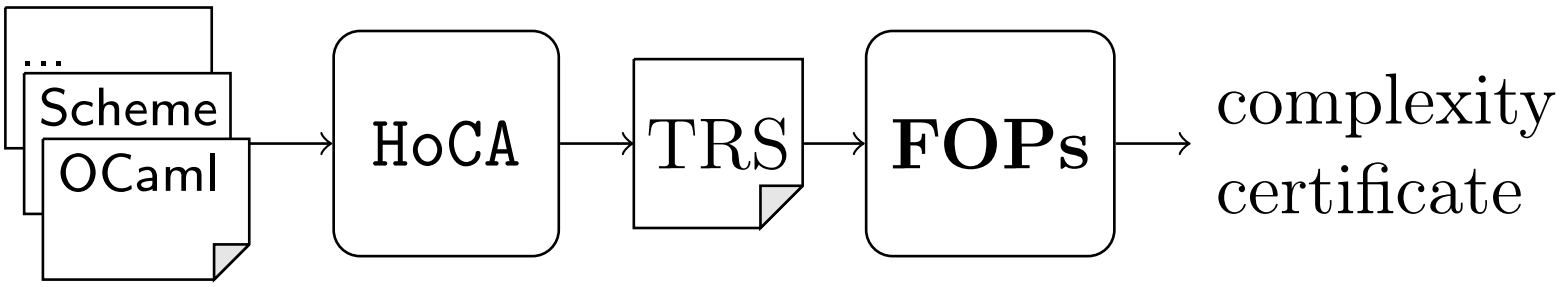
Complexity Analysis of TRSs

- **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.

Complexity Analysis of TRSs

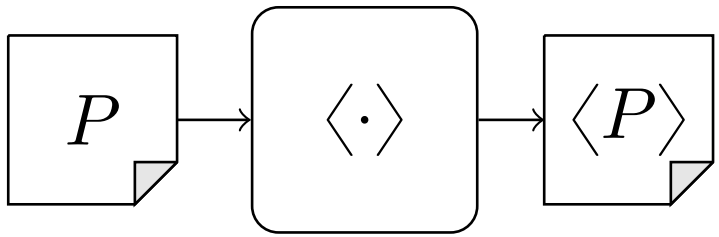
- **The Problem**
  - Given a *TRS*, evaluate its *asymptotical time complexity*.
  - First-order variation on the previous problem
- **Solutions**
  - The interpretation Method.
  - Path Orders.
  - Variations on the dependency pair methods.
  - Combinations of the previous methods.
- **Automation**
  - Many tools (here dubbed FOPs) exist.
  - There is a yearly complexity competition.


...
Scheme
OCaml
?
TRS
FOPs
complexity
certificate


...
Scheme
OCaml
HoCA
TRS
FOPs
complexity
certificate

Higher-Order vs. First-Order

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



- 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**.

Example: Reversing a List

```
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

$$
\begin{aligned}
&\texttt{@(fun\_1,}x\texttt{)} \to x\\
&\texttt{@(fun\_2(}x\texttt{),}ys\texttt{)} \to \texttt{Cons(}x\texttt{,}ys\texttt{)}\\
&\texttt{@(comp\_f\_g(}f\texttt{,}g\texttt{),}x\texttt{)} \to \texttt{@(}f\texttt{,@(}g\texttt{,}x\texttt{))}\\
&\texttt{@(comp\_f(}f\texttt{),}g\texttt{)} \to \texttt{comp\_f\_g(}f\texttt{,}g\texttt{)}\\
&\texttt{@(comp,}f\texttt{)} \to \texttt{comp\_f(}f\texttt{)}\\
&\texttt{walk\_match(Nil)} \to \texttt{fun\_1}\\
&\texttt{walk\_match(Cons(}x\texttt{,}xs\texttt{'))} \to\\
&\quad\texttt{@(@(comp,@(fix[walk],}xs\texttt{')),fun\_2(}x\texttt{))}\\
&\texttt{@(walk,}xs\texttt{)} \to \texttt{walk\_match(}xs\texttt{)}\\
&\texttt{@(fix[walk],}xs\texttt{)} \to \texttt{@(walk,}xs\texttt{)}\\
&\texttt{@(rev,}l\texttt{)} \to \texttt{@(@(fix[walk],}l\texttt{),Nil)}\\
&\texttt{main(}l\texttt{)} \to \texttt{@(rev,}l\texttt{)}
\end{aligned}
$$

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)
```

Don't Know!

Example: Defunctionalisation + Program Transformations

```
comp_f_g1(fun_1,fun_2(x),ys) → Cons(x,ys)
comp_f_g1(comp_f_g(f,g),fun_2(x),ys) →
  comp_f_g1(f,g,Cons(x,ys))
fix[walk]1(Nil) → fun_1
fix[walk]1(Cons(x,xs')) →
  comp_f_g(fix[walk]1(xs'),fun_2(x))
main(Nil) → Nil
main(Cons(x,xs')) →
  comp_f_g1(fix[walk](xs'),fun_2(x),Nil)
```

Example: Defunctionalisation + Program Transformations

```
comp_f_g1(fun_1,fun_2(x),ys) → Cons(x,ys)
comp_f_g1(comp_f_g(f,g),fun_2(x),ys) →
  comp_f_g1(f,g,Cons(x,ys))
fix[walk]1(Nil) → fun_1
fix[walk]1(Cons(x,xs)) →
  comp_f_g(fix[walk]1(xs'),fun_2(x))
main(Nil) → Nil
main(Cons(x,xs')) →
  comp_f_g1(fix[walk](xs'),fun_2(x),Nil)
```

Linear!

Our Four Program Transformations

- Inlining
- Dead-Code Elimination
- Instantiation
- Uncurrying

Inlining

```
@(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)
```

Inlining

```
@(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)
```

Inlining

```
@(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)
```

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.

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
```

Instantiation

```
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
```

Instantiation

```
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
```

Instantation

- 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.

Instantation

- 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.

Instantation

- 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.

Uncurrying

```
fun_1 @ x → x
comp_f_g(fun_1,fun_2(x')) @ x → Cons(x',x)
comp_f_g(comp_f_g(f',g'),fun_2(x')) @ x →
  comp_f_g(f',g') @ Cons(x',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
```

Uncurrying

```
comp_f_g1(fun_1,fun_2(x),ys) → Cons(x,ys)
comp_f_g1(comp_f_g(f,g),fun_2(x),ys) →
  comp_f_g1(f,g,Cons(x,ys))
fix[walk]1(Nil) → fun_1
fix[walk]1(Cons(x,xs')) →
  comp_f_g(fix[walk]1(xs'),fun_2(x))
main(Nil) → Nil
main(Cons(x,xs')) →
  comp_f_g1(fix[walk](xs'),fun_2(x),Nil)
```

Strategy

```
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.

		constant	linear	quadratic	polynomial	terminating
D	# systems	2	5	5	5	8
	FOP execution time	0.37 / 1.71 / 3.05	0.37 / 4.82 / 13.85	0.37 / 4.82 / 13.85	0.37 / 4.82 / 13.85	0.83 / 1.38 / 1.87
S	# systems	2	14	18	20	25
	HOCA execution time	0.01 / 2.28 / 4.56	0.01 / 0.54 / 4.56	0.01 / 0.43 / 4.56	0.01 / 0.42 / 4.56	0.01 / 0.87 / 6.48
	FOP execution time	0.23 / 0.51 / 0.79	0.23 / 2.53 / 14.00	0.23 / 6.30 / 30.12	0.23 / 10.94 / 60.10	0.72 / 1.43 / 3.43

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.

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?

Thank You!

Questions?