TOWARDS RELATING

Program Transformation

Incremental Computation

Parameterized Complexity

Neil D. Jones

Professor Emeritus, University of Copenhagen

2 relevant books:

Computability and complexity from a programming perspective (1997)

Partial evaluation and automatic program generation (1993)

WHY THIS WORKSHOP INTERESTS ME

1. 1970s – 1980s: started in classical complexity theory
   - Problems complete for NLOGSPACE, PTIME, etc.
   - Complexity of finite models; and then program analysis
   - Implicit complexity
2. 1980s – 1990s: shifted to programming languages
   - Complexity of programs with higher-order types
   - Abstract interpretation (semantics-based prog. analysis)
   - Compiler generation
   - Partial evaluation (a central concept: binding times)
   - Constructive uses of program self-application
3. 2009: heard about parameterized complexity from J. Flum

   looked familiar: isn't it binding times again??

A "BIG PICTURE"

About PROBLEMS:

- Mathematical insights
- Hand work

Multivariate complexity

Classical complexity

About PROGRAMS:

- Hand work → Automated

Classical program transformation

Incremental computation

Partial evaluation

Optimising compilers

Speedup

COMPLEXITY AND TRANSFORMATION ARE ABOUT...

Computational Complexity is about problems

Challenges:

- **Better upper bounds = better mathematical methods to solve the same problem**
- **(almost any) lower bounds**
- **Better choice of parameters**

Program Transformation is about programs

Challenges:

- **Greater speedup = faster ways to compute the same input/output function**
- **Greater automation**
- **Better program analysis tools**

COMPUTABILITY (À LA ROGERS, A BIT GENERALISED)

Given: two sets: $Pgms$ **(of programs) and** $Data$**.**

$$Data = Atoms \cup Data \times Data$$

1. **Semantic function:** **(must be Turing-complete)**

$$[\![\_]\!] : Pgms \to Data \to Data_\perp$$

2. **Universal program** $univ$ **such that:** **(a.k.a. self-interpreter)**

$$\boxed{\forall p \in Pgms\ \forall d \in Data\ .\ [\![p]\!](d) = [\![univ]\!](p, d)}$$

3. **Program specialiser** $spec$

$$\boxed{\forall p \in Pgms\ \forall s, d \in Data\ .\ [\![p]\!](s, d) = [\![[\![spec]\!](p, s)]\!](d)}$$

A view: $\underbrace{s}_{\text{static data}}$ **is the parameter and** $\underbrace{d}_{\text{dynamic data}}$ **is the main part of the data.**

Traditionally, $spec$ **is called an S-1-1 program.**

PARTIAL EVALUATION: NONTRIVIAL SPECIALISATION

Build $p_s = [\![spec]\!](p, s)$ by precomputing $[\![p]\!](s, d)$:

1. perform all computations of $[\![p]\!](s, d)$ that depend only on s
2. generate code for all computations that depend on d

- This can take longer to build than the trivial p_s
- But p_s can run substantially faster than $[\![p]\!](s, d)$

A baby example: let $s = 3$, and let p be the program

```
f(n,x) = if n=0 then 1 else x * f(n-1,x)
```

Precomputing all that depends on input $s = n = 3$ gives specialised program p_3:

```
f3(x) = x * x * x * 1
```

HOW SPECIALISATION CAN BE DONE

1. **Binding time analysis** (before specialisation starts):

   annotate as "**static**" all parts of program p that (symbolically) **depend only on** s.

2. During specialisation:

   - **Perform** all of p's actions that were annotated as "static"
   - **Generate code** for all the remaining actions

The baby example, as an annotated program

```
f(n,x) = if n=0 then 1 else x * f(n-1,x)
```

Precomputing all the **green-annotated parts** for static input $s = n = 3$ gives specialised program p_3:

```
f3(x) = x * x * x * 1
```

PARAMETERS AND DIVISIONS

Parameter = the static argument in the S-1-1 theorem:

$$\forall p \; \forall s, d(\; [\![p]\!](s, d) = [\![[\![spec]\!](p, s)]\!](d) \;)$$

A key for multivariate complexity: choose a good parameter, e.g.,

- SAT(vars) versus
- SAT(clause size) versus
- SAT(ones)

Analogue in partial evaluation: to choose a good division (of the input data set by static and dynamic projections).

Another parameter:

The interpreted program when using an interpreter:

$$\forall p \; \forall d(\; [\![p]\!](d) = [\![interp]\!](p, d) \;)$$

PROGRAM TRANSFORMATION BY INTERPRETER SPECIALISATION

Definition A (self-)interpreter is a program $interp$ such that

$$\forall p\ \forall d\ (\ [\![p]\!](d) = [\![interp]\!](p, d)\ )$$

1st Futamura projection: Transform p by specialising the interpreter to p.

Idea: build

$$target := [\![spec]\!](interp, p)$$

Then for any program p and data d we have

$$\begin{aligned} [\![p]\!](d) &= [\![interp]\!](p, d) && \text{By definition of interpreter} \\ &= [\![[\![spec]\!](interp, p)]\!](d) && \text{By definition of specialiser} \\ &= [\![target]\!](d) && \text{By definition of } target \end{aligned}$$

Conclusion: $[\![p]\!] = [\![target]\!]$.

TRANSFORMATION BY SPECIALISATION

The transformation:

$$p \mapsto [\![spec]\!](interp, p)$$

This is an automatic and flexible program transformation.

- **It works for any p**
- **and is (semantics-preserving (for all p) if $interp$ is a correct interpreter)**

How efficient can $p \mapsto p' = [\![spec]\!](univ, p)$ be?

- **Ideal: $time_{p'}(d) \leq time_p(d)$ for any data d. If so, specialisation has removed all interpretational overhead.**
- **We call this "optimal" specialisation.**
- **Optimal specialisation has been achieved for several programming languages.**

LINEAR SPEEDUPS BY PROGRAM TRANSFORMATION

(Subtitle: multivariate functions are tricky.)

A standard example: Knuth-Morris-Pratt pattern matcher

$$match(pat, sub) = \text{if } pat \text{ in } sub \text{ then T else F}$$

Time $O(|pat| \cdot |sub|)$. Specialised version $matcher_{pat}$ can run in time $O(|sub|)$ where the $O(\_)$ does not depend on pat.
Naive matcher:

$$\exists c \; \forall pat \forall sub \; . \; time_{matcher}(pat, sub) \leq c \cdot |pat| \cdot |sub|$$

Effect of specialising:

$$\exists c' \; \forall pat \forall sub \; . \; time_{matcher_{pat}}(sub) \leq c' \cdot |sub|$$

Linear speedup; but bigger for bigger static data.

Theorem The speedups gained by partial evaluation (and supercompilation too) are linear-bounded in this sense.

SUPERLINEAR SPEEDUPS BY TRANSFORMATION

Some superlinear speedups can be got semi-automatically by incremental computation; and automatically by distillation (Hamilton: Turchin's supercompilation carried further).

How? By avoiding the repeated solution of the same subproblems, e.g.,

- Distillation: introduce accumulating parameters. In essence, a static memo table.
- Incremental computation: use a dynamic memo table, as in dynamic programming.

Challenging problems:

- To recognise when repeated subcomputations occur.
- To generalise code so as to increase the number of repeated subcomputations.

SOME USES OF PROGRAMS AND SELF-APPLICATION

Complexity theory: self-application via diagonalisation is the basis for most (all?) standard hierarchy theorems. E.g., show

$$\text{TIME}(n^2) \subsetneq \text{TIME}(2^n)$$

Diagonalise as in proof of unsolvability of the halting problem:

- run an input program p on itself, "with a clock"; and
- produce a result different than $[\![p]\!](p)$, if $time_p(p) \leq |p|^2$.
- Technical trick: do this within the time bound 2^n.

Complete problems SAT, GAP, REGALL, W[i],... proofs: reduce from a time/space/determinism-restricted program/circuit.

Partial evaluation: program self-application is used

- ▶ positively in the Futamura projections
- ▶ (rather than negatively for diagonalisation,)

to speed up program transformers, e.g., compiler generators.

SELF-APPLICATION CAN LEAD TO SPEEDUPS

Yoshihiko Futamura showed two different ways to compile; generate a compiler; and generate a compiler generator:

Proj.	Way 1	Way 2
1. $target$	$:= [\![spec]\!](interp, p)$	$= [\![compiler]\!](p)$
2. $compiler$	$:= [\![spec]\!](spec, interp)$	$= [\![cogen]\!](interp)$
3. $cogen$	$:= [\![spec]\!](spec, spec)$	$= [\![cogen]\!](spec)$

Seen in practical computer experiments: (first in Copenhagen)

- **Each Way 2 run is about 10 times faster than Way 1.**
- **Moral: self-application can generate programs that run faster!**

QUESTIONS, DISCUSSION ?