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

   ▶ Problems complete for $\text{NLOGSPACE}$, $\text{PTIME}$, 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
Given: two sets: \( Pgm{s} \) (of programs) and \( Data \).

\[ Data = Atoms \cup Data \times Data \]

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

   \[
   \llbracket - \rrbracket : Pgm{s} \rightarrow Data \rightarrow Data \perp
   \]

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

   \[
   \forall p \in Pgm{s} \forall d \in Data . \llbracket p \rrbracket (d) = \llbracket univ \rrbracket (p, d)
   \]

3. **Program specialiser** \( spec \)

   \[
   \forall p \in Pgm{s} \forall s, d \in Data . \llbracket p \rrbracket (s, d) = \llbracket \llbracket spec \rrbracket (p, s) \rrbracket (d)
   \]

A view: \( s \) is the **parameter** and \( d \) is the **main part** of the data.

\( s \) is the **static data** and \( d \) is the **dynamic data**

Traditionally, \( spec \) is called an **S-1-1 program**.
PARTIAL EVALUATION: NONTRIVIAL SPECIALISATION

Build $p_s = \llbracket spec \rrbracket(p, s)$ by precomputing $\llbracket p \rrbracket(s, d)$:

1. perform all computations of $\llbracket p \rrbracket(s, d)$ 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 $\llbracket p \rrbracket(s, d)$

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

\[
f(n,x) = \text{if } n=0 \text{ then } 1 \text{ else } x \times f(n-1,x)
\]

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

\[
f_3(x) = x \times x \times x \times 1
\]
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

$$f(n,x) = \text{if } n=0 \text{ then } 1 \text{ else } x \times f(n-1,x)$$

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

$$f_3(x) = x \times x \times x \times 1$$
PARAMETERS AND DIVISIONS

Parameter = the \textbf{static argument} in the S-1-1 theorem:

\[
\forall p \forall s, d( \llbracket p \rrbracket(s, d) = \llbracket \llbracket spec \rrbracket(p, s) \rrbracket(d) )
\]

A key for multivariate complexity: \textbf{choose a good parameter}, e.g.,

\begin{itemize}
  \item \textbf{SAT} (\textit{vars}) \textbf{versus}
  \item \textbf{SAT} (\textit{clause size}) \textbf{versus}
  \item \textbf{SAT} (\textit{ones})
\end{itemize}

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

Another parameter:

The \textbf{interpreted program} when using an interpreter:

\[
\forall p \forall d( \llbracket p \rrbracket(d) = \llbracket interp \rrbracket(p, d) )
\]
Definition A (self-)interpreter is a program \( \text{interp} \) such that
\[
\forall p \forall d ( \llbracket p \rrbracket(d) = \llbracket \text{interp} \rrbracket(p, d) )
\]

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

Idea: build
\[
\text{target} := \llbracket \text{spec} \rrbracket(\text{interp}, p)
\]

Then for any program \( p \) and data \( d \) we have
\[
\llbracket p \rrbracket(d) = \llbracket \text{interp} \rrbracket(p, d)
= \llbracket \llbracket \text{spec} \rrbracket(\text{interp}, p) \rrbracket(d)
= \llbracket \text{target} \rrbracket(d)
\]

Conclusion: \( \llbracket p \rrbracket = \llbracket \text{target} \rrbracket \).
The transformation:

\[ p \mapsto \llbracket spec \rrbracket (interp, p) \]

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 \( p \mapsto p' = \llbracket spec \rrbracket (univ, p) \) 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**

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

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

Naive matcher:

\[
\exists c \ \forall \text{pat} \forall \text{sub} . \ \text{time}_{\text{matcher}}(\text{pat}, \text{sub}) \leq c \cdot |\text{pat}| \cdot |\text{sub}|
\]

Effect of specialising:

\[
\exists c' \ \forall \text{pat} \forall \text{sub} . \ \text{time}_{\text{matcher}_{\text{pat}}}(\text{sub}) \leq c' \cdot |\text{sub}|
\]

Linear speedup; but bigger for bigger static data.

**Theorem** The speedups gained by partial evaluation (and supercompilation too) are linear-bounded in this sense.
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.
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 \( \llbracket p \rrbracket(p) \), if \( \text{time}_p(p) \leq |p|^2 \).
- Technical trick: do this within the time bound \( 2^n \).

Complete problems \( \text{sat}, \text{gap}, \text{regall}, W[i], \ldots \) 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:

<table>
<thead>
<tr>
<th>Proj.</th>
<th>Way 1</th>
<th>Way 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. target := $<a href="interp,p">[spec]</a>$ = $<a href="p">[compiler]</a>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. compiler := $<a href="spec,interp">[spec]</a>$ = $<a href="interp">[cogen]</a>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. cogen := $<a href="spec,spec">[spec]</a>$ = $<a href="spec">[cogen]</a>$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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!