Programs as Data Objects

Neil D. Jones
DIKU, University of Copenhagen

1. Concepts, nature of our craft
2. Operations on programs
3. Broader, deeper and other frameworks
4. Limits to what can be done at all
5. Challenges, dangers, potentials

1. Concepts, nature of our craft

The software crisis

- Explosive growth in computer usage over 30 years
- Growth in trained computer scientists: much less

Obvious solution: AUTOMATE software production: Computer as tool to

- maintain and produce software
- on a large scale
- with minimal human intervention

Required: treat programs *en masse*, as members of collections, **not** as individuals. Analogies:

- Agriculture, manufacturing by Mass production
- New insights needed, just as were essential in engineering/physics for mass production of cars

We're not a physical science

We don't study natural phenomena, but still…

- There are laws and limits to what can be done, e.g., information channel capacity, P versus NP
- We don't discover beauty, but it pays to create it (value of regular versus chaotic development of programs)
- Predictability of program behavior is essential
- Experimentation is necessary once complexity goes over a certain threshhold

<u>The value of abstraction</u>

Research should be either *beautiful* or *useful*. (Too much is neither!)

From a practitioner's perspective:

1. Abstract down to the essence of a concrete problem;
2. Reason on the abstractions, using their internal logic alone;
3. Re-interpret logical conclusions in terms of concrete problem

Practical payoff: harder practical problems can be solved this way

Theoretical payoff: interesting, motivated theoretical problems can be found

Theoretical danger: infinite regress into interesting theoretical problems with less and less motivation

Some current dangers

- There are many pressures towards *quantity* instead of *quality* in training computer scientists
- And many pressures towards *applied research* with immediately visible profits
- There's a large amount of money "out there":
  - Vitalizing
  - Distracting
  - There's a risk of "eating our seed corn" if attract too many top-rate researchers *from university* to *industrial research labs*
- Essential: to preserve *an academic kernel*
  - Focus on central conceptual issues (whatever happened to ESPRIT's "basic research actions" ?)
  - Keep the heritage, maintain the foundational knowledge and world view (we shouldn't *all* solve the Y2K problem)
  - Train future basic researchers

2. Operations on programs

Current State of Software Production

A well developed *handcraft*. Bill Wulf: "high technology basket weaving"

Needed: flexible tools to manipulate programs.

A progression:

1. A program is a *text string*
2. Program = concrete representation of *abstract syntax tree*
3. Systems deal with programs' *meanings, intentions*

Treating programs as data objects

- **Static analysis** of program properties
- **Synthesis** of programs from specifications
- **Verification** that program meets specification, or satisfies boundary conditions
- **Transformation,** e.g., compiling, fixing Year 2000 bugs
- **Optimization**
- **Specialization** of slow general-purpose program to yield faster but more limited versions

The role of semantics

- Programs are *semantics-bearing objects*
- We must account for semantics in program manipulation
- Programs are written in a variety of languages
- New *domain-specific languages*are being devised all the time

Thus we need:

- A deeper understanding of semantics
- Its relation to principles for *systematic and automatic* program construction
- A *common framework* to describe the semantics of groups of languages, and to develop implementations

SLOGAN:
Semantics-based program manipulation

Program properties of interest

- *Correctness:* with respect to a specification
- *Construction:* to satisfy a specification; ideally automated
- *Termination:* the program does
- *Security:* no unwanted side effects (looping, overwriting memory, explosion,...)
- *Resource usage:* acceptable time, memory, etc.
- *Modularity:* all the above can be done one component at a time
- *Modularity:* big, reliable programs can be assembled from simpler parts with reasonable effort, efficiency

A good example: finite automata

- *Synthesis:* There are algorithms to build programs from regular expressions, logical formulas, to and from circuit specifications, many other characterizations
- *Analysis:* computable tests for emptiness, universality, containment, termination,. . .
- *Operations on programs:* intersection, union, complement; minimization; composition,. . .
- *Well-understood variants:* 2-way, infinite behaviors, tree automata,. . .
- *Efficient algorithms:* BDDs,. . .

<u>More examples</u>

Compilers:

- Well-understood front-end and back-end problems to solve
- Some technologies automated reasonably well: lexer and parser generators, retargetable code generators
- "Middle-end": this is semantics, not so well understood or automated
- Some *repeated errors* in spite of understanding achieved elsewhere, e.g., syntax of C, syntax and semantics of C++
- Still *man-years* of work for a new language

Bootstrapping and loading:

- Important but plain, unexciting
- Semantics not so well understood
- Major advances have occurred in portability

Program analysis

- Originally in support of "highly optimizing compilers"
- US: *data flow and control flow analysis*; pragmatically based.
- Europe: *abstract interpretation*, based in language's semantics
- Stronger analyses of more language features (storage, pointers, deadlocks,...)
- *Essential* to compile good code for modern computer architectures
- *More:* hear Patrick Cousot

Partial evaluation

Automatic Program Specialization from:

- a *general program* `pgm` and
- partial (so-called "static") input data `s`

To: a *specialized program* $\texttt{pgm}_\texttt{s}$, which has the known data `s` "built in".

- A limited but completely automatic form of *program transformation*
- Cf. Jørring and Scherlis' *staging transformations*

Example: A two parameter program `p` =

```
f(n,x) = if n = 0 then 1
         else  if n even then f(n/2,x)**2
         else  x*f(n-1,x)
```

$\texttt{p}_5$ = program `p` **specialized to** known $\texttt{n} = 5$:

```
      f5(x) = x*((x**2)**2)
```

Partial evaluation

Correctness:

$$[\![\mathtt{p}]\!]\,\mathtt{s}\,\mathtt{d} \;\mathtt{=}\; [\![\underbrace{[\![\mathtt{specializer}]\!]\,\mathtt{p}\,\mathtt{s}}_{\text{residual program }\mathtt{p_s}}]\!]\,\mathtt{d}$$

- **Why?** To make programs run faster.
- *Speedups:* $\mathtt{p_s}$ may be several times faster than `s`. (Depends on `p`, of course.)
- *Realistic languages:* two partial evaluators for C:
  - C-mix at Copenhagen (good for interpreters)
  - Tempo at Rennes (run-time code generation)
- Some successes, e.g.,
  - Compiling by specializing an interpreter to its source program
  - Operating system programs
- *Extensive program analysis:* abstract interpretation to achieve a staging transformation
- Many conferences and articles, a book

Compilers and Interpreters

$\mathbf{S}$ = a source language
$\mathbf{T}$ = a target language
$\mathbf{L}$ = an implementation language (the default language)

Definition Program `int` is an *interpreter for* $\mathbf{S}$ *written in* $\mathbf{L}$ if for all $\mathtt{p}, \mathtt{d} \in \mathrm{D}$

$$[\![\mathtt{p}]\!]^{\mathbf{S}}\,\mathtt{d} = [\![\mathtt{int}]\!]\mathtt{p}\ \mathtt{d}$$

Program `comp` is an $\mathbf{S}$-*to*-$\mathbf{T}$-*compiler written in* $\mathbf{L}$ if for all $\mathtt{p}, \mathtt{d} \in \mathrm{D}$,

$$[\![\mathtt{p}]\!]^{\mathbf{S}}\,\mathtt{d} = [\![[\![\mathtt{comp}]\!]\mathtt{p}]\!]^{\mathbf{T}}\,\mathtt{d}$$

Compiling and Compiler Generation

Futamura projections: compiling and compiler generation can be done by self-applying a partial evaluator.

Compilation, compiler generation and compiler generator generation can each be done two ways. The following have been realized in practice (non-trivially):

$$\begin{aligned}
\texttt{target} &= [\![\texttt{spec}]\!]\texttt{int source}\\
&= [\![\texttt{compiler}]\!]\texttt{source}\\[1em]
\texttt{compiler} &= [\![\texttt{spec}]\!]\texttt{spec int}\\
&= [\![\texttt{cogen}]\!]\texttt{int}\\[1em]
\texttt{cogen} &= [\![\texttt{spec}]\!]\texttt{spec spec}\\
&= [\![\texttt{cogen}]\!]\texttt{spec}
\end{aligned}$$

Exact timings vary with the partial evaluator and the implementation language **L** but:

The second way is often about 10 times faster than the first.

3. Broader, deeper and other frameworks

Pattern matching

The **Knuth-Morris-Pratt** pattern matching algorithm was a direct outgrowth of Stephen Cook's (very) abstract theorem on *2-way deterministic pushdown automata*.

Stripped of its theoretical garb, Cook's result defined a *programming language* such that any program written it in can be simulated in linear time. Even though it may *run* in exponential time(!)

Program transformation

The **Burstall-Darlington** transformation framework can yield major program speedups, but has not yet been fully automated.

Steps towards automation:

- Partial evaluation
- Supercompilation (Turchin, Sørensen, Glück)
- Transformers by Paige, Liu, Pettorossi,. . .

Automatic verification/validation

Application areas: circuits, programs, embedded software,. . .

- Object of study is an active computing agent
- Rapid progress, large systems analyzed
- Combination of techniques
  - Model checking
  - Abstract interpretation
  - Automata on finite or infinite words
  - Modal logic
  - BDD's, symbolic formula manipulation,. . .
- Increasing interaction with other research areas
- An emerging intellectual community

Lambda calculus, rewriting systems

- Mathematically beautiful
- Largely European
- Becoming more algorithmic, related tp programming languages
- Getting more relevant to practice

Category theory, linear logic

- Mathematically deep, sophisticated
- Largely European
- Have computational overtones
- Has shown promise for some years, not yet proven
- When will these frameworks be exploited computationally?
  - Linear logic and tracing of resources, e.g., storage
  - CAM, the *categorical abstract machine*
  - Functor categoies? (Reynolds, Moggi)

4. Limits to what can be done at all

Finite automata: well-understood theory of *regular sets*. Example: no finite automaton whatever can decide membership in

$$\{0^n 1^n \mid n \geq 0\}$$

Complexity and programming languages:

- *Expressiveness:* what language features are necessary to solve which problems?
- A *top-down* approach: "you can't beat a lower bound," no matter how good you are at programming, data design, or mathematical analysis.
- Focus on the *problem itself*, regardless of how (or whether) it can be solved. Quite unusual in computer science.
- Example: CONS-free programs with nonfunctional data can solve exactly the decision problems in PTIME.

  With first-order data, they can solve exactly the decision problems in EXPTIME.

Outside limitations and pressures

- *IT for the masses!* Great public press to train more programmers, produce software engineers, re-educate people now in place,. . .
- An insidious effect: the *loss of abstraction* in our university curricula.

  This is not "theory for theory's sake," but the ability to extract the essence from a practical problem, solve this core, and then re-embed the solution into the original context.

  Many factors seem to conspire against this skill.
- *Risk of losing accumulated knowledge.* Motivated by commercial pressures, curricula tend to fill up with: *JAVA, web page creation, GUI design,. . .* – ond so push out central concepts such as:

  parsing and grammar theory, complexity, logic, automata,. . .
- *A major challenge* is to preserve this accumulated knowledge.

Conclusions

- A great deal has happened in recent years concerning the treatment of *programs as data objects*.
- There is a danger that the theoretical knowledge basis to continue this level of work may be hindered by social and economic pressures.
- As academic researchers, we should continue to do the kind of work we are good at (abstraction, frameworks, interplay between systems, etc.)
- We should "hold the fort," and not let good parts of today's curricula get lost.