Program Analysis for Implicit Computational Complexity

Neil D. Jones

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: neil@diku.dk

This talk brings together ideas from two lines: *automatic estimation of program running times*, and *implicit computational complexity*. It describes ongoing research. Recent work in the two areas has been done by Bellantoni and Cook, Benzinger, Hofmann, Jones, Crary and Weirich, Leivant, Marion, Schwichtenberg, and others.

A main goal of implicit computational complexity is to "capture" complexity classes such as PTIMEF (polynomial-time computable functions) by computing formalisms that do not impose explicit bounds on time or space resources. Several researchers have succeeded in reaching this goal, a well-known example being the Bellantoni-Cook "safe primitive recursion on notation."

It must be said, however, that recursion-theoretic formalisms such as primitive recursion are not very close to programming practice. In particular natural *algorithms*, as seen in introductory algorithm courses, often do not fall into existing implicit complexity classes. In some cases this has even been proven impossible, e.g., Colson established that primitive recursion alone cannot express computing the minimum of two numbers by the obvious linear-time algorithm.

In this work we identify a decidable class of algorithms such that all can be executed within polynomial time (or logarithmic space); and as well, the class includes many natural algorithms that are used in solving real problems.

For a standard first-order functional language we devise a type system giving information on the *variations* of its function parameters in terms of program inputs, and on *run-time bounds* for program-defined functions. Every syntactically correct program is well-typed, i.e., the language has a so-called "soft" type system.

The type information is extracted by data-flow analysis algorithms that extend the "size-change" framework of our POPL 2001 paper to account for running times as well as termination. The analysis allows automatic detection of programs that are guaranteed to run (or be runnable) in polynomial time.

Theorems are proven that this is indeed the case; and that the class is a proper generalization of "safe recursion" and some related schemes provided by other researchers. Several representative natural and efficient algorithms are seen to fall into the class, providing evidence that the class is "large enough."