Tentative List of Project Offerings by TOPPS Members, Spring 2009

2. februar 2009

The current document is a list of bachelor thesis and master project offerings by TOPPS group members for quarters 1 and 2 2009.

The current list has been finalized Tuesday, January 13.

Each project description occurs as outlined by the supervisor; there is at present no common format for descriptions.

1 Andrzej Filinski

1.1 Implementation of Scan-Based Data Parallelism on CUDA

The vector-scan model is a general framework for expressing data-parallel algorithms in a deterministic setting. It is based on a combination of elementwise operations (arithmetic and scatter/gather primitives) and so-called "scans" or "prefix sums", which transform a vector of numbers (e.g., [1,5,2,6,3,2]) into a vector of the sums of all initial segments of that vector ([1,6,8,14,17,19]). While this may at first appear to be an inherently sequential operation, it can actually be efficiently parallelized across multiple processing elements. Moreover, that parallelization strategy works for any associative operation, not only addition.

Parallel scans enable efficient parallelization of a wide range of higher-level fundamental operations, such as packing all the elements of a vector that satisfy some property into a shorter vector. Most notably, however, they also play an important role in so-called "nested data parallelism", in which elementwise parallel operations are preformed not on atomic data, but on vectors of inner vectors, not necessarily of the same size. For example, given the vector of integer vectors [[1,5,2,6], [3,2]], we may want to compute their individual scans [[1,6,8,14], [3,5]] while still keeping all processing elements busy. The experimental nested data-parallel language NESL is based on an ML-like functional language extended with vector-scan operations and allows many classical parallel algorithms to be expressed at a very high level of abstraction, yet still be implemented efficiently.

Project:

Implement a library of vector-scan operations on top of nVidia's CUDA library for general-purpose computation on graphics processors, and assess its potential as a platform for executing data-parallel algorithms. A larger project (multi-person or M.Sc. thesis) might investigate retargeting the NESL compiler – which originally targeted several 1990s-era supercomputer architectures – to modern commodity hardware. The work can focus on practical experimentation, on more theoretical analyses of the implementation strategy, or a mixture of both.

Level: B.Sc. project, M.Sc. project, or M.Sc. thesis

Prerequisites: General knowledge of multiprogramming and compilation; practical programming/implementation skills.

1.2 Natural-Language Presentation of Formal Proofs

Formal proofs based on natural deduction can be readily checked by machine, but also quite closely mirror precise mathematical argumentation, as done by humans. The main difference is that formal logic proofs are conceptually tree-like objects built out of inference-rule instances, while natural-language proofs are inherently linear presentations of the same basic steps, obtained by some suitable traversal of the proof tree – a process which should itself be mechanizable. The goal of generating a natural-language proof is not so much to verify that the proof is correct (a machine can do that much more reliably), but to convey the main insights behind a machine-formalized proof in the most effective way – keeping in mind that humans tend to have a very short mental stack.

Project:

Survey the existing literature on natural-language proof presentation (there isn't very much) and implement some of the suggested strategies, and/or develop new ones. Depending on the level and size, the project can focus either on covering a reasonable spectrum of formalized proofs, or on generating easy-to-read natural language.

Level: B.Sc. project, M.Sc. project, or M.Sc. thesis

Prerequisites: A course in logic, semantics, or type systems. A little background in natural-language processing might prove handy, but is not required.

1.3 Pretty-Printing of Formalized Programming-Language Definitions

The LF logical framework (as used in the Twelf proof assistant) is a domain-specific metalanguage for formalizing programming language syntax, semantics, type systems, and logics. It allows one to represent a collection of inference rules defining a semantic judgment (such as "expression e evaluates to value v" or "expression e has type tau") as a set of datatypes with suitable constructors, such that checking correctness of a derivation of the judgment amounts to type-checking a collection of terms in the metalanguage. Likewise, it is possible to automatically search for derivations of a judgment, e.g., to execute test programs using a formalized operational semantics.

The formal LF metalanguage, however, is based on a plain textual syntax, while most presentations of formal systems (in research papers or textbooks) use a variety of fonts, styles, and special symbols to represent grammars, inference rules, and derivations. Thus, even if the system is fully formalized, there is a significant gap between the formalized and the presentational versions, requiring both to be developed and maintained in parallel. It would be much preferable if the presentational form could be generated directly from the formalized one, in such a way that misprints and tedious hand-typesetting could be avoided entirely. To do this, the formalized system would be augmented with markup comments saying, for example, that "<if e1 e2 e3>" should be typeset as "if <e1> then <e2> else <e3>" (but with the keywords in bold), or that the evaluation judgment "<eval e v>" would be written as "<e> ==> <v>" (with a proper arrow symbol).

Project:

Develop a system for pretty-printing LF-formalized definitions and derivations to be included as figures in LaTeX documents. There are opportunities for focusing either on specific formatting issues and the details of the markup language, or for investigating on how some advanced features of LF, such as higher-other abstract syntax (a uniform way of dealing with variable-binding constructs and substitutions) can be reliably mapped to a traditional "paper" representation.

Level: M.Sc. project or M.Sc thesis.

Prerequisites: one (or preferably more) courses on logic, semantics, or type systems. Working knowledge of TeX/LaTeX, MathML, or a similar markup language, Some prior exposure to Twelf or other proof assistants would be ideal, but is not required.

1.4 Certified Code

Certified, or proof-carrying, code consists of an executable program (whether expressed in machine code or in some intermediate language such as JVM or .NET bytecodes), together with an independently verifiable mathematical/logical proof that the code satisfies some additional property,. Typically this property dies not specify full functional correctness, but only, e.g, that the code is sufficiently well-behaved that it may be executed in a trusted environment without runtime sandboxing. Unlike the superficially related notion of cryptographic "code signing", a formal certificate does not rely on trust in a person or organization, but ultimately only in the laws of logic themselves.

The certificates can either be generated completely automatically by the compiler (which can insert additional runtime checks if it cannot guarantee that the original code is correct/safe without them), or based on additional hints from the programmer (such as loop invariants) that codify a formal reason for why the code is correct as written. In addition to proving a certificate of correctness to the end user, formalized proofs of specification satisfaction are also useful in the development process, to keep track of internal interfaces and correct implementations of modules according to these interfaces, as seen e.g., in the Spec# framework for C# or JML for Java.

Project:

A large number of projects are possible, depending on both the nature of the executable code and the properties to be certified. See `http://www.diku.dk/~andrzej/certcomp.html` for more details.

Prerequisites: One or more courses on logic, semantics, or type systems.

Level: B.Sc. project, M.Sc. project, or M.Sc. thesis

2 Robert Glück

2.1 Program Inversion and Reversible Computation

Many problems in computation can be specified in terms of computing the inverse of an easily constructed function. Program inversion is one of the fundamental operations on programs. A familiar example of two programs that are inverse to each other is the encoding and decoding of data. The proposed projects explore different manifestations of program inversion, reversible logic circuits and reversible programming languages, and their connection to quantum computing.

Prerequisites: Good knowledge of programming languages and/or compiler construction, and having fun constructing experimental software and trying new ideas. Ideally, the course Program Inversion and Reversible Computation.

3 Fritz Henglein

See "Pan-TOPPS projects".

4 Jyrki Katajainen

Contact supervisor directly.

5 Ken Friis Larsen

5.1 New Backend for Moscow ML

Design, implement and evaluate a new backend for Moscow ML. This could be a backend targetting .NET CIL or JVM bytecode, for instance.

5.2 Code Duplication Discovery and Code Merging of NAV C/AL Code with git

Use the infrastructure from the version control system git (`http://git-scm.com/`) to construct code duplication discovery and code merging tools for the programming language C/AL. C/AL is the programming language used in the ERP system Microsoft Dynamics NAV.

5.3 Additional projects offered by Ken Friis Larsen

See "Pan-TOPPS projects below".

6 Julia Lawall

6.1 Finding Bugs in Open-Source Software

Project Description Software seems to be intrinsically riddled with bugs. In the case of open-source software, everyone has the chance to search for such bugs and submits bug fixes. Still, to find a bug, one has to have some idea of where to look and for what. In recent work, members of the TOPPS group have developed, in collaboration with researchers at the Ecole des Mines de Nantes, Coccinelle, a tool for searching and transforming C code. Coccinelle has been primarily applied to Linux code, both for updating code with respect to changes in the Linux internal APIs and for finding bugs. The goal of this project is to study the applicability of Coccinelle to transforming and finding bugs in other kinds of systems software, such as Gnome, BSD, or MacOS.

6.2 Debugging oopses

Find and fix the bugs that cause the oopses in the Linux kernel oopses database. I suspect this project will have a low chance of success, if that is defined as finding and fixing a lot of bugs. What is interesting is to understanding the process and what kinds of tools could be useful in this setting.

Master's project, 7.5 ECTS.

6.3 Implementing Semantics

When designing a programming language, it is a useful exercise to write down in a formal way the semantics of the language as well as the definition of any associated analyses. But as the language evolves, the written specification tends to fall behind the actual implementation. Furthermore, while the implementation benefits from any type checking provided by the implementation language and from subsequent testing, it is very easy to make mistakes in a written specification. The purpose of this project is to evaluate and try to improve on tools that have been designed to bring together the specification of language tools and their implementation. The semantics, type systems, and flow analyses of a number of domain-specific languages are available for use as test cases.

Prerequisites: DAT-V Programmeringssprog or the equivalent.

7 Torben Mogensen

Contact supervisor directly (also see "Pan-TOPPS-projects").

8 Jakob Grue Simonsen

8.1 Automated Harvesting of Music Corpora

Traditional analysis of sheet music consists of one or more persons analysing rhythm, chord sequences and other characteristics of a single piece, set in the context of an often vague comparison of other pieces by the same composer or other composers from the same period.

Traditional automated analysis of music has barely treated sheet music, but has focused on signal analysis and the use of machine learning techniques to extract and classify within, say, mood or genre.

In contrast, incipient research at DIKU aims to automate parts of the analysis of sheet music. The added value is the potential for extracting information from large volumes of sheet music that cannot easily be done by hand and cannot be meaningfully analysed by machine learning techniques.

The current project aims at extending prototype tools developed at DIKU for mass harvesting and large-scale analysis of sheet music, with a particular focus on (i) providing analysis of chord sequences, (ii) ensuring high data quality.

The particular analysis to be performed will the verification/falsification of a number of hypotheses related to the entropy of chord selection. Informally: "How does the unpredictability of 'what the next chord will be' scale with the length of the piece?", "Does a composer's choice of chords become more unpredictable with age?", "Is there a quantitative difference in the predictability of chord choice in 'great' composers compared to their contemporaries?".

Particular tasks:

1. Implement a harvester (pre-existing DIKU-developed prototype code exists for this purpose) for automated downloading of and meta-data-generation for music scores from online repositories.
2. Modify existing DIKU-developed scripting tools for batch-processing sheet music conversion to MusicXML using commercial applications.

3. Devise and implement a method for converting sheet music to MusicXML with minimum error using an array of existing commercial applications (colloquially: Let several programs vote to obtain the correct answer).
4. Perform analysis of the entropy of chord sequences for a large dataset using chords extracted from MusicXML.

Hardware and commercial software will be supplied by the supervisor. Prototype software for much of the above exists already and will be provided.

Level: Bachelor's thesis or master's project.

Workload: Approx. 15 ECTS (may be extended to 30 ECTS).

Prerequisites: Practical programming skills. Ability to read music a distinct advantage. Basic knowledge of statistics an advantage.

Notes: Highly practical, ambitious project. Will require both significant skills in independent reasoning and contact with the supervisor several times per week.

8.2 Parameter Extraction from MusicXML

MusicXML is one of the standard formats for representing music in a hierarchical form that also allows for storage of various kinds of metadata.

Unfortunately, the MusicXML format suffers from a number of infelicities that render the extraction of information from raw MusicXML (without specific annotations) difficult. Hence, MusicXML has serious drawbacks as a data format for large-scale statistical analysis of music.

The aim of this project is to devise simple algorithms for deriving a number of parameters from MusicXML representing tonal music with the express purpose of using them for statistical analysis of corpora of sheet music.

A brief shortlist of parameters to be extracted (more will be added as the project progresses):

1. Chromatic scale degree
2. Chord progression
3. Texture (number of voices active)
4. (Un)directed melodic intervals
5. Contour (shape of melodic interval progression over time)

Workload: 15–30 ECTS

Level: Bachelor's thesis, master's project or master's thesis.

Notes: Practically-oriented project. Ability to read music an advantage.

8.3 Automated Frame-Based Paraphrasing of Natural Language

Ostensibly, the most common source of academic plagiarism is the direct syntactic copying of source material with man-made paraphrasing.

The present project will serve as a proof-of-concept that, using natural-language semantic parsing, it is possible to perform an automated paraphrasing that, apart from domain-specific terms, will fool several of the standard tools for plagiarism detection.

The tool of choice will be semantic analysis using semantic frames. A semantic frame is, loosely, an abstraction characterising some (natural language) semantic meaning such as "applying heat to an object" and an associated list of socalled "frame elements" representing the semantically important syntactic constituents of the frame. Thus, "Puff incinerated the man" is an instantiation of the frame "applying heat to an object" and the associated frame elements are "actor", "verb" and "object", instantiated as "Puff", "incinerate", and "the man".

The basic task in the project will be to associate each semantic frame with a number of different possible instantiation and perform an automated replacement of some or all frame instantiations in a text with other instantiations of the frames. Hence, the result will be semantically equivalent to the original input text, but with significantly different syntax.

Particular tasks:

1. Using an existing setup of a semantic parser using Framenet, harvest and catalogue a database of possible frame instantiations for the most common frames. Goal: For each possible frame, there is a set of "candidate sentences" with which one frame instantiation can be substituted with another.

   A subtask will be to devise a suitable database format for storing frame instantiations.

2. Implement a tool that:

   - Takes as input a plaintext document in English
   - Semantically parses said document and employs some algorithm to replace frame instantiations by instantiations found in the database described above (with suitable word-for-word substitutions).
   - Returns the document with substituted frame instantiations.

3. Assess and adapt the algorithm from 2 by

   (a) Compile a small database of Engligh-language texts from a variety of domains to use as testbed.

   (b) using state-of-the art, third-party plagiarism detection tools to detect whether the output is deemed to a plagiarised version of the input.

   (c) performing a qualitative estimate of the semantic (and literary) quality of the output ("Does it convey the same meaning as the original?", "Is t he prose readable?").

Level: Bachelor's thesis or master's project.

Workload: Approx. 15 ECTS (may be extended to 30 ECTS).

Prerequisites: Practical programming skills, at least one compiler course.

Notes: Highly practical, ambitious project. Will require both significant skills in independent reasoning and contact with the supervisor several times per week.

8.4 Empirical Equidistribution of Algebraic Numbers

A description of the previous two student projects concerning this area is available at:

`http://www.diku.dk/~simonsen/bach/pisot/`.

The current project offering concerns stress-testing the Gupta-Mittal root-finding algorithm and computing approx. 10^8 digits of a large number of algebraic numbers from different classes as well as compiling new statistics. The experimental results will, if succesful, very likely be publishable.

Level: Bachelor's thesis of master's project.

Workload: Approx. 15 ECTS.

Notes: Good mix of theory and practice with an emphasis on practics. The supervisor is *very* interested in getting the work done. The project is, unlike most others, fairly suited for solo work.

8.5 Resource-Bounded Incompleteness

It is well-known that the Gödel incompleteness theorems may be proven using Kolmogorov complexity. This fact has recently been strengthened by Calude and Jürgensen ('Is Complexity a Source of Incompleteness', Adv. App. Math. 35, 2005). However, no *quantitative* results exist concerning the relationship of incompleteness in formal logic and resource-bounded Kolmogorov complexity.

The project will consist of characterising the sets of results that cannot be proven to be true using proof systems with limited resources. For instance, given some time-constructible function $f : \mathbb{N} \to \mathbb{N}$, a relevant result is a characterization of the sets of theorems that may be proven true in time $f(n)$ where n is the length of the theorem.

Kolmogorov complexity will be used as a vital tool in establishing said characterizations.

Level: Master's project or master's thesis.

Workload: 15–30 ECTS.

Notes: Highly theoretical project with very little practical content.

8.6 Compression, Omega-Numbers and the Choice of Universal Machines

It is well-known that any finite bit sequence x may be described fully by the minimal length $C(x)$ of some other bit sequence from which a particular universal Turing machine will output x (intuitively, $C(x)$ is the smallest length that x can be compressed to using any program). Furthermore, it is well-known that, up to an additive constant (independent of x), the choice of universal machine does not affect $C(x)$. In contrast, the *density* of compressible bit sequences may apparently depend on the choice of universal *programming language* (density, here, informally means "proportion of the possible sequences of a given length").

The famous real number Ω (see e.g. the Wikipedia) representing the halting probability of Turing machines is *uncomputable* in the sense that no program computes the sequence of digits of Ω in the obvious sense. However, if sufficiently many short bit sequences do not represent well-formed programs (equivalently, the density of *compressed* bit sequences is low), a significant number of digits of Ω may be computed. It is *highly* likely that this fact is an artifact of *Turing machines* that does not carry over to other paradigms of computation.

This project aims at investigate paradigms of computation that have a *large* density of correct programs, compare these paradigms with Turing machines and establish *how few* digits of Ω may be computed for them.

Level: Master's project, possibly master's thesis.

Workload: 15–30 ECTS.

Notes: Highly theoretical project. Toy code can be provided for illustrative purposes and simple computations.

8.7 Productivity of Stream Definitions

A (first-order, untyped) functional program outputting a stream of data is said to be /productive/ if the output stream is infinite ("there is always a next element that will be output"). Equivalently, we may query some functional program for "the next element" in the stream and get an answer in finite time.

For instance, the following program (written as a term rewriting system) is productive and computes the celebrated Thue-Morse sequence:

$$\begin{array}{rcl} i(0) & \to & 1 \\ i(1) & \to & 0 \\ zip(x:xs,ys) & \to & x:zip(ys,xs) \\ inv(x:xs) & \to & i(x):inv(xs) \\ even(x:xs) & \to & x:odd(xs) \\ odd(x:xs) & \to & even(xs) \\ tail(x:xs) & \to & xs \\ M & \to & 0:zip(inv(even(M)),tail(M)) \end{array}$$

And hence

$$M \to^* 0:1:1:0:1:0:0:...$$

It is undecidable whether a given stream definition (program or term rewriting system) is productive, in fact unpublished research by the supervisor shows the problem to be complete for the class Π^0_2, i.e. loosely as hard as showing that a given Turing machine halts on all inputs.

A fair amount of research has been done to give approximate, safe characterizations of when a stream definition is productive, most recently by Endrullis et al. However, these methods all use fairly involved, novel techniques, e.g. pebble flows.

We conjecture that it is possible to massage well-known methods for proving programs terminating to obtain automatic means of proving stream definitions productive.

It is easy to see that if the stream definition contains no projections (rules on the form l -¿ x for some term l), any safe algorithm for proving termination can be immediately converted to an algorithm proving productivity by redefining the set of constructors of the system.

Unfortunately, projections are everywhere in real programming (e.g. $head(x,y) \to x$). The major technical challenge is to account for these.

Aim:

1. Give an account of stream definitions and productivity
2. Give an account of (select) existing methods for proving termination of first-order functional programs and term rewriting systems.
3. Establish a technical framework for converting a given termination method to a method for proving productivity. Prove correctness of the framework.
4. Extend the framework suitably to encompass larger classes of definitions. Compare with existing methods of proving productivity.

Level: Master's thesis

Workload: 30 ECTS.

8.8 Projects Offered in Cooperation with the Royal Library of Denmark

See `http://www.diku.dk/~simonsen/bach/kb07/projfold.pdf`

At present, only the following projects are available:

1. "Standardisering af tre digitale arkiver på fælles objektmodel"
2. " LOCKSS: Peer to peer deling af dokumenter"

8.9 Projects Offered in Cooperation with the Department of Defense

See:

- `http://www.diku.dk/~simonsen/projects/CEP.pdf`
- `http://www.diku.dk/~simonsen/projects/StatiskAnalyse.pdf`
- `http://www.diku.dk/~simonsen/projects/fuzzer.pdf`

All students must sign a non-disclosure agreement with the Department of Defense.

9 Pan-TOPPS Projects

9.1 Projects in 3rd Generation Enterprise Resource Planning

See www.3gerp.org