Optimising data structures with multi-stage programming

An MPhil project proposal

E. S. R. Range (*esrr2*), St Edmund's College

Project Supervisor: Jeremy Yallop

1 Introduction, approach and outcomes (489 words)

Generalisation and abstraction are two invaluable attributes in software engineering, allowing for the construction of elegant solutions when tackling complex problems. Many such problems will require storage and manipulation of data in data structures, with great effort devoted to ensuring efficient handling of such data. Herein, however, lies an inherent conflict; while efficient data structures, such as tries [**?**] or red-black [**?**] trees, can offer improved performance, much of the work in ensuring high performance involves careful control of memory layout and accesses, in order to maximise spatial and temporal locality and exploit caching provisions. This conflicts directly with the principles of abstraction, where the underlying complexity is abstracted away. For example, features such as algebraic data types (ADTs), originally staples of functional programming languages where abstraction and generalisation are central tenets, are now being increasingly introduced into mainstream programming languages. Data structures can be elegantly defined in terms of them, however no guarantees are made about the underlying data layout.

Previous work, such as that by Hinze [**?**] and Elliott [**?**] has demonstrated great success in applying these generalisation and abstraction principles to data structures. [**?**], for example, allows for the extension of tries, initially utilised [**?**] as a structure to represent string-indexed data, to be used as maps indexed by arbitrary datatypes. This does, however, run up against the aforementioned conflict; the structures produced by this generalised approach often result in fragmented memory layouts, producing data structures far less space and computationally efficient than hand-crafted equivalents.

A possible solution to this conflict is the application of multi-stage programming [**?**]. In multi-stage programming, compilation occurs in numerous distinct stages, allowing for the programmatic generation of hand-crafted code in a type-safe manner. Baudon et al. [**?**] demonstrate that abstractly-defined data structures can have their underlying data representations safely transformed, permitting optimisations of the packing of data. Multi-stage programming can facilitate such optimisations, by enabling the translation of a high-level interface, such as those provided by ADT-defined data structures, into lower-level representations, such as sequential nodes of a tree packed into contiguous memory, aiding cache efficiency. Staging allows this translation to occur without direct unboxing and manipulation of the high-level types, which although supported by many languages [**?**], discards any guarantees of safety, or of compatibility with other language versions or hardware systems. Control over exactly which optimisations are applied could either be given directly to the user, through some high-level interface, or could additionally be abstracted away entirely, decided through an analysis or micro-benchmarking of the target hardware.

The primary outcome of this project will be to demonstrate improved performance characteristics of common data structures, through a staging-based manipulation of their underlying representations,

invisible to clients of the data structure. To achieve and evaluate this, an analysis of the performance characteristics of existing generalised data structures will be required, necessitating the creation of representative example programs and testing and benchmarking infrastructure. From this analysis, specific causes of performance degradation can be identified and targeted with staging-based transformations.

2 Workplan (498 words)

Over the course of the 26 weeks, there are three main sections of work which must be completed. First of all is the characterisation and analysis of the performance of existing generalised data structures. This will require the aforementioned creation of evaluation infrastructure. With this baseline measurement complete, the second stage will involve targeted optimisation of the determined performance impediments with staging techniques. Finally, the last stage will entail evaluation of the success of optimisation techniques, determination of potential avenues for further optimisation, and production of the final software and report artefacts. This work has been split into the following two-week segments:

Date Range	**Objectives**
4 Dec - 17 Dec	Survey the existing landscape of generalised data structure implementations. Collect a representative sample of programs making use of these data structures. Explore relevant implementation details of potential host languages (Haskell, OCaml) and evaluate support for multi-stage programming and run-time code generation in each.
18 Dec - 31 Dec	Select a provisional implementation language. Investigate applicable staging techniques and familiarise self with staging tools for the target language. Construct benchmarking framework and evaluate performance of previously collected programs.
1 Jan - 14 Jan	Analysis and investigation of observed performance characteristics. Inspect intermediate compilation stages, core dumps, memory layouts. Determine, with reference to target language implementation, explanations for performance degradation and potential mitigations.
15 Jan - 28 Jan	Explore representations for data structure memory layouts and representation of potential optimisations within them. Connect previously identified mitigations to optimisations and design potential staging implementations of them.
29 Jan - 11 Feb	Implementation of staged optimisations to generalised data structures. Extend testing and benchmarking infrastructure to allow for comparison of implemented optimisations.
12 Feb - 25 Feb	Continue cyclical process of applying optimisations, evaluating results, and updating optimisations. Plan a more formal evaluation methodology to compare effects of optimisations. Refine representation of memory layouts if necessary.

26 Feb - 10 Mar	Preparation for project review (8th - 15th March). Begin initial draft of dissertation structure. Prepare rapid summary of achievements for one-minute madness (15th March).
11 Mar - 24 Mar	Evaluate possible extensions, particularly for more complex data structures (e.g. BTrees [**?**]). Consider potential improvements to memory layout representation to simplify optimisation representation. Explore possible formalisation.
25 Mar - 7 Apr	Implementation of extension ideas. Contingency weeks for any optimisation or benchmarking harness work left incomplete.
8 Apr - 21 Apr	Execute previously planned evaluation methodology and compile results. Generate resources for evaluation section of dissertation (graphs, tables). Plan narrative for evaluation section.
22 Apr - 5 May	Start writing full version of dissertation.
6 May - 19 May	Continue working on dissertation writing. Produce initial draft of full dissertation and send for feedback from relevant parties. Finalisation of project title after draft completion (20th May).
20 May - 2 Jun	Begin finalisation of software artefact for release. Incorporate feedback from consulted parties. Finalise any structural adjustments and conduct proof-reading for accuracy and clarity.
3 Jun - 13 Jun	Submission deadline for overall project (3rd June). Final preparation of software artefact for release. Creation of presentation and materials.