Trip Report — PLI 2001
ART Memo 23

Malcolm Wallace Olaf Chitil

September 12, 2001

1 Haskell Workshop 2001

This year's Haskell Workshop was once again the best attended of the PLI satellite workshops, with around 65 people. There were some interesting pearls and small applications, although many of the presentations could have usefully used more time to give a fuller flavour of the ideas – most felt a little rushed. No application letters were accepted.

1.1 30 minute talks

Purely Functional User Interfaces A novel approach to building GUIs, based on FRP, with a direct correspondence between GUI programs and arrow diagrams. (Arrows in the sense of Hughes' generalisation over computational models.) Reminds a bit of Fudgets. The authors do not intend to solve the Haskell GUI problem, that is, produce a complete and maintained GUI library (so yet another GUI).

Named Instances The author speculated about perhaps implementing his proposal for named instances in nhc98 rather than ghc – the prototype implementation is an extension of Mark Jones' Typing Haskell in Haskell.

A Functional Notation for Functional Dependencies The authors seem to view this notation only as an intermediate step towards Haskell with full dependent types and arbitrary computations on the type level (Cayenne?).

Rules in Ghc The RULES pragma, implemented in ghc, is a very cool and powerful way of enabling user-defined code improvements. It can completely subsume the old foldr/build deforestation stuff, although there are a number of tricky issues in practice, mainly related to interactions with other program-improvement techniques such as inlining. In a sense it is a method for achieving program-specific compiler extensions for optimality. Recommended mainly for library writers.

1.2 10 minute talks

Simon Marlow: Haskell Libraries, The Next Generation Presentation of hierarchical structure. Need more libraries and maintainers for them. Would like to have HaskellDoc and some standard Testing method.

Mark Shields: Lightweight Modules for Haskell Just said that he is working on a new module system and would like every interested person to join.

Wolfram Kahl: Animating Haskell by Term Graph Rewriting HOPS is a system for term graph rewriting that can also show the rewriting steps in a graphical animation. Similar to GHood but all intermediate redexes are shown. Translation of Haskell programs into HOPS by hand. Advertised especially for understanding space leaks. (Does HOPS have same semantics as Haskell? Could probably animate Adam's machines.)

Martin Sulzmann: TIE: A CHR-based Type Inference Engine He reformulates context simplification of Haskell classes in the constraint handling rule formalism. This is supposed to give a flexible framework for all kinds of extensions and variants of class systems. He just mentioned the implementation in the talk. I didn't quite understand the advantages.

Manuel Chakravarty: A standard Foreign Function Interface for Haskell98 Basically finished (general part and C, not yet for Java). Solicits last comments from the general community.

1.3 Discussion

The Haskell'98 Report continues to be revised in small ways for correctness and readability. Simon PJ will continue this until a whole month passes with no further changes, then issue a finalised new version.

Rather than starting work on a successor to Haskell, people generally want to see 'blessed addendums' to the Report, for common agreed extensions, e.g. FFI, MPTC, exceptions, concurrency, fundeps, etc. The FFI addendum is almost ready, but any others need volunteers, not to mention agreement between designers and implementations. The idea is that a solid reference definition/documentation of each extension should exist independent of any particular compiler (but there is a lack of volunteers).

It was noted that the future job of designing Haskell-2 will be made much easier by many small steps rather than one large effort.

Many people made a plea that the single biggest hindrance to the further spread of Haskell is the lack of a standard cross-platform GUI.

2 ICFP 2001

ICFP was well-attended again this year, although not quite as large as in 2000. Submissions were well down – the average was about 80 for many years, year 2000 peaked at 110, but this year there were only about 63. However, the quality was undiminished in the opinion of the program chair (Xavier Leroy). There were no real outstanding papers, although many good solid incremental improvements. A couple of novel application papers described a theatre stage lighting system, and recognition/charting of price movements in the stock exchange. There were three invited talks:

Syntax with Static Binders in Functional Programming Many people later remarked that they don't see handling binders and α-equivalence in a compiler as a problem. The new type system ensures that bound variables can only be used without breaking α-equivalence. The type system's approximations seem to be rather weak, forcing a rather sequential programming style.

BlueSpec A hardware description language from Sandburst, designed by Arvind and Lennart. Based on Term Graph Rewriting Systems. Looks like Haskell but compiles down to very simple bit ops (structural Verilog) for building hardware, or C for cycle-accurate simulations. A program's term graph rewriting rules are non-deterministic, and the compiler tries to schedule as many rules as possible simultaneously whilst retaining the same semantics as a sequential implementation. Essentially, all the interesting computation happens at compile-time.

Fastest Fourier Transform in the West The problem with implementing an FFT to beat all other FFTs is that the efficiency is a dynamic property – different algorithms perform better on different input sizes, e.g. prime/non-prime, and different machine architectures. FFTW chooses the best algorithms in a 'planning' run, which selects code fragments and profiles them. The code fragments are generated by a small problem-specific compiler written in O'Caml, and scheduled at runtime by a small executor written in C. In the end, the FFTW implementation equals or beats all other FFT implementations on all known target processors. The main message of the talk was to avoid improving programs by hand, because that will be non-portable and sub-optimal – instead, write a clever system to optimise automatically.

Amongst the submitted papers, the following were of particular interest.

Contification Using Dominators Continuation style seems to be the standard intermediate representation for optimising ML compilers. This and other talks show that there are a lot of optimising ML compilers around (especially full program compilers).

Optimising Pattern Matching The O'Caml implementation of pattern-matching has been improved by calculating 'pattern contexts' at compile time. This is rather like a cut-down data-flow analysis that enables the elimination of some redundant cases and the re-ordering of others to take fewer comparisons. Unfortunately the reordering pass is quadratic, and the context-calculation is exponential, but maybe you are prepared to pay the extra cost at compile-time for better run-times.

There are so many papers about optimising pattern matching around; I wonder if this is new. I think GHC's core language avoids both replicated tests and replicated rhss, using disguised gotos.

Down with Emacs Lisp: Dynamic Scope Analysis An application of flow analysis to determine where an Emacs Lisp program actually uses dynamic scoping. Only the few occurrences where it does need to be transformed to obtain a lexically scoped Scheme program.

Type-Safe Hot Swapping of Running Modules Some applications want to swap new library code for old at runtime, possibly involving a change of types in the internal representation of the abstract library interface. One approach to solving this requires a type-safe coercion function to be supplied for values in both directions old→new and new→old. The problem is that many redundant coercions can build up. This new approach allows some coercions to cancel each other out, leaving at most one coercion per operation. [Possibly relevant to the ART trusting scheme where we build a wrapper around the original definition - higher-order trusted functions might introduce redundant wrap/unwrap operations.]

Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML The Scheme community complained that most of it is known stuff, new is only the type safety. Staged computation and partial evaluation seems to be an important topic.

Generic Unification via Two-Level Types and Parameterized Modules The unification algorithm is certainly not optimal. Both two-level types and records as modules are a bit cumbersome to use. However, the two-level types have some elegance.

A Simple Implementation Technique for Priority Search Queues Priority search queues are definitely interesting data structures. The formulation of Dijkstra's shortest path algorithm is beautiful, but the complexity is unclear. Efficient functional data structures / algorithms seems still to be a promising area.

Real-time FRP To make FRP suitable for real-time guarantees, it must be restricted, and essentially detached from Haskell as a host language. The talk claimed that, given a suitable base language, the FRP layer could be proven to guarantee no core dumps, no space leaks, no deadlock, and no missed deadlines. The question remains whether you lose the ability to write programs in a nice understandable style, since the specific restrictions seem quite limiting.

Events in Haskell, and How to Implement Them Implementation of Reppy's CML in Haskell. Simpler implementation, because some functions have a different meaning.

On Regions and Linear Types The idea is to develop a strict functional language that combines both. The paper presents a calculus with a type system that is unsuitable for type inference. Author's relate their paper to work on reasoning about pointers.

Functional Array Fusion Very efficient parallel arrays for Haskell, implemented essentially as a library with an optional syntactic sugar that is implemented as a pre-processor. The efficiency is gained mainly by use of Ghc's RULES. This is a nice example of the power of user-defined program-improvement, following on from the description of the RULES mechanism in the Haskell Workshop.

Automatic Generation of Staged Geometric Predicates Described a cool method of getting exact (infinite precision) arithmetic from error-prone floating point operations in four stages. At each stage, if the remaining error is small enough, you can stop and use the result. The primary application is to geometric modelling.

A New Notation for Arrows Arrows are a generalisation of most patterns of computation. The purpose of this paper is to suggest some nice syntactic sugar to make programs more readable. Essentially, the arrow programming style must be point-free and this can be clumsy and difficult to read, but with some sugar it is possible to use something like 'do' notation to bind names to data flows within the computation. (The clever bit is that different arrows require different implementations of binding and application, depending on whether the arrow is also a monad or not.) The syntax corresponds quite closely to an intuitive boxes-and-arcs diagrammatic description of the computation, but there is no proposed translation from graphical layout to syntax.

I missed motivation why the sugar is defined this way. I find the half explicit, half implicit threading of state not very attractive. Is there no better way?

Extensible Algebraic Datatypes with Defaults Presents method to extend recursively defined data structures with operations by both new variants and new operations without changing or duplicating existing code. According to Matthias Felleisen Findler and Flatt already presented this method (in an even more flexible form) at ICFP 98.

Possibilities and Limitations of Call-By-Need Space Improvement It is possible to define a weak notion of space-improvement, where one program has the same or better space usage than another. This leads to many algebraic laws. However a strong notion of improvement leads to a much smaller set of laws. The approach in this paper is to introduce lots of little 'gadgets', that make individual laws work i.e. annotations representing tweaks to the space semantics. When reasoning about programs it is possible to introduce gadgets and then eliminate them later on. The upshot is that the equational theory is rich and it works, but it can be hard work.

Strong improvement means that both terms use exactly the same amount of space. This is very restrictive, but only for strong improvement a context lemma and a fixed point property can be established. The 'gadgets' are used to make terms for which weak improvement shall be proved strongly improving.

3 Programming Contest

The task was to take a pseudo-HTML document and reduce its length by improving the tagging, without changing the final rendering of the document. Correctness was vital, length-reduction the main judging criterion, speed was a secondary issue involving only a fairly generous timeout.

There were 171 valid entries (plus 92 excluded as late or duplicates).

Languages:

O'Caml	36
C	28
C++	16
Haskell	13
Java	12
SML	10
etc.	

57 entries remained after correctness tests (50% of O'Caml entries, only 25% of Haskell entries).

Winner of lightning round: Joe Armstrong / Blue Avengers (Erlang).

4th place:	ElegantBlob	(C)
3rd place:	[b8]clan	(O'Caml)
2nd place:	Dylan Hackers	(Dylan)
1st place:	Haskell Carrots	(Haskell)

The winners recognised the problem as an instance of a parsing problem and studied appropriate literature for the first 1 1/2 days.