Enhanced System of Interaction Combinators

Daroc Alden
Advised by Dr. P. Hatcher

What is ESIC?

We wanted to demonstrate that existing functional programming languages are passing up on opportunities for important optimizations. To illustrate this, we built ESIC, a system that:

- Represents programs as graphs
- Shares small units of work
- Is proven to be β-optimal

But how does ESIC's performance compare to existing languages? Can ESIC's guaranteed whole-program optimizations make up for its unorthodox structure?

We compared C, Haskell, OCaml, Python, and ESIC on several benchmarks, and identified the circumstances under which ESIC outperforms other languages.

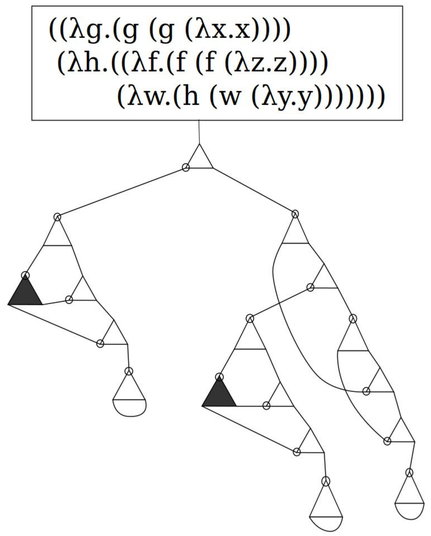
Compared With Other Languages

The most similar existing language to ESIC is Haskell - which also represents program as a graph. ESIC uses an undirected graph with a rigid structure, while Haskell uses a free-form directed graph of 'thunks'.

ESIC is also similar to other term- or graph-rewriting systems. The main feature that sets it apart is its use of 'sharing' and 'unsharing' nodes to precisely control how much is evaluated when invoking a function.

Here are some of the differences between ESIC and other languages:

	ESIC	Haskell	Strict Languages
Sharing Between Function Calls	All calls share	All calls share until an argument is evaluated	No calls share
Code Representation	Graph; Data and Code intermixed	Graph; Data and Code intermixed	Separate Code and Data
Constant Folding	Folds every function application	Programmer specified rewrite rules	Only folds specific operations
Parallelism	Automatic	Semi-Automatic	Usually Manual



Benchmarks

Benchmark	Language	Time (CPU seconds)
Mandlebrot 100x100	ESIC	4.01
	Python	0.129
	C	0.002
	Haskell	0.008
	OCaml	0.005
Fibbonacci (30)	ESIC (naïve algorithm)	8.6
	Python (naïve algorithm)	0.375
	Python	0.016
Function Fusion (2^32)	ESIC	0.000 (too fast to measure)
	Python	4 minutes 37.497 seconds
	Haskell	1 minute 53.79 seconds
	C (not quite equivalent)	8.16
Function Fusion (A(1024, 1024))	ESIC	0.000 (too fast to measure)
Binary String Manipulation (2^28)	Haskell	18.691
	Python	4 minutes 5.888 seconds
	ESIC	0.000 (too fast to measure)
Binary String Manipulation (A(1024, 1024))	ESIC	0.000 (too fast to measure)

Design Choices

Programming languages can take years to design and implement. We made some design tradeoffs in order to thoroughly explore our thesis in the time available:

- Syntax - ESIC has inflexible, spartan syntax very similar to the **λ-calculus** extended with a **let operator.**
- Reduced I/O - ESIC supports printing and reading numbers, but **no other IO**.
- Error Messages - ESIC has completely inscrutable error messages.

There are some other choices we made. We wrote a **full mark-and-sweep garbage** collector for ESIC, but were worried that it **couldn't compare to mature implementations**, so the benchmarks were done using a **faster version** that deliberately **leaked memory**.

ESIC is implemented as a **separate compiler and bytecode interpreter**. The compiler is written in Haskell, to take advantage of Haskell's parsing libraries. The **bytecode interpreter** is written in **C for speed.**

Summary of Results

We set out to prove that representing **programs** as **graphs** could speed up execution. This is **true**, for a **subset of programs.**

If programs:
- Have many calls to a function
- Do math with very large numbers
- Use their input data exactly once

Then ESIC's techniques can **speed them up** by **orders of magnitude**.

If programs:
- Use data multiple times
- Rely on lots of IO
- Rely on nested loops

Then ESIC can't speed them up, and they may be **better suited to a normal programming environment.**

Future Work

ESIC, and the techniques it demonstrates are probably better suited to be an **intermediate optimization step** in a larger system. This suggests several further things to explore:

- How to compile ESIC bytecode to an imperative IR
- How to compile other languages to ESIC

This opens up additional options for ESIC itself, such as exploring the speedup available from **JIT compilation**, or integrating ESIC into a more mature language with better tuned **garbage collection**.

Bibliography

Ian Mackie. 1995. *The geometry of interaction machine.* In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '95). ACM, New York, NY, USA, 198-208. http://dx.doi.org/10.1145/199448.199483
Yves Lafont. 1997. *Interaction Combinators.* Information and Computation, Volume 137, Issue 1, Pages 69-101, ISSN 0890-5401, https://doi.org/10.1006/inco.1997.2643.
Maia, Victor. 2018. *"The Symmetric Interaction Calculus."* Medium, Medium, 23 Aug. 2018, https://medium.com/@maiavictor/the-abstract-calculus-fe8c46bcf39c.
Kohei Honda, Olivier Laurent. 2010. *An exact correspondence between a typed pi-calculus and polarised proof-nets.* Theoretical Computer Science, Volume 411, Issues 22–24, Pages 2223-2238, ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2010.01.028.
Abubakar Hassan, Ian Mackie, Shinya Sato. 2009. *Compilation of Interaction Nets*, Electronic Notes in Theoretical Computer Science, Volume 253, Issue 4, Pages 73-90, ISSN 1571-0661, https://doi.org/10.1016/j.entcs.2009.10.018.
Beniamino Accattoli. 2015. *Proof nets and the call-by-value λ-calculus*, Theoretical Computer Science, Volume 606, Pages 2-24, ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2015.08.006.
Damiano Mazza. 2006. *A Denotational Semantics for the Symmetric Interaction Combinators.*
Damiano Mazza. 2009. *Observational Equivalence and Full Abstraction in the Symmetric Interaction Combinators*
Lamping, John. "*An Algorithm for Optimal Lambda Calculus Reduction.*" POPL '90 Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 1990, https://dl.acm.org/citation.cfm?doid=96709.96711.
Salikhmetov, Anton. "*An Impure Solution to the Problem of Matching Fans.*" ArXiv.org, 1 Apr. 2018, https://arxiv.org/abs/1710.07516.
Fernández, Maribel, and Ian Mackie. 1998. *Interaction Nets and Term Re-Writing Systems.*