MLton

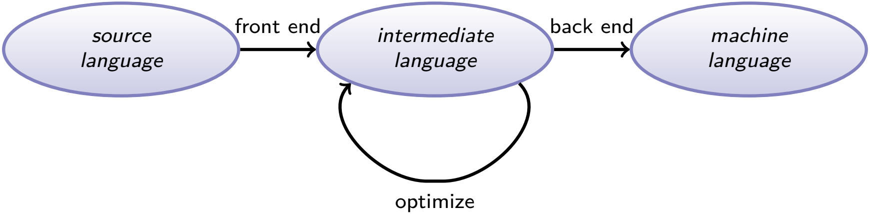
A Whole-Program Optimizing Compiler for Standard ML

Matthew Fluet
mtf@cs.rit.edu

Computer Science Community
October 6, 2014

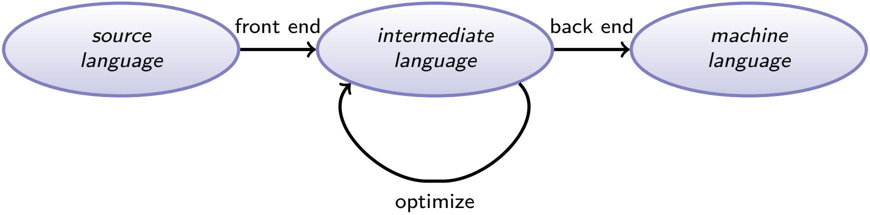
Compilers

Translate high-level program (source language)
into an equivalent executable program (machine language).



Compilers

Translate high-level program (source language)
into an equivalent executable program (machine language).



Compilers are themselves fascinating artifacts:

- theory meets practice
- large software system
- programming language design (programmer-side)
- programming language implementation (machine-side)

Language Design vs. Language Implementation

Fast programs are better than slow programs.
Short, understandable programs are better than long, confusing programs.

Advanced features make a language more attractive to programmers:

- simplify the development and maintenance of programs
- programs are shorter, easier to write and read, less likely to contain errors

Advanced features make a language more difficult for compiler writers:

- complicate the implementation of the programming language
- compilers supporting such features are often larger, more difficult to write and extend, and less likely to provide superior performance

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

Design vs. Implementation

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

- the Right tradeoff

Unfortunate if compiler writers expended great effort to implement advanced features only to have programmers avoid using such features.

Design vs. Implementation

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

- the Right tradeoff

Unfortunate if compiler writers expended great effort to implement advanced features only to have programmers avoid using such features.

> *If moving a function definition from one module to another will prevent the function from being inlined and incurs a cost of an extra procedure call at run time, then programmers are less likely to move the function.*

Design vs. Implementation

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

- the Right tradeoff

Unfortunate if compiler writers expended great effort to implement advanced features only to have programmers avoid using such features.

> *If using a generic type incurs a cost of an extra indirection or an extra word of space, then programmers are less likely to use a generic programming style.*

Design vs. Implementation

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

- the Right tradeoff

Unfortunate if compiler writers expended great effort to implement advanced features only to have programmers avoid using such features.

> *If using a higher-order function to abstract out an iterator incurs a cost of 20% in the program's running time, then programmers are less likely to use the abstraction.*

Design vs. Implementation

The lives of the programmers (the majority) are made easier,
while the lives of the compiler writers (the minority) are made harder.

- the Right tradeoff

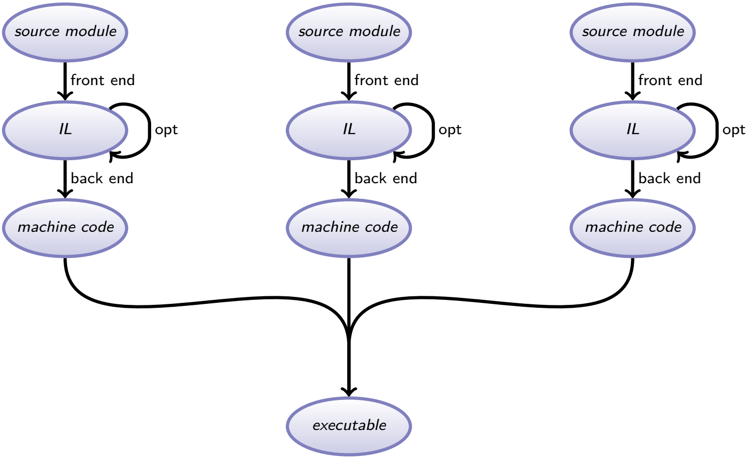
Unfortunate if compiler writers expended great effort to implement advanced features only to have programmers avoid using such features.

A compiler writer's implementation decision forces
the programmer to choose between performance and clarity.

Choosing performance (by not choosing to use advanced features) leads to programs that are longer, harder to write, read, and maintain, and more likely to contain errors.

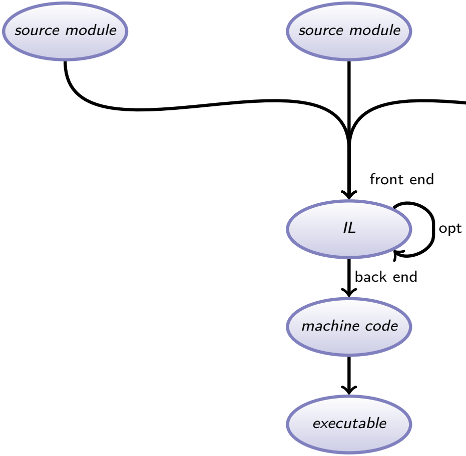
- the Wrong tradeoff.

Separate Compilation



Compiler has only partial information about the program being compiled.

Whole-Program Compilation



Compiler has maximal information about the program being compiled.
And simplifies the compiler itself (and makes experimentation easier).

MLton

A whole-program, optimizing compiler for Standard ML

www.mlton.org

MLton

A whole-program, optimizing compiler for Standard ML
`www.mlton.org`

- Developed since 1997
- Current release: 20130715
  - 145k lines SML for the compiler
  - 23k lines C for the runtime system (incl. garbage collector)
  - 37k lines SML for the Basis Library
- Feature-rich SML programming environment:
  - standalone executables, supporting large memory and large objects
  - complete implementation of Standard ML Basis Library
  - ML Basis system (MLBs) for programming in the very large
  - simple and fast C foreign function interface (FFI)
  - portable (C backend, x86 and amd64 backends, LLVM backend (next release))
  - development tools (scanners, parsers, profilers, interface generators)
  - extension libraries (continuations, signal handlers, object hashing, object size, threads, weak pointers, world save/restore)

MLton: Practical Programming in SML

Efficiency:

- raw speed
- eliminate performance disincentives of advanced features

Robustness:

- adherence to standards, completeness
- bugs and correctness are a priority
- support long runs and large inputs

Usability:

- good type error messages
- command-line interface, standalone executables
- large programs ($>$ 100k lines)
- short enough compile times ($<$ 5 minute self compile)

Commercial Users

Intel Labs

- Functional Language Research Compiler
- `dl.acm.org/citation.cfm?id=1900175`, `dl.acm.org/citation.cfm?id=2503779`

Reactive Systems

- a testing and validation tool supporting model-based design
- `www.reactive-systems.com`, `dl.acm.org/citation.cfm?id=1291172`

PolySpace (MathWorks)

- a code verifier to detect (or prove the absence of) run-time errors in src code
- `www.mathworks.com/products/polyspace`

SSH Communications Security

- a collection of internal-use software
- `dl.acm.org/citation.cfm?id=1362714`

Academic Users

Twelf (Carnegie Mellon University)

- an implementation of the LF logical framework
- `twelf.plparty.org/wiki/Main_Page`

Delta ML (Carnegie Mellon University)

- a language for self-adjusting computation
- `dl.acm.org/citation.cfm?id=1411249`

Skalpel: A Type Error Slicer for SML (Heriot-Watt University)

- a tool for explaining type errors in programs
- `www.macs.hw.ac.uk/ultra/skalpel/index.html`

Metis (Intel)

- an automatic theorem prover for first order logic with equality
- `www.gilith.com/software/metis`

OpenTheory (Intel)

- a tool for processing higher order logic theory packages
- `www.gilith.com/software/opentheory`

Academic Users (cont'd)

Exception Analysis (Aarhus University)

- flow-sensitive integer interval analysis to remove overflow checks
- `cs.au.dk/~alx/`

Popeye (Purdue University)

- infer expressive safety properties of functional programs
- `link.springer.com/chapter/10.1007/978-3-642-35873-9_19`

Catalyst (Purdue University)

- verify complex invariants over the shapes of algebraic datatypes
- `dl.acm.org/citation.cfm?id=2628159`

Spartacus (Saarland University)

- tableau prover for hybrid logic with global modalities
- `www.ps.uni-saarland.de/spartacus/index.html`, `www.sciencedirect.com/science/article/pii/S1571066110000320`

CAVA (Technische Universität München)

- computer-aided verification of automata
- `cava.in.tum.de/`, `link.springer.com/chapter/10.1007/978-3-642-39799-8_31`

Less Academic Users

SML3d (University of Chicago)

- graphics programming using SML and OpenGL
- `sml3d.cs.uchicago.edu/`, `www.youtube.com/watch?v=D8ltkS1AhAo`

HaMLet

- a reference implementation of SML
- `www.mpi-sws.org/~rossberg/hamlet/`

`http://mlton.org/Users`

Performance: SML Compilers

Compares all of the active SML compilers

- ML Kit, Moscow ML, MLton, Poly/ML, SML/NJ

with 40+ benchmarks ranging from 15 to 23k lines.

Run-time ratios over all benchmarks (out-of-date, from Sep. 2005):

	MLton	ML Kit	MoscowML	Poly/ML	SML/NJ
median	1.0	2.4	30.6	4.6	3.1
geo. mean	1.0	3.3	25.9	6.2	3.9

Microbenchmarks helpful to compiler writers,
but miss the point for users and whole-program optimization.

- Every compiler is "whole-program" if the program is 15 lines.

Performance: Large Programs

Large successes:

- Twelf (67k): "significantly faster" than SML/NJ
- HaMLet (22k): 3x faster than SML/NJ
- HOL (120k): 10.3x faster than Moscow ML
- ML Kit (120k): "significantly faster" than SML/NJ
- MLton (145k): 81x faster than SML/NJ
- RML (22k): 2x faster than SML/NJ
- SML.NET (80k): 3x faster than SML/NJ
- PolySpace (>100k): commercial, speedup not public

Large failures:

- HOL (400k)

What is Standard ML?

SML is a general-purpose functional programming language with

- strict evaluation
- strong and static typing
- polymorphic types
- type inference
- datatypes and pattern matching
- functional impurities (mutable objects, side-effects)
- exceptions
- a sophisticated module system
- a rigorous formal definition

Compiling SML Efficiently is Hard

Advanced features lead to missing information.

- higher-order functions ⇒ missing control-flow info
- polymorphism ⇒ missing type info
- functors ⇒ missing control-flow and type info

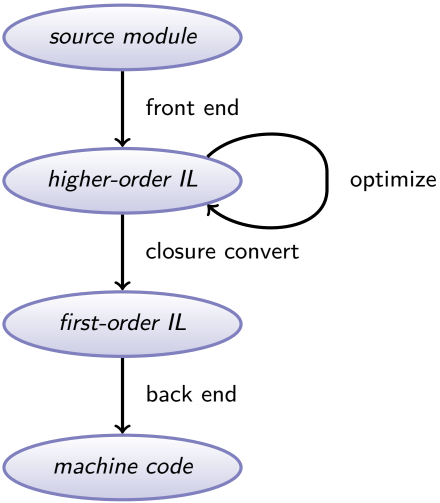
Missing information leads to bad code.

- inefficient data representations:
  - tagged integers, boxing, no packing, extra variant tags
- missed control-flow optimizations:
  - inlining, loop optimizations, dead-code elimination

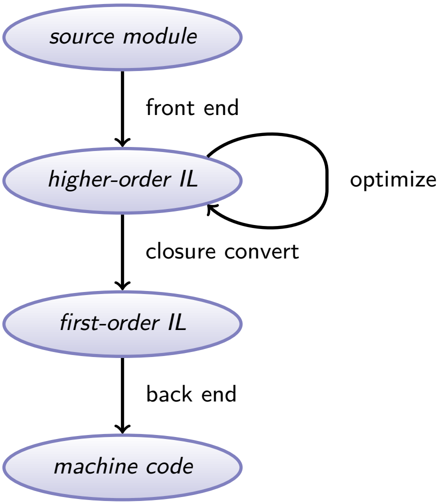
Compiler writers tie both hands behind their back.

- separate compilation
- complex, nonstandard intermediate languages for optimization

Traditional Approach to Compiling SML

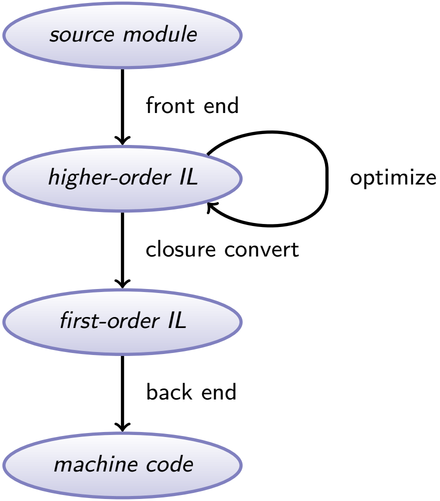


Traditional Approach to Compiling SML

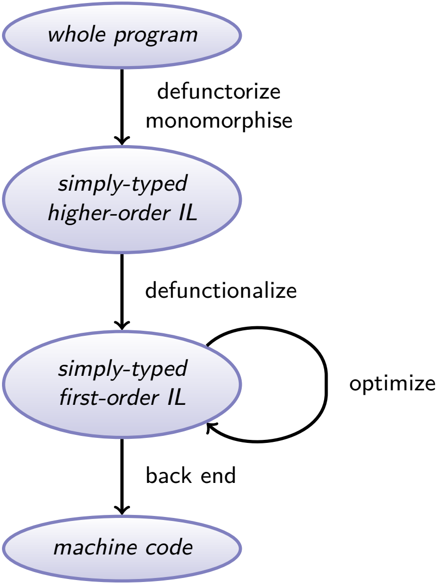


- Separate compilation
  - bad type info
  - bad control-flow info
- Polymorphic or untyped IL
  - bad data representations
  - bad dataflow analyses
- Higher-order IL
  - can't use traditional optimizations
  - optimizations do their own control-flow analysis
- Poor closure optimization

Traditional Approach

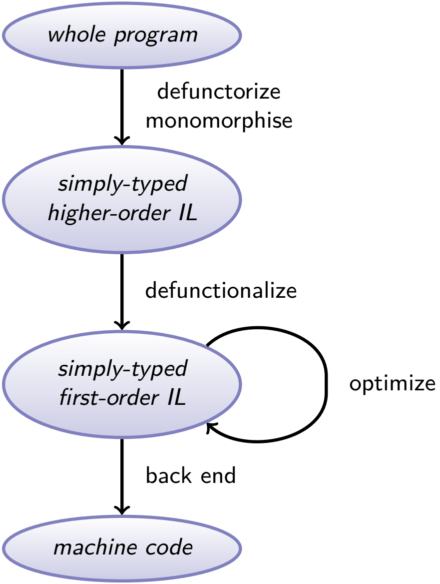


MLton's Approach



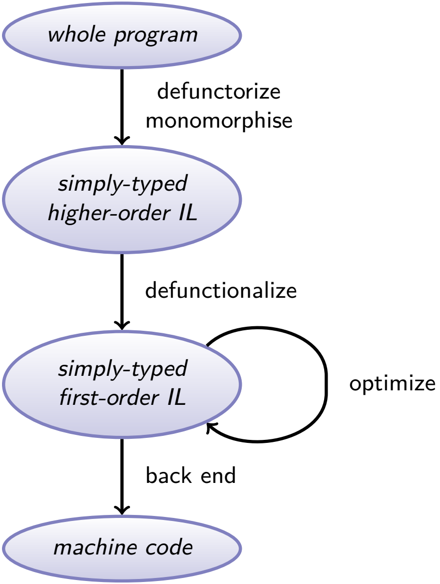
MLton's Approach: Benefits

- Absolute efficiency
  - massive optimization
  - good control-flow info
  - good data representations
- Relative efficiency
  - zero- or low-cost advanced features
- Simplicity
  - simple, typed IL
  - traditional optimizations
  - optimizations don't have to do their own CFA

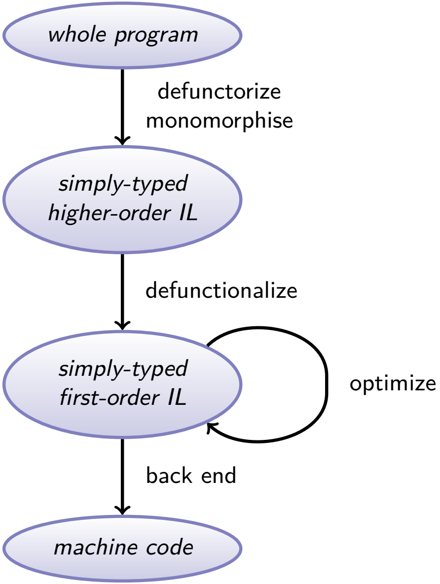


MLton's Approach: Drawbacks

- Compile time.
- Compile memory.
- Executable size.



MLton's Approach



Defunctorization

Goals:

- turn full SML into a polymorphic, higher-order IL
- expose types hidden by functors and signatures
- expose function calls across modules
- zero-cost modules for programmer

Method:

- eliminate structures and signatures
- duplicate each functor at every use

Code explosion in theory, but not in practice.

Defunctorization: Examples

All variables bound in **val** and **fun** declarations are renamed:

```
val x = 13
val y = x
fun f x = g x
and g y = f y
```

$\Rightarrow$

```
val x_0 = 13
val y_0 = x_0
fun f_0 x_1 = g_0 x_1
and g_0 y_1 = f_0 y_1
```

Type abbreviations are removed and expanded wherever they are used.

```
type 'a u = int * 'a
type 'b t = 'b u * real
fun f (x : bool t) = x
```

$\Rightarrow$

```
fun f_0 (x_0 : (int * bool) * real) = x_0
```

The type and value constructors in **datatype** declarations are renamed.

```
datatype t = A of int
           | B of real * t
```

$\Rightarrow$

```
datatype t_0 = A_0 of int
             | B_0 of real * t_0
```

Local declarations are moved to the top-level.
The environment keeps track of the variables in scope.

```
val x = 13
local val x = 14
in val y = x
end
val z = x
```

$\Rightarrow$

```
val x_0 = 13
val x_1 = 14
val y_0 = x_1
val z_0 = x_0
```

Defunctorization: Examples

Structure declarations are eliminated,
with all declarations moved to the top level.

Long identifiers are renamed.

```
structure S =
   struct
      type t = int
      val x : t = 13
   end
val y : S.t = S.x
```

$\Rightarrow$

```
val x_0 : int = 13
val y_0 : int = x_0
```

Open declarations are eliminated.

```
val x = 13
val y = 14
structure S =
   struct
     val x = 15
   end
open S
val z = x + y
```

$\Rightarrow$

```
val x_0 = 13
val y_0 = 14
val x_1 = 15
val z_0 = x_1 + y_0
```

Defunctorization: Examples

Functor declarations are eliminated,
and the body of a functor is duplicated wherever the functor is applied.

```
functor F(val x : int) =
   struct
     val y = x
   end
structure F1 = F(val x = 13)
structure F2 = F(val x = 14)
val z = F1.y + F2.y
```

$\Rightarrow$

```
val x_0 = 13
val y_0 = x_0
val x_1 = 14
val y_1 = x_1
val z_0 = y_0 + y_1
```

Defunctorization

Goals:

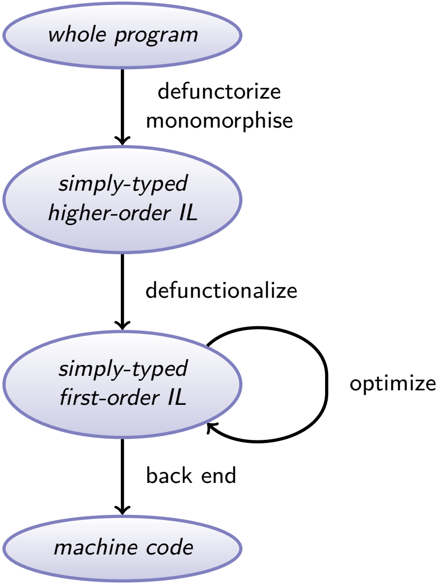
- turn full SML into a polymorphic, higher-order IL
- expose types hidden by functors and signatures
- expose function calls across modules
- **zero-cost modules for programmer**

Method:

- eliminate structures and signatures
- duplicate each functor at every use

Code explosion in theory, but not in practice.

MLton's Approach



Monomorphisation

Goals:

- eliminate polymorphism, producing a simply-typed IL
- enable good data representations
- zero-cost polymorphism for programmers

Method:

- duplicate type declarations at each type used
- duplicate function declarations at each type used
- rely on properties of SML for termination

Code explosion in theory, but manageable in practice.

- (Max increase seen is 30% in MLton)

Subleties: non-uniform datatypes, phantom types.

Monomorphisation: Example

Polymorphic program:

```
datatype 'a t = T of 'a
fun 'a f (x: 'a) = T x
val a = f 1
val b = f 2
val z = f (3, 4)
```

Monomorphic program:

```
datatype t1 = T1 of int
datatype t2 = T2 of int * int
fun f1 (x: t1) = T1 x
fun f2 (x: t2) = T2 x
val a = f1 1
val b = f1 2
val z = f2 (3, 4)
```

Monomorphisation

Goals:

- eliminate polymorphism, producing a simply-typed IL
- enable good data representations
- **zero-cost polymorphism for programmers**

Method:

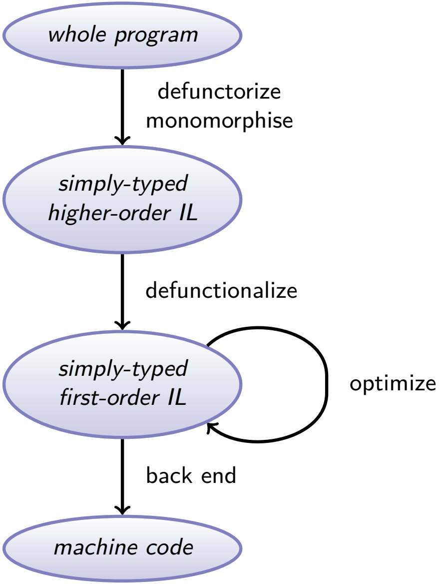
- duplicate type declarations at each type used
- duplicate function declarations at each type used
- rely on properties of SML for termination

Code explosion in theory, but manageable in practice.

- (Max increase seen is 30% in MLton)

Subleties: non-uniform datatypes, phantom types.

MLton's Approach



Defunctionalization

Goals:

- eliminate higher-order functions, producing a first-order IL
- make direct top-level calls, which are easy to optimize
- make control-flow info available to rest of optimizer
- optimize closures just like other data structures

Method:

- moves nested functions to top level
- function = tagged record of free variables
- call = dispatch on tag followed by top-level call
- control-flow analysis to minimize dispatches

Control-Flow Analysis (0CFA)

0CFA: whole-program dataflow analaysis

- computes set of functions at each call site

Imprecise in theory, but precise in practice.

- almost no calls require case dispatch

Cubic time in theory, but very fast in MLton.

- less than 2s to analyze MLton itself
- preprocessing based on types
- ignore first-order values
- hash cons sets and cache binary operations
- use union-find for equality constraints

Prior code duplication helps speed and precision.

Defunctionalization: Example

Higher-order program:

```
val f = fn a => fn b => a
val g = fn c => fn d => d
val h = if x < y then f else g
val m = h 13
val z = m 7
```

Flow analysis:

$$
\begin{aligned}
F(\mathtt{f}) &= \{\mathbf{fn}\ \mathtt{a}\} \\
F(\mathtt{g}) &= \{\mathbf{fn}\ \mathtt{c}\} \\
F(\mathtt{h}) &= \{\mathbf{fn}\ \mathtt{a}, \mathbf{fn}\ \mathtt{c}\} \\
F(\mathtt{m}) &= \{\mathbf{fn}\ \mathtt{b}, \mathbf{fn}\ \mathtt{d}\}
\end{aligned}
$$

Defunctionalization: Example

First-order program:

```
datatype t1 = C1 of unit (* fn a *)
datatype t2 = C2 of int  (* fn b *)
datatype t3 = C3 of unit (* fn c *)
datatype t4 = C4 of unit (* fn d *)
datatype t5 = C5 of unit (* fn a ∈ F(h) *)
            | C6 of unit (* fn c ∈ F(h) *)
datatype t6 = C7 of int  (* fn b ∈ F(m) *)
            | C8 of unit (* fn d ∈ F(m) *)
fun F0 (r, a) = C2 a      (* code for fn a *)
fun F2 (r as (a), b) = a (* code for fn b *)
fun G0 (r, c) = C4 ()     (* code for fn c *)
fun G2 (r, d) = C5 ()     (* code for fn d *)
val f = C1()
val g = C3()
val h = if x < y
          then (case f of C1 r => C5 r)
          else (case g of C3 r => C6 r)
val m = case h of
           C5 r => (case F0 (r, 13) of C2 r => C7 r)
         | C6 r => (case G0 (r, 13) of C4 r => C8 r)
val z = case m of
           C7 r => F1 (r, 7)
         | C8 r => G1 (r, 7)
```

Defunctionalization

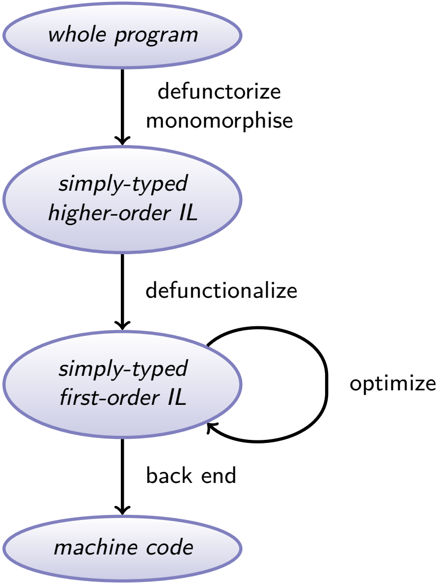
Goals:

- eliminate higher-order functions, producing a first-order IL
- make direct top-level calls, which are easy to optimize
- **make control-flow info available to rest of optimizer**
- **optimize closures just like other data structures**

Method:

- moves nested functions to top level
- function = tagged record of free variables
- call = dispatch on tag followed by top-level call
- control-flow analysis to minimize dispatches

MLton's Approach



SSA Intermediate Langauge

Traditional, simple IL.

- simply-typed, first-order
- program = datatypes + functions
- function = type + arguments + control-flow graph
- usual SSA conditions: def once, def dominates use

300 line interface, 2K line implementation

- immutable IL
- pretty printer, CFG visualizer, DFS, utilities

MLton options:

- `-drop-pass`, `-diag-pass`, `-keep-pass`

SSA Type Checker

Verifies:

- uniqueness of names
- variable definitions dominate uses
- control-flow graphs are well formed
- types at primitive applications and calls

Runs at beginning and end of optimizer.

Can be run after each pass (`-type-check true`).

- slows down optimizer by 50%
- catches bugs in optimizer

700 lines of code.

SSA Example: Nontail Fib

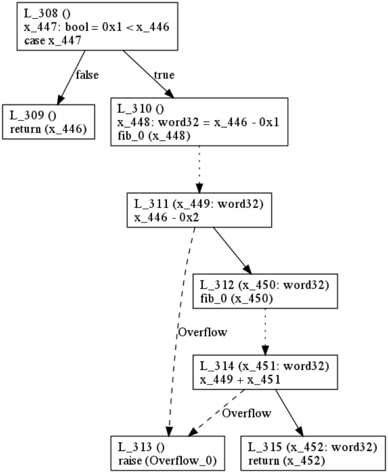
SML Source Function

```
fun fib n =
  if n <= 1
    then n
    else fib (n - 1) +
         fib (n - 2)
```

SSA Top-Level Function

```
fun fib_0 (x_446: word32) :
  {raises = Some (exn),
   returns = Some (word32)} =
  goto L_308 ()
```

SSA Control-Flow Graph



fib_0 control-flow graph

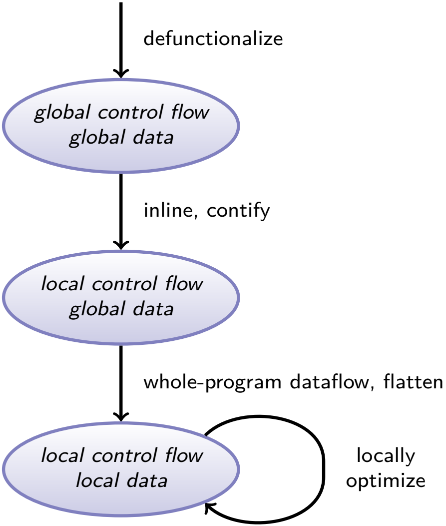
SSA Optimizer

Goals:

- turn function calls into control-flow graphs
- expose interprocedural data
- reduce tuple allocation
- traditional local optimizations

Method:

- 25 small, independent, SSA-to-SSA rewrite passes
- each pass:
  analyze, transform, shrink



SSA Shrinker

Goals:

- perform "obvious" local simplification
- let other optimizations focus on what they do best
- keep SSA IL programs small

Method:

- depth-first search of control-flow graph for each function
- reduce: primapps, case of variant, select of tuple, ...
- Appel-Jim shrinker applied to SSA.

One of the largest SSA pass, 1400 lines.

Inlining and Contification

Goals:

- turn function calls into control-flow graphs
- eliminate call overhead

Leaf inlining.

- uncurrying for free

Call-graph inlining.

- inline if: $(numCalls - 1) * (size - c) \leq limit$

Contification.

- turns functions used as continuations into jumps

Relies on 0CFA and first-order whole program.

Whole-Program Dataflow Optimizations

Goals:

- expose data to shrinker and later optimizations
- clean up across modules

Constant propagation.

- analyze: forwards from constants with a flat lattice
- transform: replace variables with constants

Useless-component removal.

- analyze: backwards from primitives, tests, FFI, ...
- transform: eliminate useless component

Flattening Optimizations

Goals:

- eliminate indirection (save space and time)
- pack tuples
- reduce allocation

Method:

- flatten function arguments and results
- flatten constructor applications
- flatten ref cells into data structures and stack frames
- flatten array components
- flatten basic-block arguments

Caveat: space safety

SSA Example: List Fold Becomes a Loop

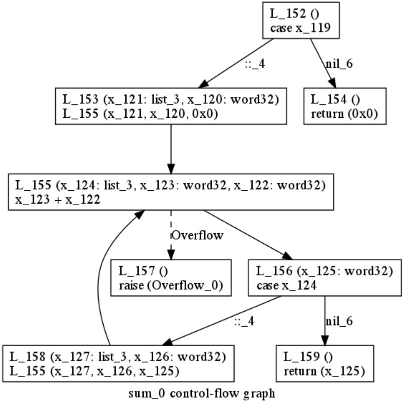
SML Source Functions

```
fun fold (l, b, f) = let
  fun loop (l, b) =
    case l of
      [] => b
    | x :: l => loop (l, f (x, b))
  in loop (l, b) end

fun sum l =
  fold (l, 0, fn (x, y) => x + y)
```

SSA Top-Level Function

```
list_3 = nil_6
       | ::_4 of (list_3, word32)

fun sum_0 (x_119: list_3) :
  {raises = Some (exn),
   returns = Some (word32)} =
  goto L_152 ()
```

SSA Control-Flow Graph



Dominator-based Local Optimizations

Goals:

- apply traditional intraprocedural optimizations
- take advantage of prior whole-program optimization

Method:

- compute dominator tree for each function's CFG
- recursively walk tree, use known facts in subtrees

Examples:

- common-subexpression elimination
- known-case elimination
- redundant-test elimination (includes bounds checks)
- overflow-detection elimination

Data Representations

Goals:

- choose efficient representation for each IL type
- save space and allocation
- make GC fast and easy

Method:

- pack tuples and array elements
- unbox datatype variants (including lists)
- reorder fields
- use untagged integers and words
- fast card marking

Simply-typed whole program is essential.

Technical Lessons

Whole-program compilation is feasible.

- compile a 100k line program in minutes with 1G RAM
- myths: defunctorization, monomorphisation, 0CFA

Whole-program compilation is effective.

- fast code and compact data representations
- total information $\Rightarrow$ optimizations rewrite at will

Whole-program compilation is simple.

- simplifies compiler
- simplifies optimizations
- simplifies intermediate languages

Technical Lessons

Simply-typed, first-order SSA is an excellent compiler IL, even for advanced languages.

- complete type information
- all passes benefit from CFA, which is only done once
- traditional optimizations

Structuring an optimizer as small, independent rewrite passes on an immutable IL makes life easy.

- easy to develop new passes
- easy to debug old passes
- easy to experiment with phase ordering
- easy for passes to help each other

Recent/Current Student Work

- LLVM Codegen (Brian Leibig; RIT)
- SIMD Support for MLton (Tucker DiNapoli; University of NH)
- Multi-entry functions (David Larsen; RIT)

Recent/Current Student Work

- LLVM Codegen (Brian Leibig; RIT)
  This project explored the design and implementation of an LLVM code generator, modeled after MLton's original C code generator. The LLVM code generator was found to offer a greatly simpler implementation compared to the existing C, x86, and amd64 code generators, along with comparable compile times and performance of generated executables.
- SIMD Support for MLton (Tucker DiNapoli; University of NH)
- Multi-entry functions (David Larsen; RIT)

Recent/Current Student Work

- LLVM Codegen (Brian Leibig; RIT)
- SIMD Support for MLton (Tucker DiNapoli; University of NH)
  This 2013 Google Summer of Code project explored the design of a core set of SIMD primitives, the implementation of the SIMD primitives in the C and amd64 code generators, and the application of the SIMD primitives in an SML library.
- Multi-entry functions (David Larsen; RIT)

Recent/Current Student Work

- LLVM Codegen (Brian Leibig; RIT)
- SIMD Support for MLton (Tucker DiNapoli; University of NH)
- Multi-entry functions (David Larsen; RIT)
  This thesis is exploring the design, implementation, and application of a new IL that supports "multi-entry" functions, whereby a function can be called at one of multiple entry points. Such "multi-entry" functions can be motivated by the control-flow graph representation of functions, by allowing multiple entry nodes (just as multiple exit nodes are often allowed). A "multi-entry" function can be used to perform call-pattern specialization and to improve the performance of mutually tail-recursive functions.

Current/Future Work

Positioning MLton for Next-Generation Programming Languages Research
`http://www.nsf.gov/awardsearch/showAward?AWD_ID=1405770`

- NSF Computing Research Infrastructure Award
- Joint with Lukasz Ziarek (University at Buffalo)
- Support for 2-3 students per semester per institution through July 2017
  - work on CRI funded infrastructure enhancements for a semester, then undertake a more research oriented activity (independent study, Honors project, or MS capstone project/thesis)
- type-checking and optimization infrastructures, threading and garbage collection frameworks, configuration and benchmarking support systems, and documentation

Future Work

New Optimizations:

- Improved closure representations
- Array bounds check elimination
- Overflow check elimination
- Loop-invariant code motion
- Partial redundancy elimination
- Loop unrolling
- Type splitting

Full employment theorem for compiler writers.

Future Work

New Optimizations Frameworks:

- Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation
  - `http://portal.acm.org/citation.cfm?id=1863539`
- Equality Saturation: A New Approach To Optimization
  - `http://portal.acm.org/citation.cfm?id=1480915`
- Supercompilation
  - `http://portal.acm.org/citation.cfm?id=1480916`
  - `http://portal.acm.org/citation.cfm?id=1863588`
  - `http://portal.acm.org/citation.cfm?id=1863540`

New Optimization Verifications:

- Safe-for-space

Future Work

New Analyses:

- Uncaught exception analysis
- Alternative control-flow analyses for defunctionalization

CFA research has had a renaissance;
an opportunity to test new "theory" in "practice".

Future Work

New Tools:

- Heap profiler
- Debugger
- Statistically sound benchmarking

Future Work

New Language Features (easy):

- Record punning, Record extension, Record update
- Or-patterns
- Vector literals
- Higher-order functors
- Nested signatures
- Local modules

Future Work

New Intermediate-Language Features (moderate):

- Multi-return function calls
- Heap-allocated stack frames

Future Work

New Runtime System Features (moderate to difficult):

- Garbage collector improvements
- Multicore

Future Work

New Language Features (hard):

- Polymorphic recursion
- First-class polymorphism
- Generalized abstract datatypes
- Higher-order polymorphism

These features cannot be *completely* monomorphised.

Future Work

New Language Features (hard):

- Polymorphic recursion
- First-class polymorphism
- Generalized abstract datatypes
- Higher-order polymorphism

These features cannot be *completely* monomorphised.

"Type-Flow Analysis for Partial Monomorphisation"

- monomorphise "as much as possible, and no more"

"Representation Monomorphisation"

- monomorphise with respect to back-end data representations

Future Work

???

Languages and Compilers provide lots of room to explore.

MLton Credits

Design:

- Henry Cejtin, Matthew Fluet, Suresh Jagannathan, Stephen Weeks

Implementation:

- Matthew Fluet, Stephen Weeks

Code:

- `gdtoa`, GnuMP, ML Kit, Moscow ML, SML/NJ

Support:

- NEC, InterTrust, PolySpace, Reactive Systems, NSF

People:

- Jesper Louis Andersen, Johnny Andersen, Christopher Cramer, Alain Deutsch, Martin Elsman, Brent Fulgham, Adam Goode, Simon Helsen, Joe Hurd, Vesa Karvonen, Richard Kelsey, Ville Laurikari, Geoffrey Mainland, Eric McCorkle, Tom Murphy, Michael Neumann, Barak Pearlmutter, Filip Pizlo, John Reppy, Sam Rushing, Jeffrey Mark Siskind, Wesley Terpstra, Luke Ziarek

Hosting:

- `github.com/MLton`, `sourceforge.net/projects/mlton/`

www.mlton.org