Whole-Program Compilation in MLton

Stephen Weeks
sweeks@sweeks.com

MLton

- MLton is an open-source, whole-program, optimizing Standard ML compiler.
- Developed since 1997.
- Code:
  - 145k lines SML for the compiler
  - 19k lines C for the runtime system
  - 35k lines SML for the basis library
- Platforms:
  - arch: x86, hppa, PowerPC, Sparc
  - OS: Linux, Cygwin, MinGW, Mac OS X, Solaris, *BSD
- Tools:
  - profiler, lexer/parser generator, FFI

MLton: Practical Programming in SML

- Efficiency:
  - raw speed
  - eliminate performance disincentives for 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)

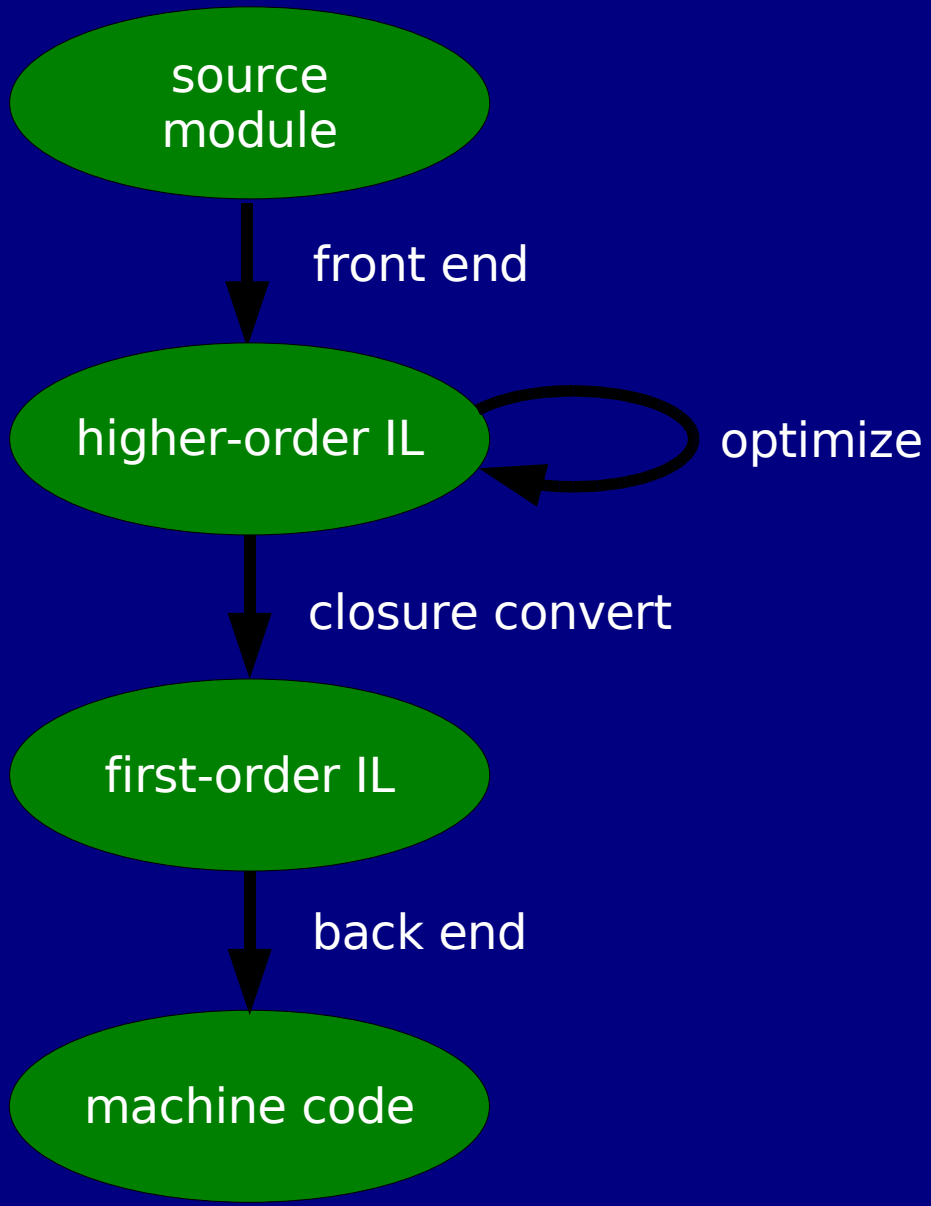
Compiling SML Efficiently is Hard

- 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

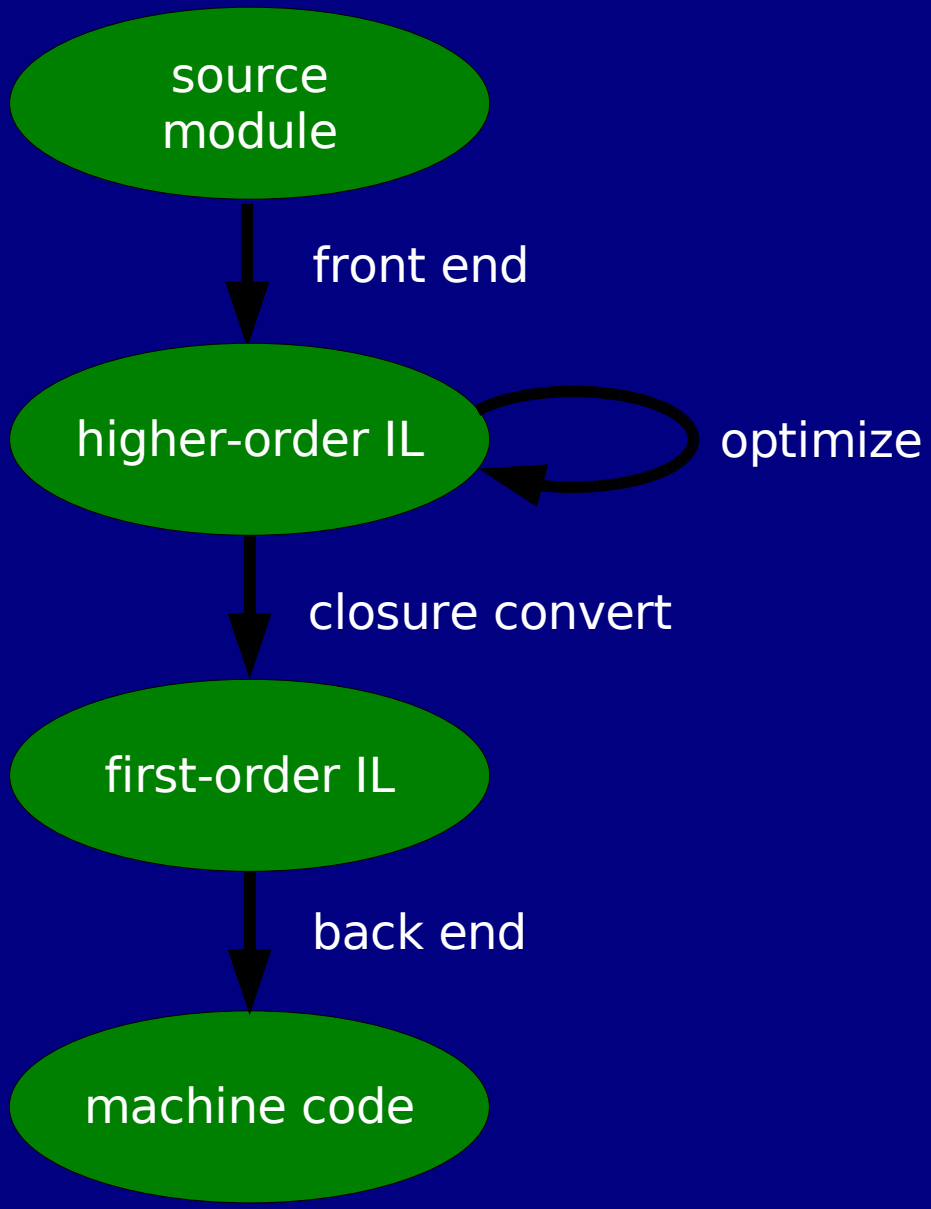
SML Implementations

1983 Poly/ML -- first
1986 SML/NJ -- most widely used
1989 ML Kit -- regions
1990 ML Works -- commercial
1994 Moscow ML -- byte-code compiler
1995 TIL -- typed intermediate languages
1996 MLj -- targeted JVM
1997 MLton -- whole-program optimization
1999 SML.NET -- targeted .NET
1999 HaMLet -- reference interpreter

Traditional Approach to Compiling SML

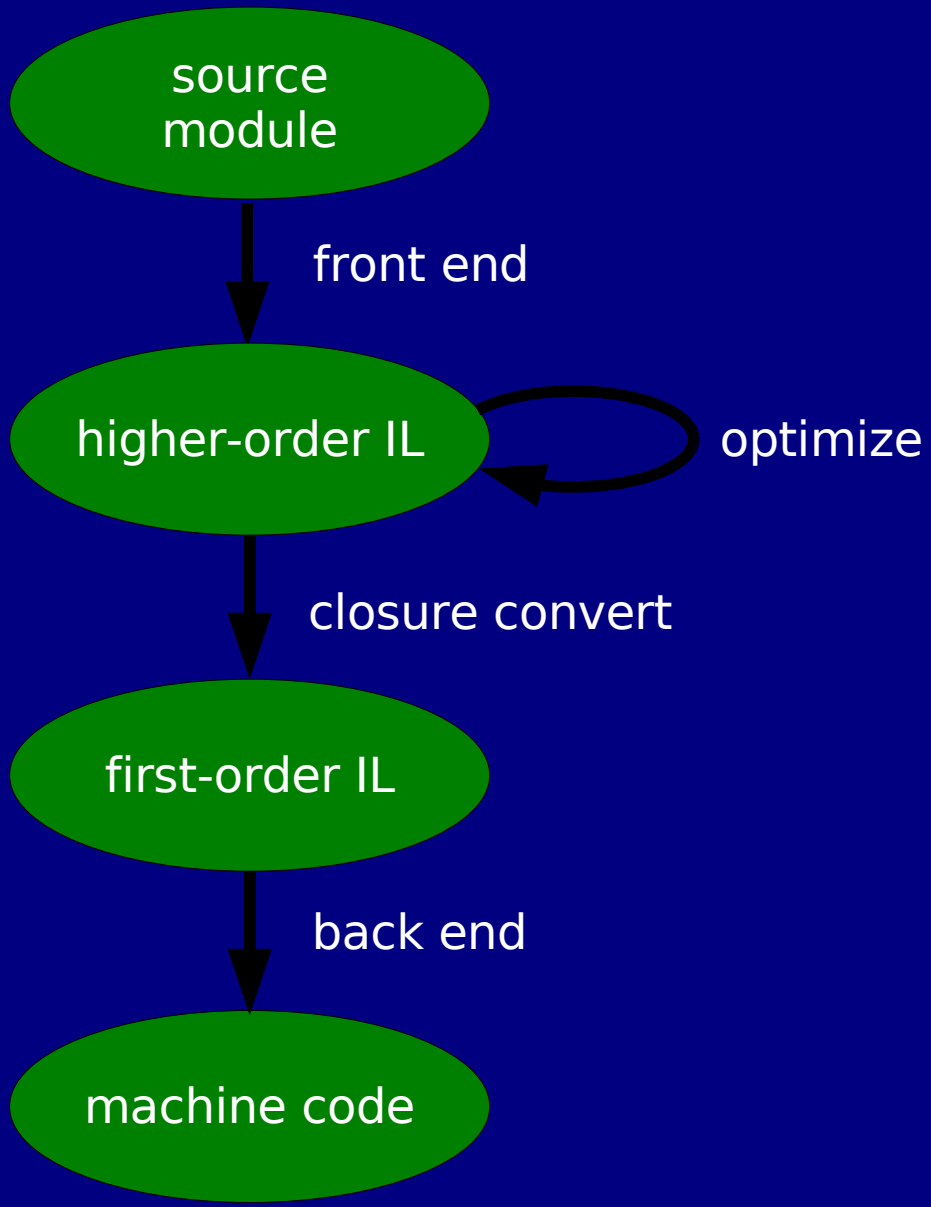


Problems with Traditional Approach

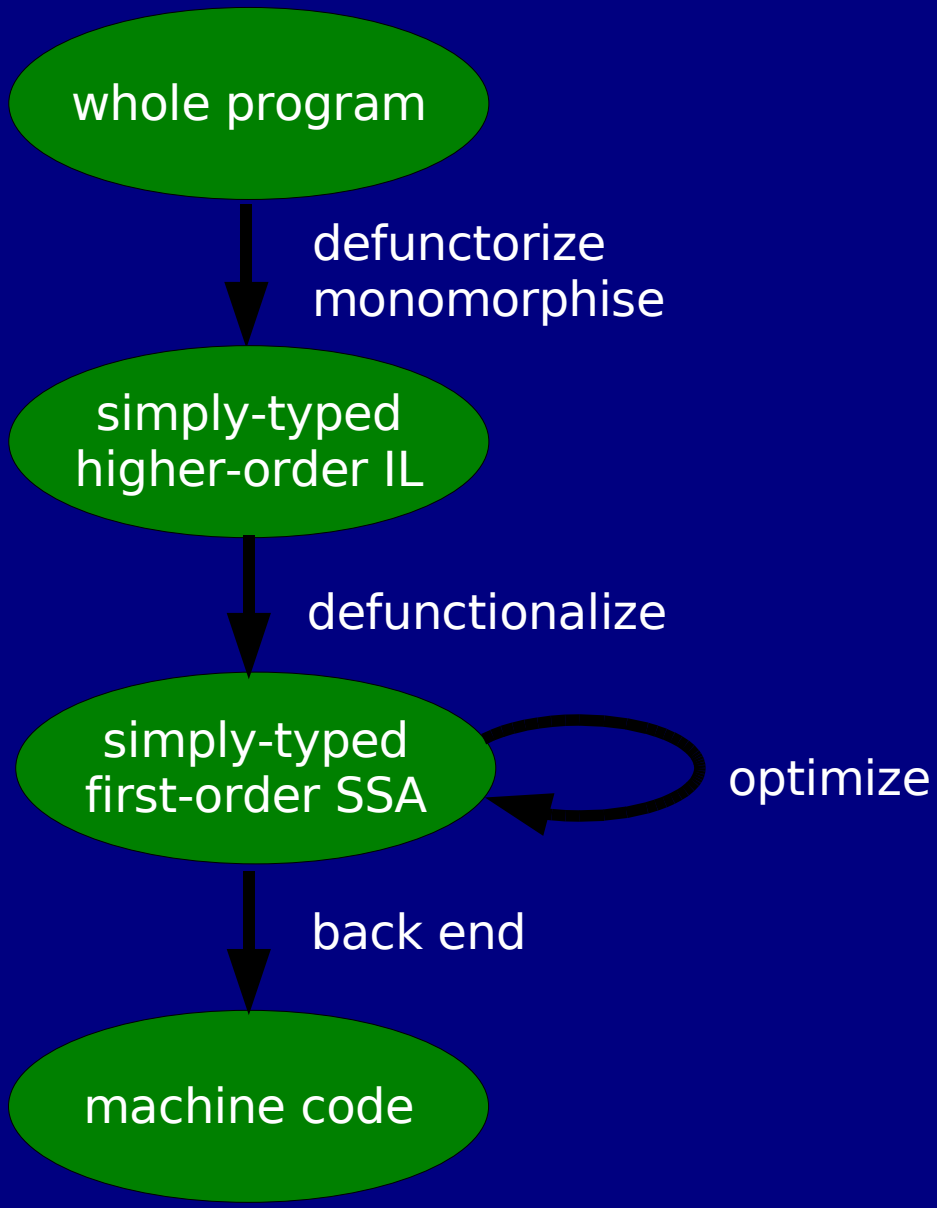


- Separate compilation:
  - bad type info
  - bad control-flow info
- Polymorphic or untyped IL:
  - bad data representations
  - bad dataflow analyses
- Higher-order IL:
  - can't use traditional optimizations
  - optimizations do their own control-flow analysis
- Poor closure optimization.

Traditional Approach

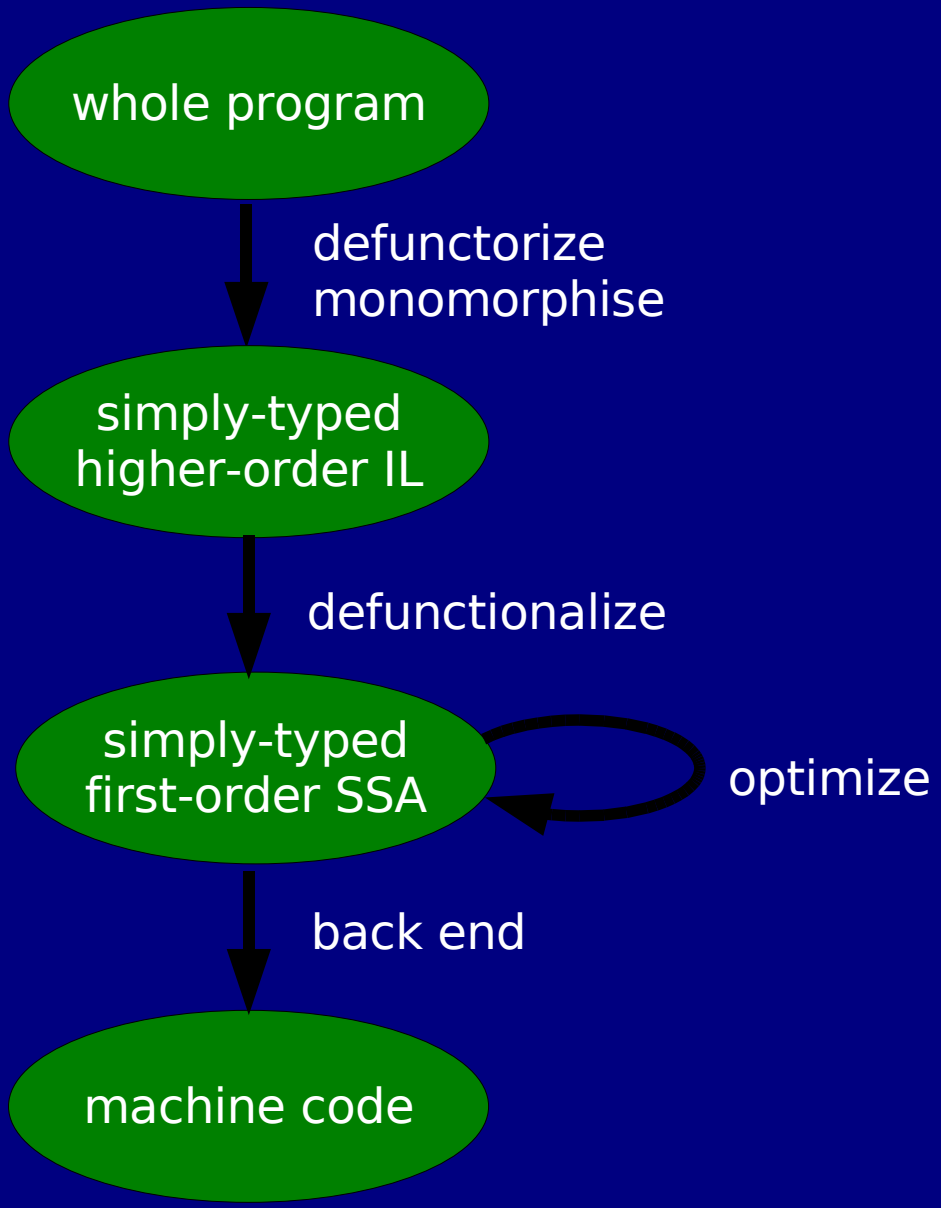


MLton's Approach



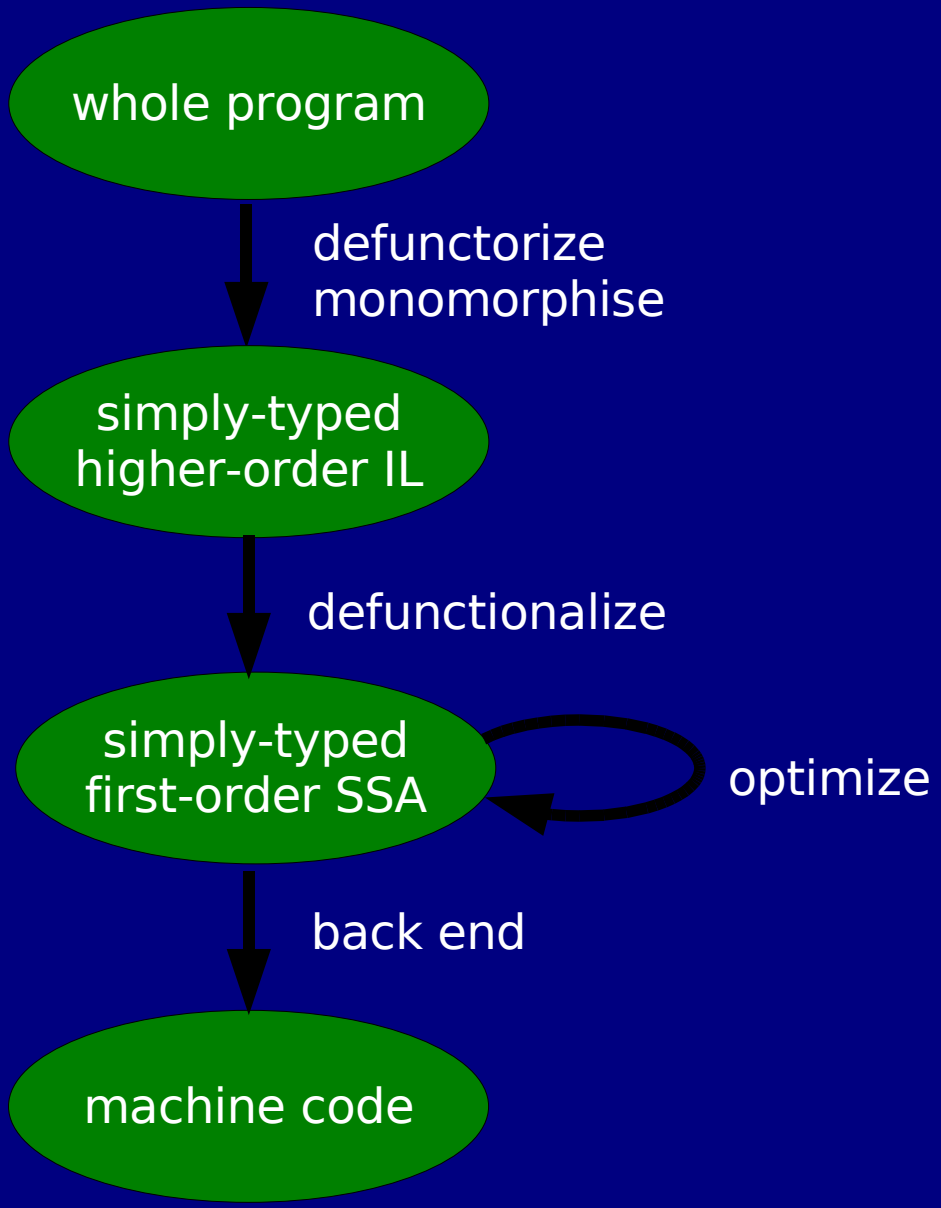
Benefits of MLton's Approach

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



Drawbacks of MLton's Approach

- Compile time.
- Compiler memory usage.
- Executable size.



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.
- The ML Kit also defunctorizes.

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.

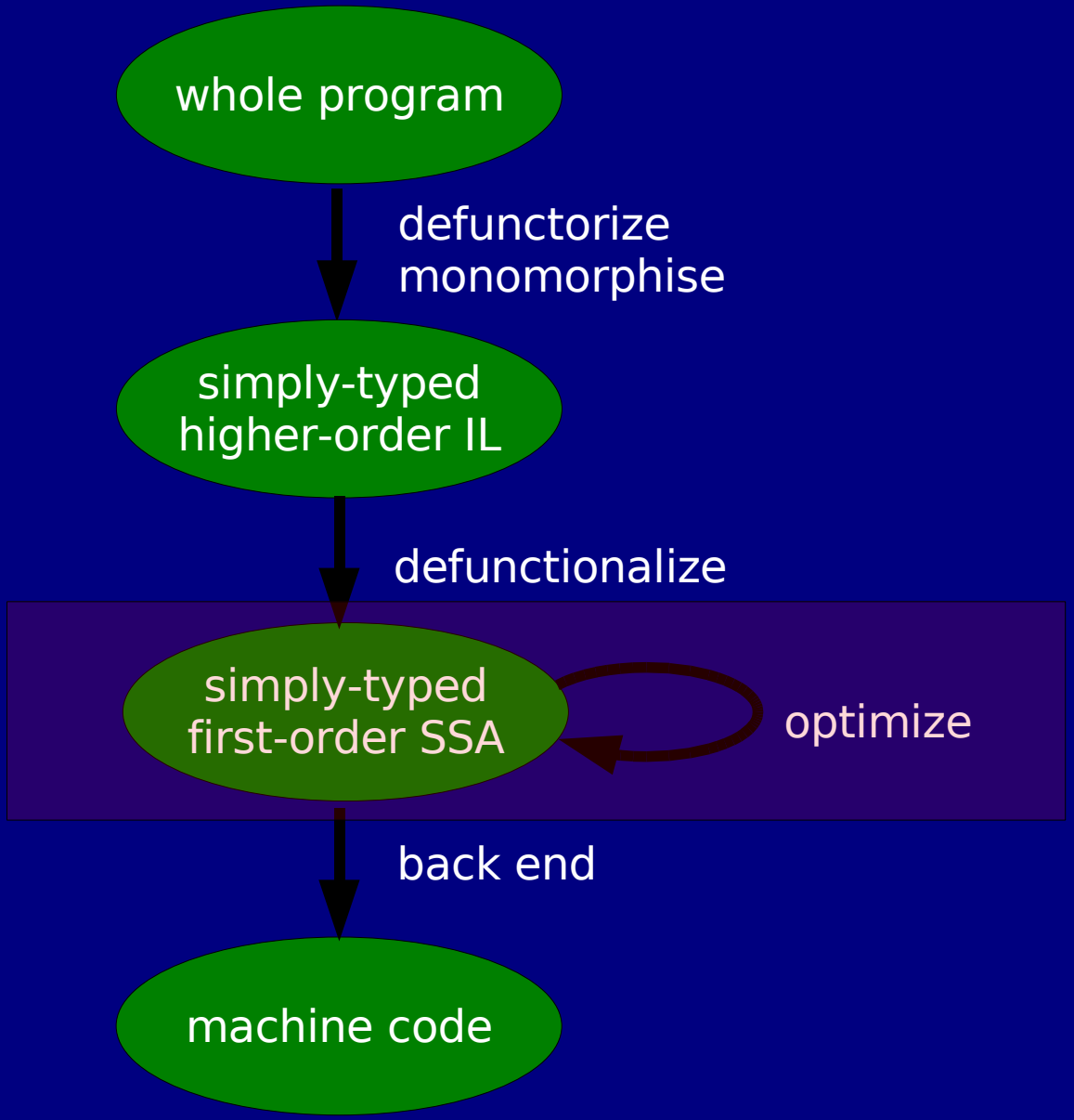
Defunctionalization

- 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 analysis.
  - computes set of functions at each call
- 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.

SSA IL and Optimizer



SSA Intermediate Language

- Traditional, simple IL.
  - simply-typed, first order
  - program = datatypes + functions
  - function = type + arguments + control-flow graph
  - usual SSA conditions: def once, def dominates use
- 250 line interface, 2k line implementation.
  - immutable IL
  - pretty printer, CFG visualizer, DFS, utilities
- 23k lines of code for optimizer.
- MLton options: `-drop-pass, -diag-pass, -keep, -keep-pass, -show-types, -verbose`

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%
- 700 lines of code.

SSA Example: Nontail Fib

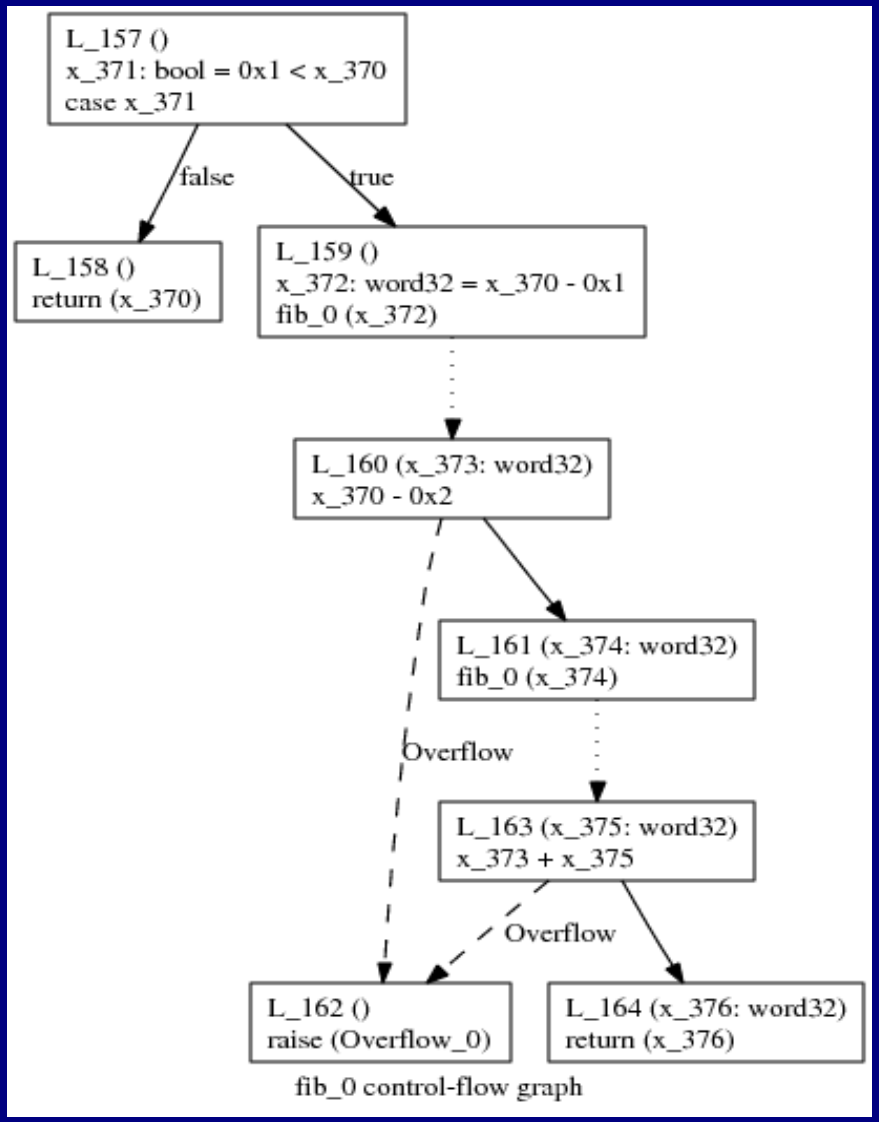
SML Source Function

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

SSA Top-Level Function

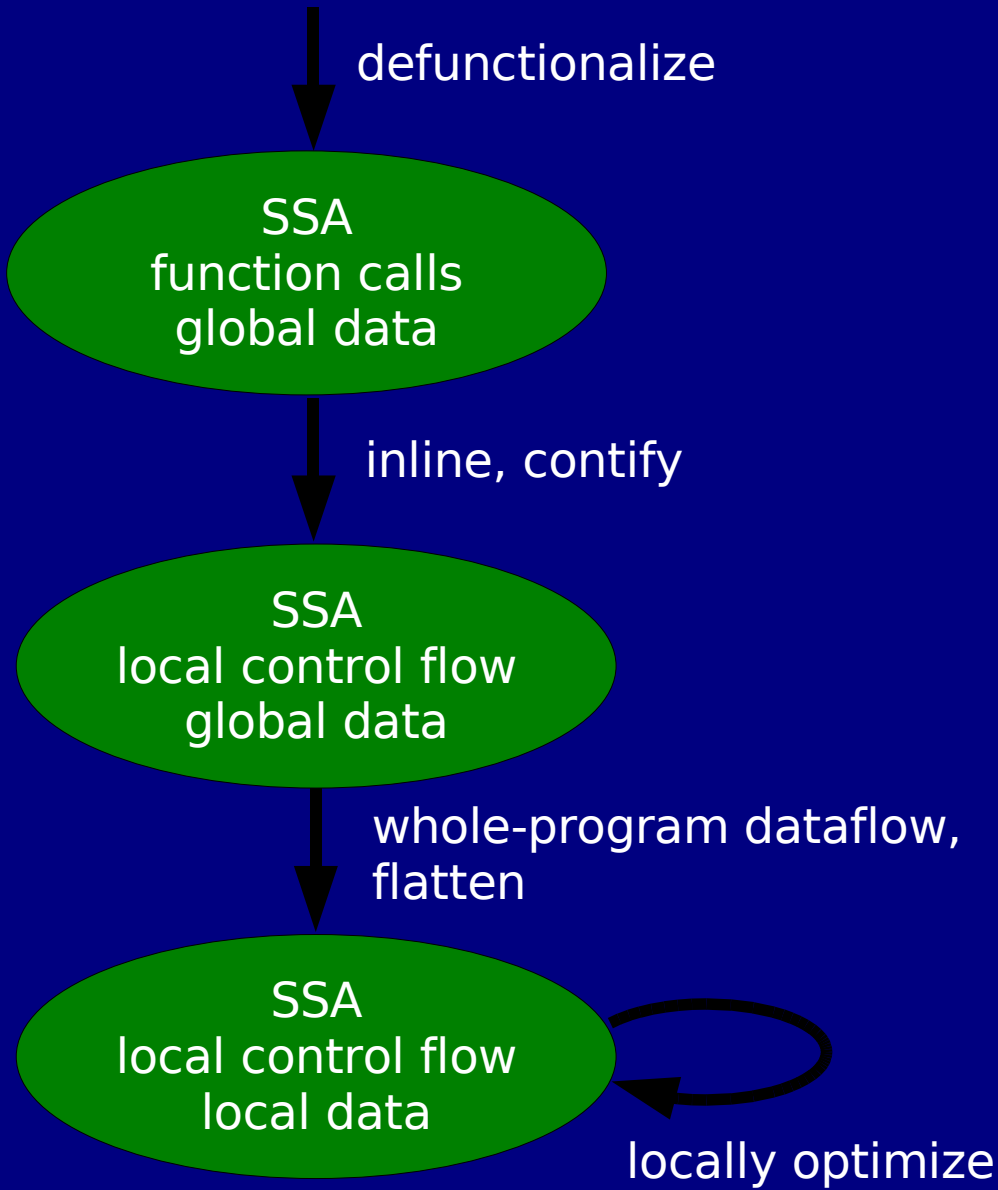
```
fun fib_0 (x_370: word32)
    : word32 =
 goto L_157
```

SSA Control-Flow Graph



SSA Optimizer

- Goals:
  - turn function calls into control-flow graphs
  - expose interprocedural data
  - reduce tuple allocation
  - traditional local optimizations
- Method:
  - 22 small, independent, SSA→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.
- 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.

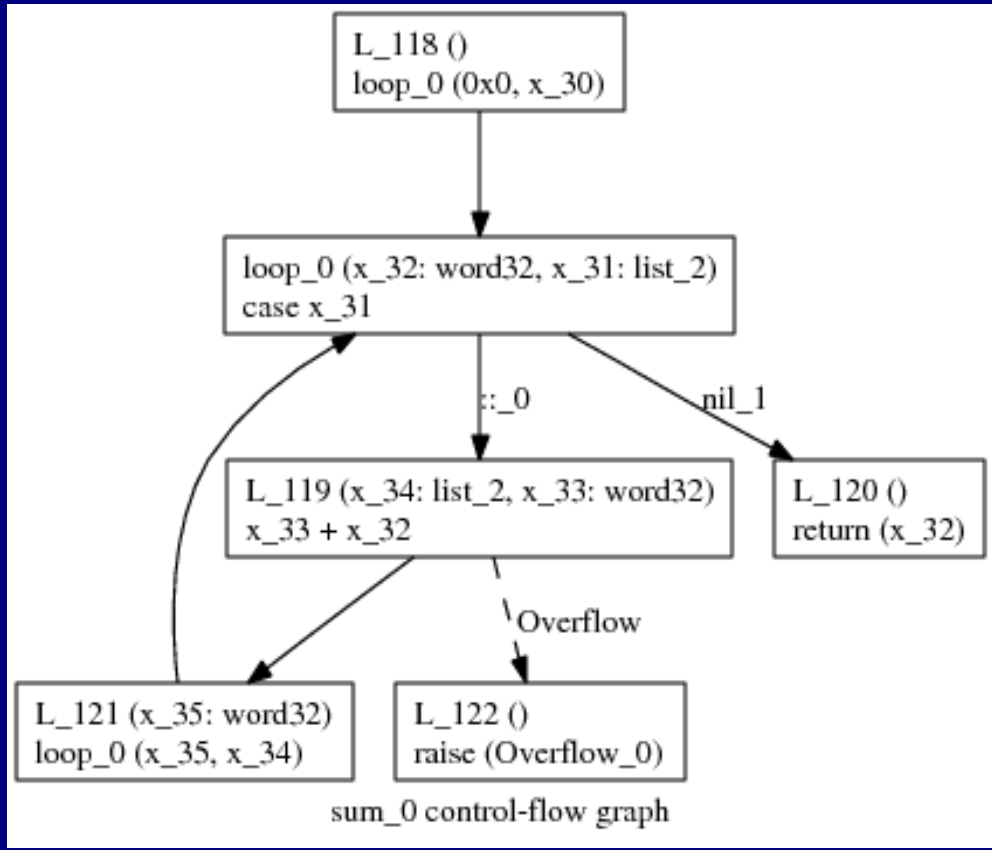
SSA Example: List Fold Becomes a Loop

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 ns =
  fold (ns, 0,
        fn (x, y) => x + y)
```

SSA Control-Flow Graph



sum_0 control-flow graph

SSA Top-Level Function

```
datatype list_2 =
    nil_1
  | ::_0 of (list_2, word32)
fun sum_0 (x_30: list_2): word32 = goto L_118
```

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

SSA Optimizer During a Self Compile

After Pass	#functions(k)	#statements(k)	size(M)
start	26.2	719	101
leaf inline	18.5	555	72
contification	7.7	536	68
const prop	7.7	341	50
inline	1.8	491	58
flatten	1.8	387	53
local opts	1.8	365	52

Data Representation

- 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.

Performance: SML Compilers

- `http://mlton.org/Performance`
- Compares: ML Kit, Moscow ML, MLton, Poly/ML, SML/NJ.
- 40+ benchmarks up to 4k lines.
- MLton faster on all benchmarks but two.
- Run-time ratios over all benchmarks:

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

Performance: Shootout

- `http://shootout.alioth.debian.org`
- 18 micro-benchmarks comparing 30+ languages.
- C/C++/D top tier.
- Haskell/MLton/OCaml second tier, within 2x top.
  - Haskell: `-O2 -optc-O3 -funbox-strict-fields`
  - Ocaml: `-noassert -unsafe -cc-opt -O3 -inline 10`
  - MLton:
- Microbenchmarks helpful to compiler writers, but miss the point for users and whole-program optimization.

Performance: Large Programs

- Large successes:

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).

Technical Lessons

- Whole-program compilation is feasible.
  - compile a 100k line program in minutes with 1G RAM
  - myths: defunctorization, momonorphisation, 0CFA
- Whole-program compilation is effective.
  - fast code and compact data representations
  - total information ⇒ optimizations rewrite at will
- Whole-program compilation is simple.
  - simplifies compiler
  - simplifies optimizations
  - simplifies intermediate languages

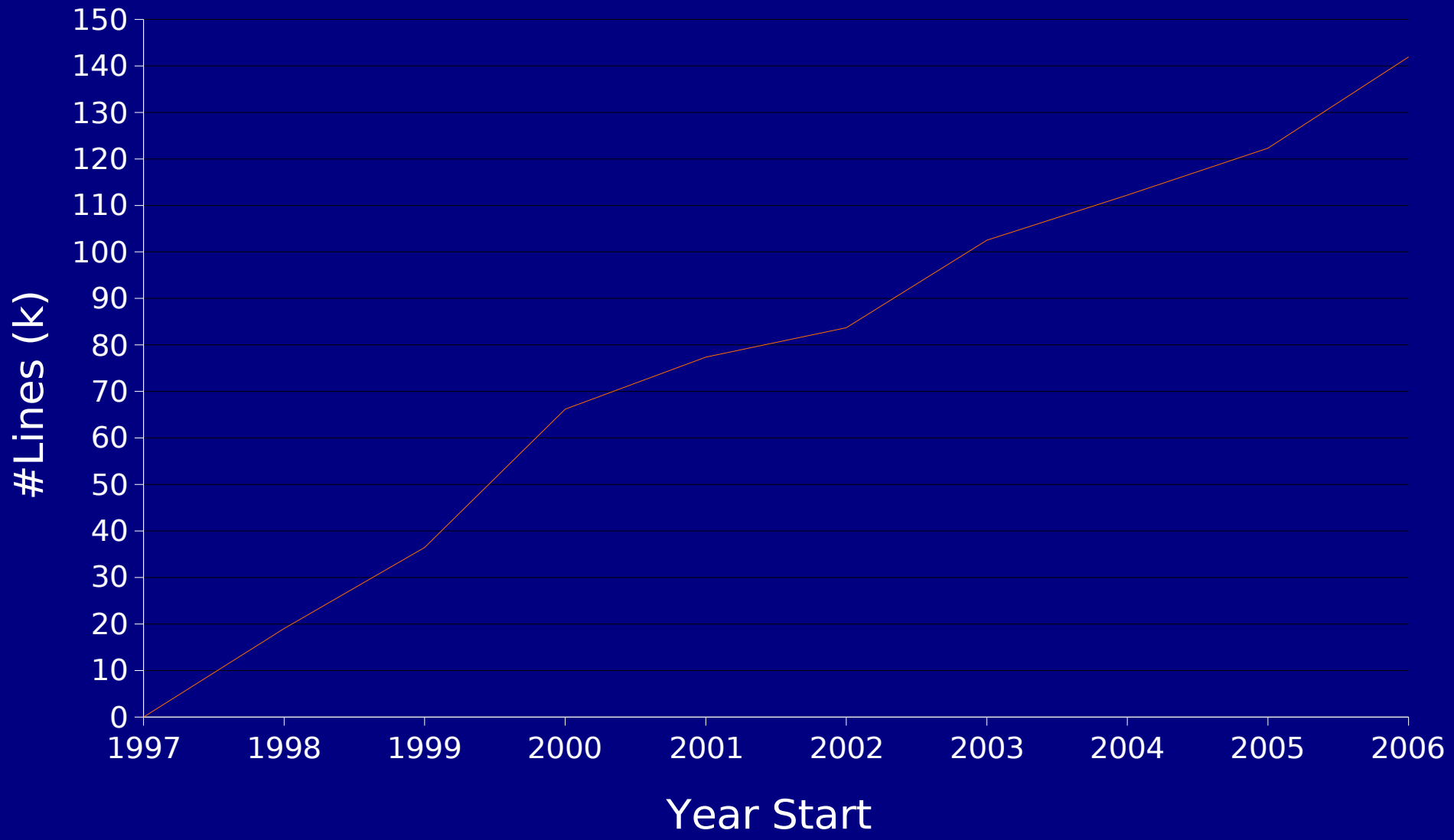
Technical Lessons

- 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

MLton Timeline

1993 research into higher-order flow analysis
1996 experiments with SML/NJ
1997 development starts
1999 first public release, hosted at NEC
2000 x86-codegen started
2001 x86-codegen released, CVS
2002 cross-compilation, new GC, SourceForge, public list
2003 profiler, Sparc/Solaris, complete basis, mlton.org
2004 real front end, new platforms, MLB, CML, wiki
2005 new platforms, SVN, BSD license
2006 64 bits

Lines of Code in the Compiler



Future Work

- `mlton.org/Projects`
  - new optimizations
  - new tools: debugger, heap profiler, interpreter
- Multicore, STM.
- Native codegens: C--, MLRISC, LLVM.
  - MLton wins because of its SSA optimizer, and in spite of its simple native codegen
- New language features and new languages:
  - Haskellton, OcaMLton, sMLton
  - polymorphism: non-uniform, higher-order, recursion, defunctionalization

MLton Users

- `http://mlton.org/Users`
- Commercial:
  - AnswerMine, PolySpace, Sourcelight
- Applications:
  - ADATE, ConCert, Guuglehupf, HOL, mlftpd, RML, SMLNJTrans, STING, Tina, Twelf
- Compiler writers:
  - MLOPE, SSAPRE, Δ-CFA and Γ-CFA, Sharing-constraint errors, Stabilizers

Project Lessons

- Users matter.
  - packaging, platforms, bugs, licensing, ...
  - friendliness and promptness on lists
  - surveys
- Marketing matters.
  - mlton.org, t-shirts, progress report
- Community matters.
  - open mailing lists with public archives
  - wiki
  - SVN access (with commit rights)
- Infrastructure matters.
  - SVN, ViewCVS, wiki, mailing lists

Join Us

- Join the MLton or Mlton-user mailing lists.
  - `http://mlton.org/Contact`
- Discuss SML programming techniques.
- Use MLton/SML to build your next project.
- Try out your own analysis or optimization.
  - SSA IL is simple and has lots of infrastructure
  - whole program optimization gives lots of info
  - SML has lots of nice, large examples
- Use MLton as an optimizer for another language.
- Use MLton as input for another code generator.

MLton Credits

Design: Henry Cejtin, Suresh Jagannathan, Matthew Fluet, Stephen Weeks

Implementation: Matthew Fluet, Stephen Weeks

Code: gdtoa, GnuMP, ML Kit, Moscow ML, SML/NJ

Companies: NEC, PolySpace

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