Reusable, Generic Compiler Analyses and Transformations

Jeremiah Willcock, Andrew Lumsdaine, and Daniel Quinlan
Indiana University and Lawrence Livermore National Laboratory

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by the Laboratory Directed Research and Development Program at LLNL under project tracking code 07-ERD-057.

Motivation and overview

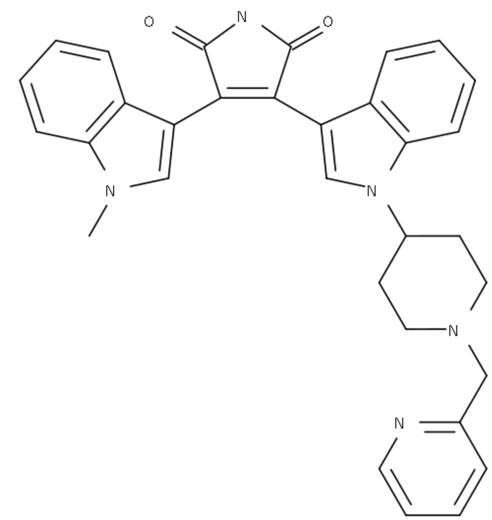
- Compiler optimizations are limited to the optimizations and types built in by the compiler writer
- Cannot be extended to user-defined types
- Cannot be extended with user-defined (high-level) optimizations
- Leverage ideas from generic programming to enable
  - Applying optimizations to classes of types
  - Extending compiler with new optimizations

Optimizations are like pharmaceuticals

- Vendors work on “blockbusters”
  - Optimizations that apply to many programs
  - Tend to be low-level
- Many other optimizations are left out
  - Not enough impact to justify implementing
- See Robison, “Impact of Economics on Compiler Optimization” (Java Grande/ISCOPE 2001)

“Orphan” optimizations

- We all have application-specific optimizations that we want
- None of them by itself is worthwhile to put into a production-grade compiler
- Therefore, vendors will not add them
  - And users cannot add the optimizations themselves
- But users would still benefit from them
  - Both for performance and readability



Compilers lack high-level optimizations

- Consider ATLAS (auto tuning)
  - Well-studied problem (matrix-matrix multiplication)
  - Needs hand-applied, library-specific optimizations
- User-defined data types have no custom optimization support at all
  - But would benefit from having such support
  - Example: $\mathrm{std::list}$ (can cancel iterator $++$ and $--$, etc.)
- Functional language compilers do some because of guarantees on the algebraic structure of data types
  - But there is more that cannot be done that way

Optimization reuse

- Good optimizations are hard to write
  - Many corner cases (pointers, casts, exceptions, etc.)
  - Use results of pointer analysis, path-sensitivity, etc.
- Users are not able to write them
- Compiler writers do not want to write too many
- Reuse of a few optimizations for different tasks would mitigate these problems

Benefits of optimization reuse

- Better performance of user code
- Compilers more effective and easier to write
- Allows user-written, sophisticated optimizations by even unsophisticated users by building from expert-written generic optimizations
- Increased adoption of abstract data types due to simpler interfaces
  - cf. Mateev et al's matrix library

Identities

- Many types and operations have similar identities:

$$\textbf{int } \mathrm{x};$$
$$\textbf{int } \mathrm{y} = \mathrm{x} + 0;$$
$$\quad \rightarrow \mathrm{y} = \mathrm{x}$$

$$\textbf{double } \mathrm{w};$$
$$\textbf{double } \mathrm{v} = \mathrm{w} * 1.;$$
$$\quad \rightarrow \mathrm{v} = \mathrm{w}$$

$$\mathrm{matrix\ m};$$
$$\mathrm{matrix\ m2} = \mathrm{mul(m, identity(nrows(m)))};$$
$$\quad \rightarrow \mathrm{m2} = \mathrm{m}$$

Monoids

- In all of these cases, operation with an identity is a null operation (and can be removed)
- Mathematicians have a name for all operations with the identities $0 + x \to x$ and $x + 0 \to x$: a *monoid*
  - Binary associative operator with identity
- ***Write the optimization in terms of monoid***
- **One** optimization can optimize all monoids
  - Including all previous cases
  - Even though they seem very different

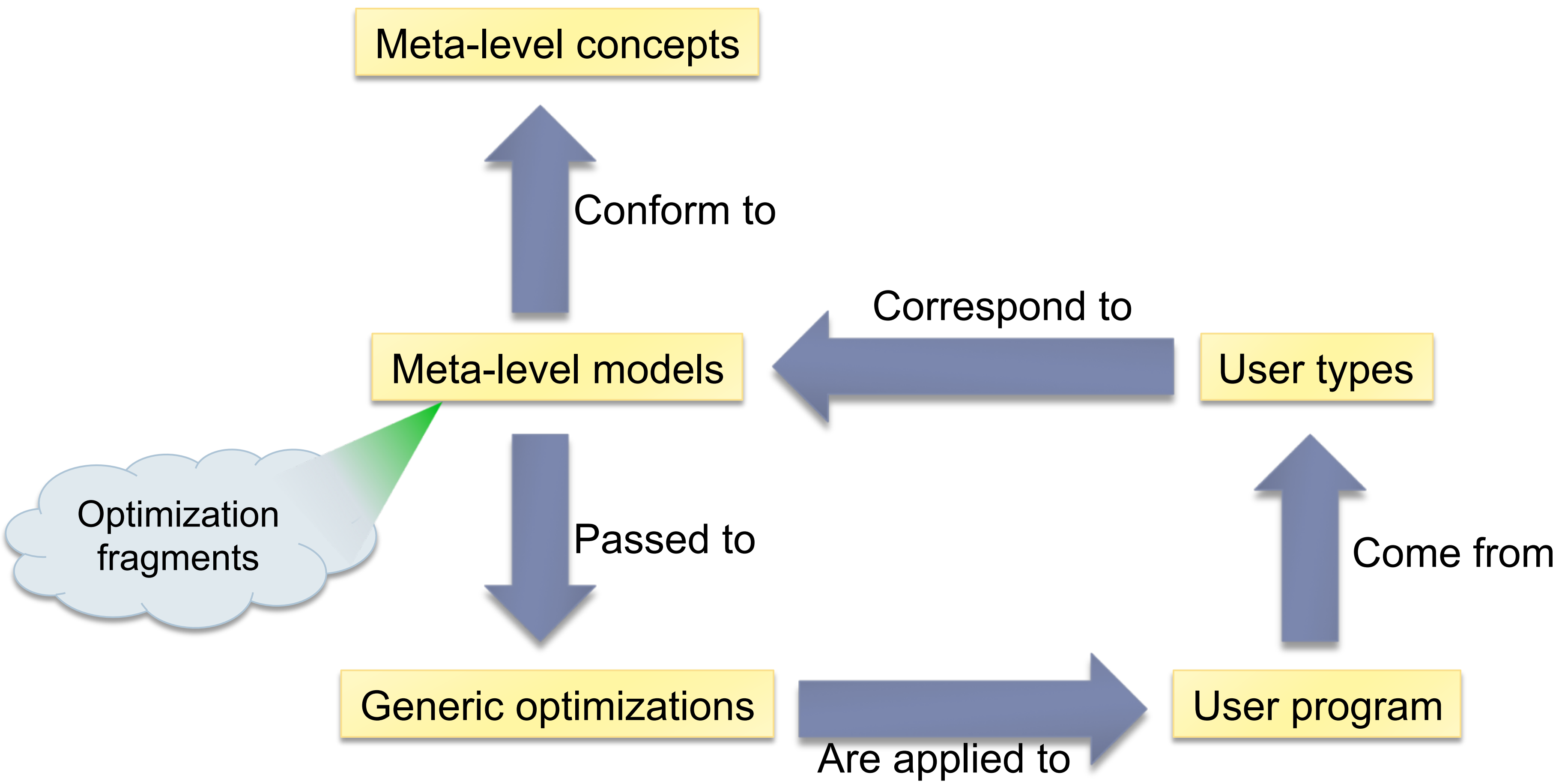
Generic programming

- An organizational principle for software libraries
  - Based on properties of types
- Three major components:
  - Concepts: constraints on types
  - Models: satisfaction of those constraints
  - Generic algorithms/data structures: apply to all types that model certain concepts
- Similar constructs are in several languages

Concept-based optimization

- Implementing compiler optimizations using the generic programming approach allows reuse
- Optimizations either in compiler, library, or individual program
- Reuse allows:
  - Higher-quality optimizations
  - Reduced effort
  - Optimizations by users

Concept-based optimization



Meta-level concepts and models

- Meta-level concepts are requirements for fragments
- Meta-level models provide the fragments
  - Code run within a larger optimization
- Optimizations are generic programs at the meta-level
- Can be implemented via Haskell-style dictionaries

Monoid meta-level concept
• Find identity elements ➔ Set of program expressions • Find binary operation ➔ Set of program expressions and pairs of arguments

Optimization fragments

- Analysis and transformation fragments contain parts of a full optimization
- Fragments are customized for each type in program
- Analysis fragments locate program points
  - That do a particular operation, modify a variable, etc.
- Transformation fragments modify the program
  - Change an operation found by an analysis fragment, etc.

$$\begin{array}{l} \mathrm{x} = \mathrm{y}; \\ \mathrm{z} = \mathrm{w}; \end{array} \quad \Longrightarrow \quad \begin{array}{l} \langle s_1, \mathrm{x}, \mathrm{y} \rangle \\ \langle s_2, \mathrm{z}, \mathrm{w} \rangle \end{array}$$

Optimizing a program

- Optimizations applied for each combination of input types and operations in the program
- Changes are applied after all optimizations
  - To avoid invalidating analysis results

Identity removal	**int**, $+$ **double**, $*$ matrix, $+$
Copy propagation	**int**

```
int x, y;
double z;
matrix m;
x = nrows(m + matrix(0));
y = x;
z = (double)y * 1.;
```

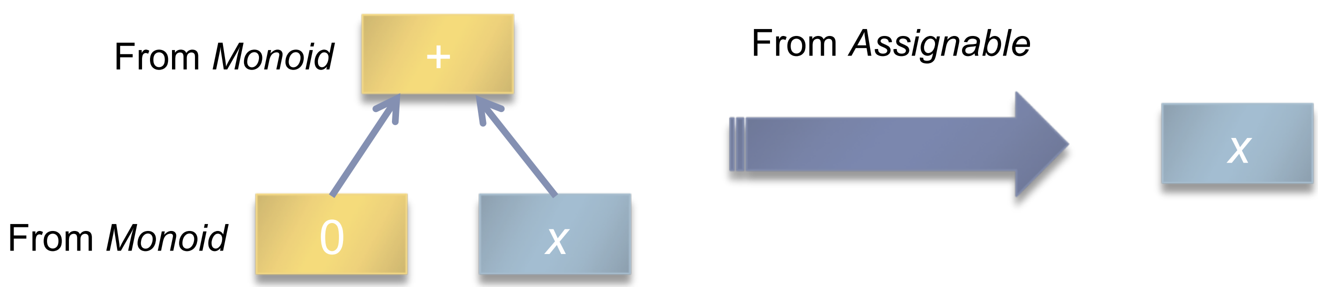
```
// ...
x = nrows(m);
y = x;
z = (double)x;
```

Proofs of concept

- Feasibility demonstrated with prototypes
  - Regular-expression-based optimization specification language
  - Traditional flow equations
- Both are embedded into Scheme and apply to simple C++ programs (using the ROSE framework)

Identity operation removal

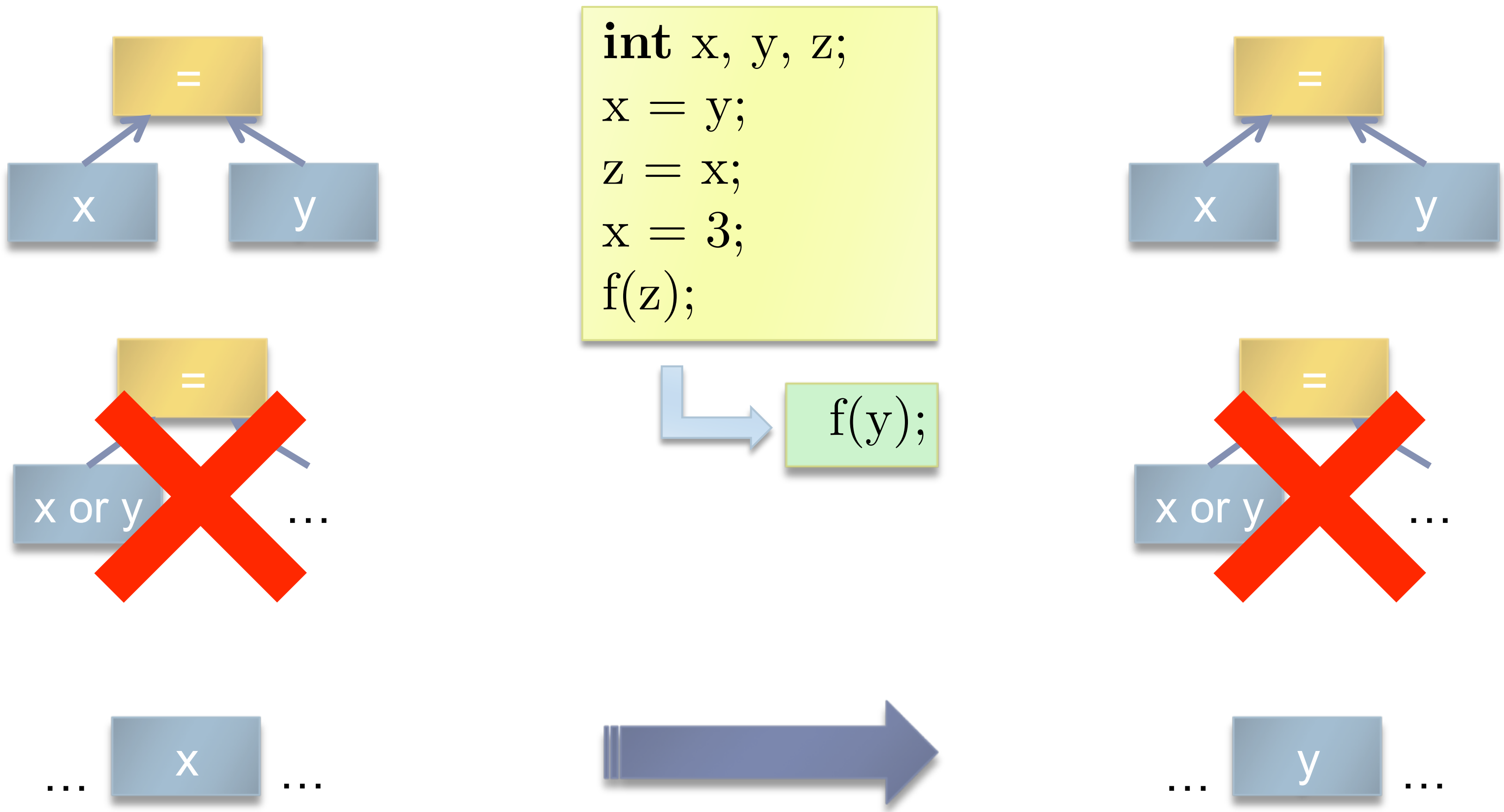
- $0 + x \to x$ and $x + 0 \to x$ (for generalizations of 0 and $+$)
- Applies to any monoid
- Two meta-level concepts required: **Monoid** and **Assignable**



- Transforms $\textbf{int}\ \mathrm{w} = 0 + (\mathrm{x} + 3 * \mathrm{y});$ to $\textbf{int}\ \mathrm{w} = \mathrm{x} + 3 * \mathrm{y};$

Generic copy propagation

- Only **Assignable** is required



Conclusions

- Generic optimizations allow optimizations to be applied to entire classes of types
- Optimizations can be encoded in library to extend compiler
- Optimizations can be reused
- Feasibility demonstrated with implementation

Future work

- Analysis and transformation fragments that work on many types at once
- Ordering and profitability of generic optimizations
- Using axioms or a different high-level specification language
- Generic transformations in MetaOCaml
- User-defined type optimization in Haskell or other languages