Compiler Technology in Gwydion

Robert A. MacLachlan

October 28, 1993

Overview

The purpose of a compiler is to provide an *efficient* implementation of language *semantics.* Gwydion differs from other programming environments because:

- Dylan semantics are different from conventional implementation and prototyping languages.
- The Gwydion compiler will apply different techniques to efficiently implement these semantics.
- Efficient implementation of high-level operations allows programs to be written differently.
- Instead of being text or syntax oriented, Gwydion is *semantics* oriented.
- Incremental semantic analysis is stored in a shared OODB, allowing consistent concurrent browsing of multiple semantically-based views.

Efficiency

Delivered speed:

- State-of-the-art back-end technology (register allocation, etc.)
- High-level optimization of high-level programs.
- Special techniques developed for efficient compilation of dynamic languages.
- Whole-program compilation.

Delivered space:

- Shared libraries.
- Library-oriented system architecture with non-resident development tools (as opposed to monolithic development/runtime environment.)
- Static demand linking of referenced library components.
- Byte compilation for rarely used code.

Semantics and Optimization

Dylan differs from conventional implementation languages such as C++:

- Dynamic typing provides the lowest barrier to evolution and reuse.
- Function closures greatly increase the power of functional abstraction.
- The Dylan object model is significantly more powerful and pervasive.
- Augmenting the compiler with arbitrary transformations provides a unique degree of language extensibility.

Dylan differs from prototyping languages in that:

- It is designed with efficiency in mind, and
- Rather than being specialized, it is specializable.

High Level Programming

High-level programming via extension is the hidden secret of high-productivity general-purpose languages such as Smalltalk and Lisp:

- Extensible systems provide a sound base on which special-purpose high-level operations can be built.
- Designing new high-level operations is inevitably hard, but profitable (a form of reuse.)
- Efficient extension is easier than writing a whole new compiler.

Programming at a high level:

- Improves expressiveness and programmer productivity,
- Hides irrelevant implementation details, and
- Has a reputation for poor implementation efficiency.

Extensibility and Metaprogramming

High-level programming via extensions of a simple language can be efficient because:

- Missing high level operations can easily be added; programmers are not forced to use high-level operations inappropriately (and inefficiently.)
- When necessary, speed-critical portions of programs can recoded using lower-level operations without requiring an abrupt transition to a different low-level language.
- High level operations can be optimized automatically based on their underlying implementation (partial evaluation.)
- Meta-programming of the compiler allows extensions to optimize themselves according to their own semantics.

Optimization and Tuning

The strength of high-level programming (hiding the implementation) is also a weakness:

- Efficient coding requires knowledge of the costs of operations; in low-level languages, these costs are fairly constant and context-independent.
- High-level operations require high levels of optimization for acceptable efficiency, and
- High-level optimization relies on exploiting obscure context dependencies.
- The tools for efficiency may exist, but it isn't clear when and how to use them.

Accountable Optimization

Accountable optimization eliminates the need for a mental efficiency model by telling the programmer:

- when an inefficient implementation is chosen for non-obvious reasons,
- how serious the situation is,
- precisely what part of the program is being referred to, and
- why the inefficient choice was made (and how to enable efficient compilation.)

Unconditional output from a file compiler can easily overload the programmer with spurious information. Gwydion's optimization tools will integrate compiler information with profiling results and complexity-managing browsers.

Partial Evaluation

Partial evaluation is a generalization of constant folding. Instead of only replacing completely constant expressions with the compile-time result of the expression, partial evaluation transforms arbitrary code using information on parts of the program that are constant.
Partial evaluation comes into its own when programs are *written to be optimized:*

- Tuned programs written in low-level languages show little benefit from partial evaluation because they are already hand-optimized.
- Partial evaluation helps the efficiency of high-level programming by specializing high-level operations according to their context.
- Partial evaluation helps the efficiency of dynamic and polymorphic (or OO) typing by eliminating unused generality.

Partial Evaluation Example

```
(declaim (inline find-in))
(defun find-in (next element list &key (key #'identity)
                                       (test #'eql test-p)
                                       (test-not nil not-p))
  (when (and test-p not-p)
    (error "Both :Test and :Test-Not supplied."))
  (if not-p
      (do ((current list (funcall next current)))
          ((null current) nil)
      (unless (funcall test-not (funcall key current)
                                element)
        (return current)))
      (do ((current list (funcall next current)))
          ((null current) nil)
      (when (funcall test (funcall key current) element)
        (return current)))))
```

Partial Evaluation Result

When inline expanded and partially evaluated, this transformation results:

```
(find-in #'foo-next 3 foo-list
         :key #'foo-offset :test #'<)
⇒
(do ((current foo-list (foo-next current)))
    ((null current) nil)
  (when (< (foo-offset current) 3)
    (return current)))
```

Whole-Program Compilation

Whole-program compilation is the opposite of separate compilation:

- Separate compilation reduces the time and space overhead of compilation at the cost placing an arbitrary horizon on optimization.
- High-level dynamic languages need whole-program compilation in order to optimize across abstraction levels and to remove spurious dynamism.
- Gwydion is unusually well-positioned to exploit whole-program compilation, since all code resides in a unified persistent database.
- Incremental semantic analysis and compilation minimize the need for whole-program *recompilation.*

Fast Development Cycle

- Studies of the advantages of dynamic prototyping environments often find the fast design—implement—evaluate cycle to be the primary cause of improved productivity.
- Because of its quantifiable and immediate benefits, environments for C++, etc., are starting to adopt incremental compilation.
- Although Gwydion emphasizes support for new development paradigms such as evolution and high-level programming, we still believe responsive development tools are important.
- Gwydion advances incremental development into new areas: concurrent incremental design, coding, testing and documentation. Quick transition between all these functions is required to avoid short-term memory loss.

Environment Integration

The Gwydion compiler is highly integrated with the rest of the programming environment:

- The compiler implements language semantics; because the compiler is integrated with the environment, the environment can be *semantics directed.*
- A semantically aware editor can provide clear, prompt notification of type errors, dead code, or other semantic problems.
- Since the database contains semantic information (like cross-references), it can be searched by semantic criteria, such as "who extends this method."
- Incremental tools with this level of integration are only possible when the compiler and environment are co-designed.

Summary

- Efficiency and semantics are both environment issues; the compiler is the primary analytic part of the environment.
- We have experience implementing practical highly optimizing compilers.
- We know how to efficiently implement evolving high-level programs.
- We are exploiting new technology (OODBs) to allow both new implementation techniques (whole-program compilation) and new environment features (multiple semantic views.)