Chapter 8

Concluding Remarks and Future Work

‘‘So many things are possible just as long
as you don’t know they’re impossible.’’
– Norton Juster, *The Phantom Tollbooth*

1. Introduction

In this chapter I recapitulate the main result of this dissertation, distill some practical lessons learned in the design process, talk about the caveats of language design, and give directions for future research.

2. Main result

My thesis is that logic programming can execute as fast as imperative programming. For this purpose I have implemented a new optimizing Prolog compiler, the Aquarius compiler. The driving force in the compiler is to specialize the general mechanisms of Prolog (i.e. the logical variable, unification, dynamic typing, and backtracking) as much as possible. The main ideas in the compiler are: the development of a new abstract machine that allows more optimization, a mechanism to generate efficient code for deterministic predicates (converting backtracking to conditional branching), specialization of unification (encoding each occurrence of unification in the simplest possible way), and the use of global dataflow analysis to derive types.

The resulting system is significantly faster than previous implementations and is competitive with C on programs for which dataflow analysis is able to do sufficiently well. It is about five times faster than Quintus Prolog, a popular commercial implementation.

3. Practical lessons

During the design of this compiler I have found four principles useful.

(1) **Simplicity is common**. Most of the time, only simple cases of the general mechanisms of the language are used. For example, most uses of unification are memory loads and stores. Many of these simple cases are easily detected at compile-time.

(2) **Use the design time wisely**. There are many possible optimizations that one can implement in a compiler of this sort. To get the best results, rank them according to their estimated performance gain relative to their implementation effort, and only implement the best ones. Do not be distracted by clever ideas unless you can prove that they are effective.

(3) **Keep the design simple**. For each optimization or transformation, implement the simplest version that will do the job. Do not attempt to implement a more general version unless it can be done without any extra effort. It is easy to become entangled in the mechanics of implementing a complex optimization. Often a simple version of this optimization achieves most of the benefits in a fraction of the time.

(4) **Document everything, including bugs**. Documentation is an extension to one's memory and it pays for itself quickly. The mental effort spent in writing down what one has done results in a better recollection of what happened. In this design, I have maintained two logs. The first is a file in chronological order that documents each change and the reason for it. The second is a directory containing bug reports contributed by the users of the compiler and brief discussions of the fixes.

The first three of these principles are corollaries of what is sometimes called the ‘‘80-20 rule’’: 80% of the results are obtained with 20% of the effort. Using this principle consistently was very important for my work and for the BAM project as a whole.

4. Language design

The Prolog language is only an approximation to the ideal of logic programming. During this research, our group has grappled with some of the deficiencies of Prolog. There are deficiencies in the area of logic: Prolog's approximation to negation (i.e. negation-as-failure) is unsound (i.e. it gives incorrect results) when used in the wrong way. Prolog's implementation of unification can go into infinite loops when creating circular terms. The default control flow is too rigid for data-driven programming.

There are deficiencies in the area of programming: The correspondence between a program and its execution efficiency is not always obvious. Unification is only able to access the surface of a complex data structure. Because the clauses of a predicate are written separately, many conditions have to be repeated or extra predicates have to be defined. There is a sense in which Prolog is a kind of assembly language.

All of the above problems have solutions, some of which have been implemented in existing systems and in the Aquarius system. However, for three reasons I have resisted the impulse to change the language more than just a little. First, of all logic languages, the Prolog language has the largest and most vigorous user community, and this is a resource I wanted to tap. There are many programs written in Prolog, in various styles, and I wanted to see if this existing pool of ingenuity could be made to run faster. Second, it is unwise to change more than one component of a system at the same time, especially if they can interact in unpredictable ways. That is, one should not design a new language and a compiler for it at the same time. Third, I do not deem myself competent yet to design a language. I believe in the rule of bootstrapped competence: Before writing a compiler, write programs. Before designing a language, write compilers. Competence in each task is limited by competence in its prerequisite.

The best languages are those which distill great power in a small set of features. This makes such languages useful as tools for thought as well as for implementation. Practical aspects such as how efficient it can be implemented are as important in a good language design as theoretical aspects. A good language is theoretically clean (i.e. easily understood) as well as being efficiently implementable. Examples of such languages are Pascal (many algorithms are specified in an Pascal-like pseudo-code), Scheme, and Prolog. To create such a language, a person must have completely digested a set of ideas as well as have a large amount of practical experience. This is a difficult combination—it is easy to gloss over the areas one does not know well.

5. Future work

The goal of achieving parity with imperative languages has been achieved for the class of programs for which dataflow analysis is able to provide sufficient information, and for which the determinism is accessible through built-in predicates. To further improve performance these limits must be addressed.

To guide the removal of these limits it is important to build large applications and study the interaction between programming style and the implementation. This is a problem of successive refinement. A more sophisticated implementation catalyzes a new style of programming, which in its turn catalyzes a new implementation, and so forth. The first step in this process was the development of the first Prolog compiler and the WAM. The Aquarius system is only the second step. It is able to generate efficient code from programs written in a more logical style than standard Prolog. However, the limits of this style are not yet understood as they are in the WAM. Further work in this area will lead to a successor to Prolog that is closer to logic and also efficiently implementable.

5.1. Dataflow analysis

When writing a program, a programmer commonly has a definite intention about the data's type (intending predicates to be called only in certain ways) and about the data's lifetime (intending data to be used only for a limited period). Because of this consistency, I postulate that a dataflow analyzer should be able to derive this information and a compiler should be able to take advantage of it.

There has been much good theoretical work on global analysis for Prolog, but few implementations, and fewer still that are part of a compiler that takes advantage of the information. Measurements of the Aquarius system show that a simple dataflow analysis scheme integrated into a compiler is already quite useful. However, the implementation has been restricted in several ways to make it practical. As programs become larger, these restrictions limit the quality of the results. I hope the success of this experiment encourages others to relax these restrictions. For example, it would not be too difficult to:

- Extend the domain to represent common types such as integers, proper lists, and nested compound terms. This is especially important for general-purpose processors.
- Extend the domain to represent variable aliasing explicitly. This avoids the loss of information that affects the analyzer.
- Extend the domain to represent data lifetimes. This is useful to replace copying of compound terms by in-place destructive assignment. In this way dynamically allocated data becomes static. The term ''compile-time garbage collection'' that has been used to describe this process is a misnomer; what is desired is not just memory recovery, but to preserve as much as possible of the old value of the compound term. Often a new compound term similar to the old one is created at the same time the old one becomes inaccessible. Destructive assignment is used to modify only those parts that are

changed.

- Extend the domain to represent types for each invocation of a predicate. For example, the analyzer could keep track not only of argument types for predicate definitions, but of argument types for goals inside the definitions. This is useful to implement multiple specialization, i.e. to make separate copies of a predicate called in several places with different types. For the chat_parser benchmark, making a separate copy of the most-used predicate for each invocation results in a performance improvement of 14%.

5.2. Determinism

The second area in which significant improvement is possible is determinism extraction. The Aquarius compiler only recognizes determinism in built-in predicates of three kinds (unification, arithmetic tests, and type checking). Often this is not enough. In many programs, user-defined predicates are used to choose a clause.