NORTHWESTERN UNIVERSITY

Meta-Compilation of Language Abstractions

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Pankaj Surana

EVANSTON, ILLINOIS

June 2006

ABSTRACT

Meta-Compilation of Language Abstractions

Pankaj Surana

High-level programming languages are currently transformed into efficient low-level code using optimizations that are encoded directly into the compiler. Libraries, which are semantically rich user-level abstractions, are largely ignored by the compiler. Consequently, library writers often provide a complex, low-level interface to which programmers "manually compile" their high-level ideas. If library writers provide a high-level interface, it generally comes at the cost of performance. Ideally, library writers should provide a high-level interface and a means to compile it efficiently.

This dissertation demonstrates that *a compiler can be dynamically extended to support user-level abstractions*. The Sausage meta-compilation system is an extensible source-to-source compiler for a subset of the Scheme programming language. Since the source language lacks nearly all the abstractions found in popular languages, new abstractions are implemented by a library and a compiler extension. In fact, Sausage implements all its general-purpose optimizations for functional languages as compiler extensions. A meta-compiler, therefore, is merely a shell that coordinates the execution of many external extensions to compile a single module. Sausage demonstrates that a compiler designed to be extended can evolve and adapt to new domains without a loss of efficiency.

Table of Contents

List of Figures

Chapter 1. Introduction

Compilers can be dynamically extended to support user-level abstractions. Computer programming is, at its best, the act of building and manipulating abstractions that are closer to the problem domain. These abstractions are layered on top of simpler abstractions, in a tower supported by the syntax and semantics of the host programming language. These abstractions are valuable software engineering artifacts with a specific semantics all their own. However, compilers are oblivious to these semantics and attempt to optimize programs despite the many layers of abstractions. This dissertation describes a method of dynamically extending a compiler to efficiently compile and statically error-check programmer-defined abstractions.

1.1. Example

This example serves to illustrate the issues raised and the solution proposed by this dissertation. Scheme[68] is such a simple language it even lacks a record type. Instead, a variety of modules exist which implement records as type-safe projections on vectors[67]. Since records are implemented by a module of macros and code, the compiler and runtime have no specific optimizations for them. Most record libraries generate functions that perform a type test on the record before returning the value for a slot. The record type test ensures the argument is a vector and the correct type tag is in the first position. In addition, the vector-ref function always checks that the

index is within the bounds of the vector. This implementation is simple and straightforward.

```
(define (make-pair x y) (vector 'pair x y))
(define (pair? p)
  (or (and (vector? p) (eqv? 'pair (vector-ref p 0)))
    (error "Not a pair type.")))
(define (first p) (and (pair? p) (vector-ref p 1)))
(define (rest p) (and (pair? p) (vector-ref p 2)))
```

Figure 1-1. Simple implementation of records

The problem is that every call to `first` and `rest` will perform the same `pair?` test repeatedly, even if the tests appear in the same function body. One solution is to implement the accessors as macros, which is effectively the same as inlining the function body into the program, and rely on the compiler to remove redundant type tests. However, macros cannot be used in higher-order functions, i.e. one could not access the first element of every pair in a list using `map` if accessors were only available as macros. This is an unacceptable limitation on a higher-order programming language, where functions are frequently passed around as first-class values.

Of course, a *sufficiently smart compiler* could inline the body of `first` and `rest`. Still, it would not know that `pair?` is a pure function without inlining it, too. It would also need to inline `vector?`, `eqv?` and `vector-ref` to make sure `pair?` is a pure function, assuming the compiler does not have built-in knowledge about these primitives. Only then could

it use common subexpression elimination[30] to reduce the number of type- and bounds-checking in a function. Since conventional compilers cannot do this automatically, the specific optimizations for record operations are added to the compiler as another set of primitives. For most Scheme systems, however, programmers simply accept the performance penalty of record abstractions.

Though a more advanced compiler might be able to automatically optimize the abstraction penalty incurred by records, the overhead for other abstractions is not so easy to remove. Other user-defined programming abstractions commonly implemented in Scheme include objects, generic methods, list comprehensions, pattern matching, backtracking, concurrency, and so on. Abstractions also include operations commonly found in libraries, e.g. polymorphic I/O, data structures, distributed programming and graphics, to name just a few. To correctly use these libraries, programmers must carefully adhere to the guidelines described in the documentation without much, if any, help from the compiler. As with the record abstraction, compilers often do not have enough information to safely optimize or check for errors in the use of an abstraction.

1.2. Motivation

General-purpose programming languages provide facilities for programmers to define abstractions to model their problem domain. There is often a tension between

abstraction and efficiency, because the programmer must rely on the compiler to optimize the implementation of an abstraction. Since language designers and compiler writers cannot foresee all possible abstractions, they can only implement a general substrate on which domain-specific abstractions can grow and evolve. Consequently, abstractions often expose a low-level interface so programmers can hand-optimize the implementation for their particular use. This dissertation demonstrates how a compiler can be dynamically extended by programmers to support new abstractions, thereby allowing a high level of optimization and error-checking without resorting to hand-optimization using a separate low-level interface.

1.2.1. The Problem

Programming involves designing and using abstractions to solve a problem. An abstraction is a reusable artifact that can be applied to many specific problem instances. Abstractions are constructed from functions, classes, libraries and components. Examples include a parameterized sort function, a polymorphic streaming I/O class hierarchy, a list processing module and a graphics subsystem. Each is a reusable artifact that vastly simplifies the tedium of writing software. Each provides an interface that balances the competing demands for simplicity, power and flexibility. In addition, each interface has a particular semantics described in the documentation. Programmers are expected to understand the semantics well enough to apply abstractions correctly and efficiently. In a sense, programmers are “hand compiling”

their high-level abstractions into low-level code without much support from the compiler.

Programmers would like the simplest interface possible without sacrificing flexibility or efficiency. As Butler Lampson observes[82], “The main reason interfaces are difficult to design is that each interface is a small programming language: it defines a set of objects and the operations that can be used to manipulate the objects. Concrete syntax is not an issue, but every other aspect of programming language design is present.” Of course, a programming language designer uses a compiler to bridge the gap between simplicity and efficiency, but a module designer does not have this luxury. All too often, module designers opt to provide a fast, low-level interface at the cost of simplicity, thus requiring programmers to play the role of the compiler.

When a consensus on the general importance of an abstraction is reached, it may be added to a programming language as additional syntax and/or compiler optimizations. The purpose of syntax is to provide a simpler and safer interface to an abstraction. In addition, this is the traditional way to capture high-level concepts and pass them to the compiler for optimization and error-checking. For example, a DO-loop in FORTRAN[2] can be trivially expanded into code that uses a GOTO statement. However, the semantics for DO-loops constrain the use of GOTO into a form that can be aggressively optimized by a FORTRAN compiler. Programmers could have used GOTO in a stylized way and the same compiler optimizations could work. The DO-loop, as additional

syntax, ensures a too-clever programmer cannot confuse the compiler's conservative optimizations. In this way, the accretion of syntax in most programming languages is a way to satisfy the growing needs of their programmer community for better optimizations and compile-time error detection.

In fact, most programming languages have built-in support for record types because they are critically important for all applications. But what about abstractions that are important to a sizeable minority of a programming language's community? Since language designers and compiler writers can not support all[103], or even most, abstractions required by a heterogeneous user community, most abstractions are implemented as modules. For those abstractions that require compiler support, e.g. scientific programming[56], the language may splinter into dialects or new languages may emerge to better serve those communities. This is only a temporary solution because programmers will inevitably want support for even more features. The problem is that languages and compilers evolve too slowly to meet new demands by programmers because changes must be made in the language's syntax and/or compiler to support new features.

1.2.2. A Solution

Many compilers for functional languages are implemented as transformations from a high-level language to a low-level language through a series of successively simpler

intermediate languages[69, 122, 99]. These transformations can often be implemented as small, modular passes, rather than as large passes that perform many transformations at once. In addition, many compilers use transformations which only require local, rather than global, information to execute properly. This style of compiler construction has proven effective for many functional programming languages.

The solution proposed here is an extension of existing compilers for functional languages. A module interface can be considered a definition of a domain-specific language extension to the base language[44]. A program that uses a module can be viewed, instead, as a program written in an extension of the base language. Whereas conventional compilers, in the presence of separate compilation, ignore calls to modules, an extensible compiler can transform the program from its extended form into the base language. Since abstractions are often layered on top of other abstractions, an extensible compiler can transform the program through all the intermediate "languages" to reach the base language[80]. Obviously, a compiler writer cannot foresee all possible abstractions that will be added to a language. Instead, the compiler can be designed to allow module writers to extend the compiler with domain-specific transformations. Since compiler transformations can be mutually beneficial, they are all applied together on the source code. The compiler's task is to load, order and apply these transformations until a fixed point has been reached.

More broadly, this dissertation argues that compilers should be extensible, reusable systems, rather than inflexible, black box transformation tools. Today, only compiler writers can modify or extend a compiler, preventing it from adapting as users' needs change. By recognizing that module designers are, in fact, designing a domain-specific programming language, compilers can and should be designed to be extended with new transformations for these modules. The compiler extensions are only applied when a programmer uses those abstractions by linking with that module. Give programmers the power to evolve the compiler independently and it will quickly adapt to new domains in surprising ways.

For example, a module designer can implement an abstract record type for Scheme and add specific transformation rules to optimize their use. First, it can inline first and rest when they are not used as first-class values. Second, it can tag pair? as a pure expression so common subexpression elimination can eliminate redundant uses. Finally, it can replace uses of vector-ref with a low-level vector accessor that does not perform bounds checking. These simple rules are similar to those found in compilers that optimize record types automatically. With an extensible compiler, however, the library designer can add these rules directly.

1.3. Implementation

The implementation is a source-to-source compiler, called Sausage, for a small functional language. The compiler is implemented in Scheme, which is an ideal laboratory for an extensible compiler. There is a long and rich history of performing aggressive program transformations with lexical macros[28], a powerful preprocessor which allows arbitrary program transformations. The syntax of the language is simply an s-expression, which can be destructured and created like lists. Therefore, the issue of creating new syntactic abstractions is largely solved with the macro system. Once all the macros are expanded, all that is left is a small and simple, yet powerful, higher-order programming language. Being small is important, since it reduces the number of patterns to match against. Simplicity reduces the number of program analyses required to perform meaningful transformations. Scheme stands alone as a popular language with these vital attributes.

The source language to the compiler is Core Scheme with all the macros fully expanded. The intermediate language is the untyped λ-calculus represented in A-normal forms (ANF)[49], a direct-style intermediate program representation. Transformation rules are written to match and expand user-defined syntax extensions. These same rules can be used to compute information about the subtrees in a form. To drive the compilation process, every module imported by the root module may optionally supply a set of transformation rules. These transformation rules are sorted

and applied to the root module until no more rules match. The result is still Scheme, but the abstractions have been optimized.

1.3.1. Transformation

The heart of the compiler is the transformation engine. The transformation rules are written in a modular style, each doing a small local transformation. The rules are matched on the intermediate language, including the core syntactic terms that are typically off-limits in conventional macro systems. The body of the rule is arbitrary Scheme, which allows programmers to perform complex transformations. The result is a syntax tree constructed using a small set of data constructors. The tree is updated and the transformation engine then looks for another rule to apply.

The transformation rules are separated from the tree-walking code. A rule might indicate which direction it wishes to be applied, whether bottom-up or top-down, for example. This allows all the rules that run in a particular direction to be executed together. Some rules, like constant folding, can run in any direction. The rules are often mutually beneficial, i.e. constant folding may compute a constant that triggers another rule. Therefore, all the rules are run repeatedly by walking up and down the tree until a fixed point is finally reached.

1.3.2. Analysis

An abstraction designer can also write new program analyses. Using the same transformation engine and rules infrastructure, designers can derive auxiliary information about the program and pass it up or down the syntax tree. This is very similar to attribute grammars, a method for computing information during the parsing phase of a compiler. The analysis can easily compute synthesized, or bottom-up, attributes. For inherited, or top-down, attributes the system keeps track of intraprocedural control flow. This flow is represented as a path from the matched term back up to the enclosing lambda form. Programmers can query this path to discover the context in which a transformation rule is to be fired. This is vital for transformations like common subexpression elimination. Using these tools, designers can write more powerful domain-specific and general-purpose analyses.

1.3.3. Ordering

When compiling a module, all transformation rule sets are loaded for each imported module. This chain of module imports is followed recursively, forming an acyclic directed graph. To choose which rule sets to apply, the modules are topologically sorted. Within each module, the rules are applied in the order in which they are loaded. Each module's transformation rule set is applied until fixed point, and then the next module's rule set is applied. When none of the rule sets apply to the program, the

compiler is finished. In this way, the compiler supports separate compilation by dynamically creating a domain-specific compiler for each module.

All the transformation rules, including the compiler's rules, must run in a single phase. A rule may transform the tree into a form that allows another rule to execute. The compiler phases within most compilers are ordered by hand to maximize the efficient interaction of mutually beneficial transformations. Constant folding, propagation and dead-code elimination are often run together because each exposes additional opportunities for another to run. Since the Sausage compiler dynamically loads many transformation rules, it cannot know the most efficient ordering ahead of time. Instead, it runs them all together to make sure each has an opportunity to match against the latest tree form.

1.4. Limitations

The system presented here is a prototype and, therefore, fails to provide some features that a more practical system must offer. A more complete explanation of the shortcomings and possible solutions will appear in the appropriate sections in the dissertation. To preempt some obvious concerns, the following briefly describes extensions that could resolve some important issues. Notably, a powerful static type system can improve many aspects of this implementation. However, a static type

system is orthogonal to this implementation. It can be added in various forms without affecting the main thrust of this dissertation.

1.4.1. Global Analysis

There is ample evidence that data flow analysis for higher-order languages can drive more powerful transformations[111]. Unfortunately, most of these frameworks require a global analysis of the program. Since the transformation rules are intended to be local and incremental, it would be awkward to support any form of global analysis that runs concurrently with the transformation rules. A transformation rule could easily cause dataflow changes to ripple throughout the program. An incremental dataflow analysis framework would not need to process the entire program whenever a transformation rule makes a small change. In fact, a demand driven analysis could delay computing dataflow facts until they are needed. Nevertheless, global dataflow analyses can have undesirable interactions with a transformation-based compiler.

1.4.2. Safety

When adding transformation rules, there should be some assurance that the syntax forms matched on and generated by the module designer are correct. Though this system relies on Scheme's latent type system to detect errors at runtime, an implementation that uses a static type system could ensure that the syntax forms are correct at compile-time. This would guarantee that the system would always generate a

correct syntax tree, though the contents of the tree might still be incorrect. For example, a statically typed tree structure would still allow one to replace the integer expression (+ 1 2) with the string constant "a", since it is a valid syntax tree. Nevertheless, it is a good start.

To provide a stronger guarantee of safety, the intermediate language itself can be statically typed[93]. By carefully checking that every replacement tree has the same type as the original tree, a programmer can get a stronger safety check. Such a system would reject replacing (+ 1 2) with "a", but 100 would be a type-safe, though incorrect, replacement. In fact, since the interface to an abstraction would itself be statically typed, it can help ensure that the replacement tree is correct according to the typing rules defined for the interface. Unfortunately, this could only work for a strongly-typed language, not a latently-typed language like Scheme.

1.4.3. Correctness

Though there has been much research aimed at ensuring the correctness of optimizers, there have been few realistic implementations. An approach that blends well with this implementation is to write a formal specification of the module interface to ensure that the replacement form has the same behavior as the input form. For example, a module for a stack data structure can prove that when a push is followed by a pop in the input program, both the push and pop can be removed. A system could statically check the

pattern and replacement tree in a transformation rule to make they have the same behavior[85]. This provides a much stronger check that the transformation rules are correct than just type safety alone.

1.5. Conclusion

Programming languages can be small and simple; in fact, the λ-calculus is all one really needs. The gratuitous syntax that clutters most programming languages is intended to provide language and/or compiler support for a privileged set of abstractions. Since languages and compilers evolve slowly, programmers construct their abstractions from the features available in the language. Most languages provide a module system where programmers can define an abstract interface to a body of reusable code. Lisp and Scheme programmers have successfully used macros for decades to add syntactic abstractions to the language. More recently, the profusion of domain specific languages[13] demonstrates a growing interest in creating simpler interfaces to complex abstractions. However, these programmer-defined abstractions do not get the same compiler support for error detection and optimization provided for built-in abstractions. The Sausage compiler aims to rectify this.

The Sausage compiler extends two similar ideas in compiler design. It extends the idea of compilation by local program transformations by recognizing that abstractions are, in fact, an extension of the base programming language. Like many functional language

compilers, it needs to transform this extended language into the core language. To do so, new transformation rules need to be added to the compiler. Since macros are program transformation systems, the same principle can be used to add transformation rules to the compiler. The result is a compiler that can be extended by programmers with new transformations and analyses to perform static error detection or compile-time optimizations[43].

The ultimate goal is to make programming much easier. A compiler makes programming easier by providing simple abstractions and translating them into efficient concrete implementations. If a compiler can be extended to handle user-defined abstractions, then those abstractions can be made much simpler, too. The piles of programming tutorials in bookstores explain how to use libraries efficiently; in other words, they teach programmers how to optimize by hand. With the Sausage compiler, many of those specific optimization rules can be embedded in the compiler. This should simplify abstractions, which should simplify programming.

Chapter 2. Prior Work

Programmers often want to exert more influence over their languages and compilers. There are many points in the typical compiler pipeline, from parsing to code generation, where programmers would like more control. The goal is not merely to generate more efficient programs, but also to simplify programming and check for additional errors at compile-time. Since languages and compilers evolve slowly, there should be some way for programmers to mold their tools to fit their domains. Extensible languages and compilers are one way to put more control in programmers' hands.

Extensibility requires language and compiler designers to make trade-offs between safety, correctness, and power. A safe extension is one that cannot cause the compiler to fail or crash. However, an extension that is safe is not necessarily correct. A system that ensures correctness must check that the transformations generate an equivalent, legal program. Short of this lofty standard, a system can check that some properties are correct. Power, the last quality, implies programmers should be able to perform arbitrarily complex transformations. However, that makes it difficult to ensure safety and correctness. These three qualities must be balanced to provide a useful compiler extension mechanism.

This chapter is not intended to be an exhaustive survey of every known compiler extension technique. Instead, it serves to illustrate a variety approaches and their limitations. The categories below are somewhat misleading, however, since there is a great deal of overlap between different systems. A system in one category may be capable of performing the task of a system in another category. The category names are a reflection of different communities with different goals inventing different, incompatible jargon. Nevertheless, the categories and examples illustrate different kinds of extension mechanisms.

2.1. Parameterization

Many compilers and/or programming languages are designed to accept hints from the programmer to guide a compiler's operations. A *parameterized* compiler pass is one which can be guided by the programmer, usually by extra information embedded in the program's source code. This extra information can range from a simple lexical token to a complex declarative specification language. However, the annotations do not allow the full expressive power of a general purpose programming language. The annotations are constrained to work within the limits of the compiler pass. Programming language researchers have made great strides in inventing increasingly powerful annotation specifications and flexible compiler passes.

Whether a parameter is a compiler extension or a part of a language is a contentious point. This dissertation takes the rather extreme view that anything beyond Core Scheme is a compiler extension. Since the computation itself can be expressed in a very minimal programming language, anything more is either syntactic sugar, a compiler/runtime directive or a software engineering safety feature. For example, C#'s[3] `virtual` method modifier tells the compiler whether it is safe to perform an early binding optimization, whereas the `public` and `private` modifiers control the visibility of methods outside the class without necessarily affecting the compiler. Adding syntactic constructs to a language is often the easiest way to pass extra information into a compiler in a portable way.

2.1.1. Annotations

The OpenMP[4] standard for FORTRAN and C++[119] includes a set of annotations for shared memory parallel programming. In FORTRAN the annotations are specified by stylized comments in the code, whereas in C++ it uses the `#pragma` directive. For example, the `parallel` directive triggers the creation of a group of threads to execute a body of code in parallel. This directive also includes many clauses that control variable allocation and access across threads, i.e. `shared` and `private` variable lists. The programmer must carefully annotate the code to trigger and control the compiler's parallel code transformation phase. If the annotations are incorrect, the generated

code will be incorrect. The OpenMP annotations allow programmers to generate more efficient code than might be possible from an automatic parallelizing compiler.

The Broadway compiler[59, 58] for the C programming language allows library writers to annotate their library interfaces to drive more aggressive compiler optimizations. The annotations provide detailed, low-level information for specific optimization passes. For example, a library designer can summarize the potential operations on data structures, e.g. whether they are accessed, modified, or copied. The Broadway compiler does not statically ensure the annotations are correct; in fact, incorrect annotations can cause the compiler to generate incorrect code. Without this interprocedural information, a compiler must conservatively assume that any operation could take place.

With Broadway's advanced annotations, a library designer can specify their own abstract dataflow lattice, where the library procedures serve as transition rules from one state to another in the lattice. This information can be used to replace a procedure call with code specialized for that context. It is a declarative annotation language for building a simple custom compiler pass. However, the code specializations cannot perform more complex transformations on a program, nor can it generate specialized code on-the-fly. Nevertheless, with these library annotations the Broadway compiler can generate scientific programs nearly as efficient as hand-optimized code.

2.1.2. Static Type Inference

A static type system is one where the types at every program point are known at compile-time. Whereas many programming languages, like Java[54] or C++, require programmers to annotate source code with type declarations, languages with type inference systems[90, 34] can derive the correct types for most program points at compile-time. However, there are times when the type inference algorithm fails to determine the correct type at a program point and a type annotation is required from the programmer. This annotation allows the type inference algorithm to overcome ambiguities in the type system and statically type check an entire program.

An advanced type system can be viewed as an extensible program verification tool[101]. The type inference algorithm itself is implemented in the compiler since it is bound intimately to the language syntax. Programmers cannot change the inference rules; they can only provide annotations to help the inference system get past ambiguities. However, polymorphic type systems provide a flexible, declarative specification language which can describe complex type transformations. Within the limits of its specification language, the type checking system is programmable.

Whereas static type checking ensures that each program point has a single type, typestate checking[118] is a dataflow analysis that ensures that all possible states of a variable conform to a user's specification. The canonical example is tracking whether a

file is in the open or closed state, thus statically ensuring that all read and write operations are performed when the file is open. A programmer annotates his program with typestate specifications that describe the behavior of his data types. The compiler performs a dataflow analysis on the program using the programmer's specifications. If the specification is violated, it raises a compiler warning. Typestate checking is a programmable dataflow analysis that is proving increasingly useful for ensuring the correctness of user-defined program constraints.

2.1.3. SPECIFICATIONS

The Speckle compiler[126, 127] merges a programming language with a specification language so a compiler can perform more aggressive optimizations on programs. The compiler uses the specifications to enhance standard compiler optimizations like loop hoisting or common subexpression elimination. In addition, the programmer can provide a general procedure interface and a suite of specialized implementations, and the compiler can use the specification to select the best implementation for that procedure. This system is limited by the flexibility and expressiveness of the specification language, which drives the compiler optimizations.

2.2. Syntax Rewriting

A popular technique for extending a language is by transforming the syntax of the source code before it is sent to the compiler, typically by a preprocessor. For example, a C preprocessor[71] performs a lexical search and replace of tokens in the program, i.e. replace the token FOO with the number 1. Lisp[114] and Scheme provide an exceptionally powerful macro system, which allows programmers to build new syntactic abstractions with ease. Syntax rewriting systems are more useful than custom language parsers because the syntactic abstractions are composable. Separately written syntax rewriting rules can be combined at compile-time. By contrast, it is difficult to merge different parsers with different syntax extensions into a single parser.

The Lisp macro system is implemented as a preprocessor that repeatedly matches and replaces terms in the source code until a fixed point is reached, and then forwards the result to the main compiler. The macros match on the symbol in the function application position and capture the enclosing s-expression. The body of a macro is written in Lisp, which gives programmers enormous power to perform arbitrary computations on the s-expression. The replacement s-expression is inserted back into the program and the process continues. Different macros for different syntactic abstractions from different modules can be merged and applied simultaneously at compile-time.

Since the syntax of Lisp is spare, macros are used extensively to solve the issue of syntactic abstractions[55]. For example, in other languages list comprehensions must be added to the compiler's language parser; however, Lisp and Scheme programmers can implement a more powerful and extensible comprehension package with the clever use of macros[41]. Macros can also be used to provide convenient syntax for tree regular expressions[11], state machines, parsers, pattern matchers, record types, etc. It is often said the first thing a Lisp programmer does is use macros to implement a domain-specific language and compiler in which to write his program. In fact, most of the syntactic forms in Scheme, a variant of Lisp, are implemented with macros.

Unfortunately, there are a few limitations with macro systems. Since the macros are applied by a preprocessor, it cannot take advantage of the compiler's internal transformations. For example, a macro can be written to compute a factorial when given a constant. However, if the constant is just out of reach then the macro cannot compute the answer. In Figure 2-1 the macro can not know that x is 10 without the compiler's constant propagation pass.

```
(let ((x 10))
     (factorial x))
```

Figure 2-1. The value of x is a constant, but beyond the scope of factorial

Another problem is that macros cannot match against the special forms in the language (define, lambda) nor can they match special cases of existing macros (cond, and). This makes it difficult to write new analyses and more complex transformations. Next, a macro must appear at the head of an s-expression; therefore, it cannot be used as a first-class value, which severely limits the use of macros. Finally, a macro cannot discover the context in which it is called, since it only has information about its enclosing s-expression. Despite these limitations, macros have been used in astoundingly clever and powerful ways.

2.3. Program Specialization

Program specialization is a broad term that includes nearly all compiler optimizations. It could be defined more narrowly to mean optimizing a general-purpose algorithm or library for a specific instance of a problem. For example, a Fast Fourier Transform (FFT) algorithm specialized for a specific input size usually runs much faster than the general algorithm. This input size might be discovered at run-time, triggering a run-time compiler to specialize for that value. In addition, the input size does not need to be a single constant value. Instead, if the value is usually 20, but not always, then the compiler can generate a specialized version wrapped in a guard that checks if the size is 20; otherwise, it uses the general algorithm. The FFT has been specialized for a specific

size, but can still execute other sizes as well. A program specializer could automatically discover and optimize an FFT by discovering how it will be used by a program.

2.3.1. TEMPLATES

Templates in C++ were originally designed to implement generic types by performing compile-time macro expansion. In this respect, it is quite similar to syntax rewriting systems, particularly Scheme's define-syntax form. A peculiar quirk in the implementation of C++ templates, however, opened the door to more powerful code transformations[130]. The templates can perform integer operations at compile-time, and the value of enumerations can be propagated throughout the body of a template. In Figure 2-2, the factorial of 5 is computed at compile-time by expanding the templates until the base case Factorial<1> is reached. The Factorial template recursively expands itself, decrementing N at each step. The templates behave like a rule-based rewrite system or a less powerful Lisp macro processor.

```
template<int N>
struct Factorial {
    enum { value = N * Factorial<N-1>::value };
};

template<>
struct Factorial<1> {
    enum { value = 1 };
};

// Example use
const int fact5 = Factorial<5>::value;
```

Figure 2-2. C++ template specialization to compute factorials at compile-time

This technique is called template meta-programming, where specialized routines are generated at compile-time. A sub-genre is called active libraries[131], where template meta-programming is used to customize the use of a library. The Blitz++ library[128] for scientific computing uses template meta-programs to unroll loops, inline code and specialize algorithms at compile-time, thus approaching the performance of optimized FORTRAN code. The programmer can use a much simpler API without paying any performance penalty because the templates act like a domain-specific compiler for that specific library. An active library includes a set of template meta-programs that specialize a programmer's use of those libraries. With active libraries, the C++ language and compiler can be extended by library writers to support new domains.

Though this technique has been used to generate efficient, specialized scientific routines, it is still less powerful than Lisp macros and suffers from many of the same limitations. The use of Factorial must explicitly use a constant integer, not a variable,

even if the compiler can prove the variable is a constant. That is, one might wish to call Factorial<x>, where x is a local variable. If the compiler determines that x is a constant value, then it should compute the Factorial at compile-time; otherwise, compute the factorial at runtime. In addition, C++ templates can only operate over integer constants at compile-time, whereas Lisp can do nearly anything. Nevertheless, performing some computations at compile-time opens the door to powerful domain-specific extensions using C++ templates.

2.3.2. Partial Evaluation

In principle, partial evaluation is an aggressive application of constant propagation[63]. Given a function, the inputs can be divided into two sets: those known at compile-time (static) and those provided at run-time (dynamic). A partial evaluator, often labeled *mix*, merges the function with the static inputs to yield a specialized function that only takes the dynamic inputs. Ideally, the partial evaluator would apply a wide range of compiler optimization techniques to eliminate the execution overhead associated with the static inputs, resulting in a more efficient specialized function. But compiler optimizations can only go so far, since they must be safe and conservative, especially in the presence of separate compilation.

The concept of partial evaluation can be applied to program compilation in surprising ways[51], as described in Figure 2-3. The first Futamura projection states that an

interpreter partially evaluated with a program's source code will yield an executable program, e.g. a Python[105] interpreter and a specific Python program can be merged to create a native executable for that program. The second Futamura projection combines a partial evaluator with an interpreter to yield a compiler. Finally, the third Futamura projection yields a compiler-generator, which can take an interpreter and produce a compiler for that language. For the latter two projections, it is implied that the partial evaluator tool is written in the same programming language for which it partially evaluates.

Program Evaluation:

interpreter [source, input] = output
compiler [source] = program
program [static, dynamic] = output

Partial Evaluation:

mix [program, static] = specialized
specialized [dynamic] = output

First Futamura Projection:

mix [interpreter, source] = program

Second Futamura Projection:

mix [mix, interpreter] = compiler

Third Futamura Projection:

mix [mix, mix] = compiler-generator

Figure 2-3. Partial evaluation and Futamura projections

While there are examples of fully automatic partial evaluation systems[123, 15], they cannot yet perform these optimization miracles. Without hints from programmers, partial evaluators tend to over- or under-specialize programs. Instead, the general idea of partial evaluation has been applied to a number of domains, i.e. operating systems, graphics, networking, etc. These systems add domain-specific optimizations tailored to eliminate many compile-time computations in combination with conventional optimizations like constant folding and loop unrolling. These systems are really domain-specific compilers because they cannot optimize a program for a different domain. Once again, a program that spans multiple domains cannot be fully optimized by such limited compilers.

2.3.3. Multi-Stage Programming

A multi-stage program[120] allows programmers to generate code at any stage of a program. The usual C compiler is a two-stage language: compile-time and run-time. C++ is a three-stage language: template expansion stage, compile- and run-time. Multi-stage programming generalizes this staging concept so programmers can generate and execute code at any stage. For example, a program can generate code on-the-fly during the run-time stage. Of course, this can already be done with Lisp given macros and eval (run-time evaluation of program code)[23]. Multi-stage programming adds type safe program generation, as opposed to Lisp's latent type checking. Like Lisp macros, multi-

stage programming gives programmers the ability to specialize programs and implement partial evaluation by hand.

Multi-stage languages are an implementation of the multi-stage programming concept. Languages like MetaOCaml and MetaML[52, 121] extend the base language with a strongly-typed variant of Lisp's quasiquote and eval feature. Quasiquote is a convenient syntax for constructing expressions with "holes", which will be filled in with a value at a later stage. For example, Figure 2-4 shows a staged version of the exponentiation function in MetaML to compute x to the nth power, specialized for n equal to 10. The extra staging annotations are used to specify how to generate code when given a specific value for n. The angle brackets are equivalent to Lisp's quasiquote, which indicates that the code will be generated in the next stage. The tilde is like Lisp's unquote, which are holes in the code templates that will be evaluated later. Finally, the run command is similar to Lisp's eval function, which executes the function and produces a new, specialized function by concatenating the code templates together. The specialized code for exp'(10,<x>) will be generated and assigned to exp10. The novelty of MetaML and other multi-stage languages is that the compiler can show at compile-time that any program generated by these staging annotations is type correct.

```
fun sqr' x = <let val y = ~x in y * y end>.
fun exp' (n, x) = if n = 0 then <1.0>
                  else if even n then sqr' (exp' (n div 2, x))
                       else <~x * ~(exp' (n – 1, x))>.
val exp10 = run <fn x => ~(exp' (10, <x>))>.
```

Figure 2-4. Specialized version of exp to compute x^{10}

While multi-stage languages lack the syntax manipulation advantage of Lisp macros, it provides a similar program generation capability, much like C++ templates. It still suffers from many of the same problems as Lisp macros. Like Figure 2-1, the programmer must explicitly know the value of n and call run to generate code. What if the compiler could have deduced the value of n during program analysis? An automatic partial evaluator would have specialized exp, but MetaML cannot do so automatically.

2.4. Domain Specific Languages

A domain-specific language (DSL) is one that more closely models the problem domain. A DSL can be implemented as a specialized programming language with its own compiler. For example, the ANTLR parser generator[102] has a declarative syntax that simplifies the task of building LL-based recursive decent parsers, and a special compiler translates this specification into executable code. There are a few problems with this approach[39]. First, building a compiler is a complex task that may not be worth the effort for domains with a small user community. Second, DSLs are often limited languages that lack the power of a general-purpose language. Third, many software

systems use many different abstractions at once. It can be difficult to compose many different DSLs to write a single procedure (e.g. pattern matching and concurrency). Finally, programmers are often reluctant to learn a DSL and incorporate its compiler into their build process. One can sidestep the compiler issue by enclosing a DSL in strings within a general-purpose language, e.g. SQL queries, regular expressions and XPATH. Overall, DSLs are suitable for certain specialized domains, but do not scale for large, multi-paradigm systems.

Embedded domain-specific languages (EDSL)[62] solve many of the problems described above. Rather than build a different language and compiler, the DSL is implemented within a general-purpose *host* language. The ideal syntax for an EDSL can be approximated with macros or with overloaded operators, data constructors and/or higher-order functions. Usually, the EDSL creates a data structure at runtime representing the computation, similar to an abstract syntax tree (AST). The AST is executed at runtime by an interpreter, which the EDSL designer must write in the host language. However, both tasks are performed at runtime, not compile-time. Therefore, an EDSL lacks compile-time error detection (though a static type system can help) and optimization. Furthermore, the EDSL program is interpreted, which can be slow. For example, if a LL parser were implemented as an EDSL, it would generate and interpret a recursive decent parser at runtime. This would be unacceptably inefficient for most

applications. EDSLs are more convenient than stand-alone DSLs, but at the price of static error detection, compile-time optimization, and runtime performance.

2.5. Reflective Systems

Reflection is the ability to examine and often change the behavior of a system from within. In the context of this dissertation, the system can be the runtime environment or the compiler. Programmers have access to an extensive API with which they can modify the default behavior of the system. Generally, reflection requires that the programmer have a deep understanding of the underlying system to modify it correctly.

The Metaobject Protocol (MOP)[72] is a reflection API first implemented for the Common Lisp Object System (CLOS). It provides a set of classes which represent each of the elements in CLOS, like methods, classes, and slots. By overriding the default implementation in the MOP, a programmer can modify the runtime behavior of an object oriented program. The MOP is generally used to customize the CLOS runtime to better suit a particular problem domain. Unfortunately, the various implementations of the MOP expose a complex reflection API which, if used improperly, can result in strange runtime behaviors or outright errors. Nonetheless, the MOP has proven useful and resilient, such that many other languages have also implemented a MOP.

The OpenJIT project[88] implemented a MOP for a Java JIT compiler that allows programmers to extend the virtual machine with new compiler passes. This MOP is really a library toolkit for manipulating a Java AST and writing program analyzers. These passes are complex, manipulating very low-level compiler implementation details. A programmer can annotate a Java class to load and execute a specific compiler pass, which may apply domain-specific optimizations to that class. However, each custom compiler pass runs without coordination with the others. For example, if a programmer wishes to write a transformation pass that requires constant propagation and folding, he will need to rewrite those passes even though they already exist in the JIT. The OpenJIT compiler has been used to dynamically extend the Java runtime to support distributed shared memory systems.

Aspect-Oriented Programming (AOP)[74, 73] is a reflective system that allows programmers to intercept and modify the control flow of a program. These points in the control flow include method calls, field access, exception handling and ever more as AOP systems become more powerful. There are many different implementations of the AOP concept for different languages with varying power[1]. The goal of AOP is to enable a controlled form of global program transformation, either statically or dynamically, on the control flow of a program. It is limited by the set of program elements that can be matched and the degree to which elements can be transformed. For example, a programmer can intercept a set of methods and replace them with

another, or execute a method before or after the matching method. A programmer can not write a program analysis to find methods worth intercepting. Instead, he must explicitly name the methods he is interested in transforming. Despite its limitations, AOP advocates claim these extensions simplify programming by separating code that might usually be duplicated throughout a program.

2.6. Extensible Compilation

An extensible compiler is one which allows programmers to add new or modify existing compiler passes. To be sure, the previous systems allow different degrees of extensibility. In an extensible compiler, however, extensions are more closely integrated with the compiler. An extension can make use of information discovered by the compiler, like types, use-def chains, side-effects, etc. Also, the transformations performed by different compiler passes are often mutually beneficial. For example, constant propagation and constant folding perform transformations that help each other find new optimization opportunities. In an extensible compiler, it is useful to run an extension alongside other passes so they can perform better together. An extensible compiler allows programmers to extend the capabilities of a compiler to suit new domains.

Several systems have attempted to augment macro systems to provide more control over transformations. McMicMac[81], an extension of Scheme's macro system, allows

macros to manipulate the object code directly and gather information bottom-up from the syntax tree. The RScheme system[78] moves Scheme's macro-expansion phase into the compiler. This allows macros to use reflection to access the compiler's environment at every program point; for example, to gather type information. Programmers can write custom analysis routines to gather more information from the subtrees. Neither system interleaves macro transformations with other compiler passes, however. Consequently, both suffer from the same limitation illustrated in Figure 2-1.

Catacomb[115, 116] is an extensible compiler for the C language which goes one step beyond conventional macro processors. Given a set of user-defined code templates written in an extended dialect of C, the compiler analyzes and optimizes the code and templates together. Catacomb solves the issue plaguing Lisp and other macro systems by merging the macro-expansion phase into the compiler. The code templates do not allow complex transformations beyond a single statement or expression, nor does it allow custom analysis routines. Catacomb ensures that code templates will run whenever another optimization creates a new optimization opportunity. This system has been used to explore a variety of parallel array assignment strategies for high-performance computing by extending the compiler with new optimizations[117].

Similarly, the Glasgow Haskell compiler (GHC) supports custom rewriting rules over programs[100]. Programmers can write typed rewrite rules using a simple extension of the Haskell language. These rules are loaded and executed along with all the built-in

rewrite rules in GHC; therefore, the custom rules will fire whenever another rule creates an opportunity. The rewrite rules do not allow for any analysis or custom code generation; it simply matches the pattern on the right-hand side and replaces it with the code template on the left-hand side. It makes no attempt to ensure the rewrite is correct or more efficient. Despite its simplicity, it has been used to implement list and tree deforestation, among other optimizations.

The MAGIK system[44] is an extensible C compiler that dynamically loads and applies user-written compiler passes on the source code. Each extension has full access to the abstract syntax tree and compilation environment. The compiler passes can perform any arbitrary transformation or analysis. Like GHC's rewrite rules, these custom passes are executed along with the compiler's built-in passes until a fixed point is reached. Though the MAGIK system is certainly powerful, it is a compiler specific extension mechanism. That is, a library writer cannot write domain-specific transformations and analyses in a portable way. For example, it is not possible to write portable active libraries because the extensions only work with the MAGIK compiler's AST and environments.

2.7. Sausage

The Sausage compiler aims to provide maximum power at the expense of compile-time safety and correctness checking. It aims to match the power of the MAGIK system

without requiring programmers to wed themselves to a particular compiler implementation. Instead, the transformation rules in Sausage are similar to Lisp macros. The rules manipulate s-expressions rather than a compiler-specific AST. Sausage avoids providing a reflection API because, again, the extensions would be tied to a particular compiler. All the extensions in Sausage are composable and are executed together. Like macros, different programmers can write different extensions and know that the compiler will coordinate their execution. The goal of Sausage is to write all optimizations as extensions; that is, Sausage is a meta-compiler with no built-in compiler optimizations at all.

The Sausage extension framework can be simplified to suit the needs of different user communities. Compiler writers will need the full power provided by Sausage to write complex optimizations. Library writers may prefer to sacrifice some power for additional safety and correctness checks. This can easily be done writing meta-extensions, i.e. parameterized Sausage extensions, which expose a simpler, safer extension framework to a programmer. Library writers developing core libraries, e.g. .NET's System or Java's java.* namespace, may want more power than another programmer developing a less critical library. Programmers may not wish to write extensions at all. Instead, they may prefer to write provably safe and correct annotations to guide a meta-extension. This can also be done with Sausage. The point is

that a powerful system can always be scaled back to provide a safer and easier to use extension interface for different user needs.

The Sausage system should be powerful enough to overlap with most of the systems described above. For example, an extensible type system can be implemented by a set of analysis rules that transform the type attributes for a program. AOP can be implemented by a meta-extension that transforms a set of named methods or slots. EDSLs can be analyzed and optimized at compile-time. Active libraries can be easily implemented with transformation rules that mimic templates. By providing a transformation system over s-expressions that is embedded in the compiler, programmers can write useful extensions to the libraries, language, and compiler to better suit their problem domain.

2.8. Conclusion

The systems described here attempt to give programmers a bit more control over their language and compiler. The Sausage compiler begins with this point-of-view. Programmers should use a meta-compiler and meta-language from which they can implement domain-specific programming tools, languages and optimizations. Of course, most programmers will not invent their own DSLs or active libraries; instead, they will use those written by experts in their domain. By giving programmers the ability to evolve the language and compiler to suit their needs, they no longer need to

wait for compiler release cycles or language standards bodies to finally resolve their issues. The Sausage compiler makes everyone a language designer and compiler writer.

Chapter 3. Language

The Sausage meta-compilation system can operate over a variety of programming languages. It simply organizes and applies a collection of compiler extensions to an abstract syntax tree. The semantics of the language are captured by the extensions themselves, not by Sausage. It is up to the author of the extensions not to violate the semantics of the language. Sausage merely provides the infrastructure for building a meta-compilation system.

The current implementation compiles a subset of the Scheme programming language[68]. Scheme was chosen for a few reasons. First, the language is very small and simple. Second, Scheme programmers are already familiar with writing program transformations on s-expressions using macros. Third, a latently typed language avoids some static typing issues that may complicate the compiler. Fourth, a large set of programming language features can be simulated in Scheme, including object-oriented programming, backtracking, and pattern matching. Finally, Sausage is itself written in Scheme and, therefore, can be used to optimize itself. Though Scheme is an ideal language for the prototype Sausage compiler, the techniques used here can be applied to many different languages.

3.1. Source Language

The current implementation of Sausage uses as its source language a higher-order call-by-value lambda calculus extended with mutation and recursion, as shown in Figure 3-1. It is commonly referred to as Core Scheme, a minimal functional language that can serve as the kernel of the Scheme language. The language is small, simple and complete.

3.1.1. CORE SCHEME

P	::=	(define x T)
T	::=	(T T_1 ... T_n)
	|	(if T_1 T_2 T_3)
	|	(let ((x T_1)) T_2)
	|	(set! x T)
	|	(letrec ((x_1 L_1) ... (x_n L_n)) T_{n+1})
	|	L | x | c
L	::=	(lambda (x_1 ... x_n) T)
x	∈	Variables
c	∈	Constants

Figure 3-1. Definition of Core Scheme

The components of the language follow the standard definitions from Scheme. A define form binds a location in the environment named by the variable x to the value of the term T. The *lambda* abstraction binds a list of parameters to the scope of the body term T. Function application assumes the first argument will reduce to a *lambda* abstraction, and then it computes and binds each argument with the corresponding parameter. These two terms define the simple untyped lambda calculus.

The next two terms are convenient programming notations. The *if* term computes the first argument and, if it is the boolean constant #f (false), executes the third argument. Otherwise, it executes the second argument. The *let* term is a convenient notation for a lambda that is executed immediately. The variable x is bound to the result of T_1, and then it executes the body of the form. Both of these terms can be derived from the simple lambda calculus, but the extra syntax makes some patterns easier to encode.

The final two terms add practical programming features. The *set!* form overwrites a location in the store named by the variable x with the result of T. This means that every location, or variable, is implicitly a reference that can be modified by the programmer. Finally, the *letrec* term is a convenient syntactic form for writing mutually recursive definitions. It takes a list of new locations and binds each with a *lambda* form L, which can refer back to any variable names in the definition list. When all the locations are bound to a value, it executes the body of the term. Though this can be derived from *let* and *set!*, the *letrec* syntax makes it easier to detect recursive definitions.

The *letrec* form described here is stricter than the standard definition of *letrec* in Scheme, which, just like *let*, allows any value on the right hand side. A Scheme programmer can still use the standard Scheme *letrec*, but a preprocessor should lift out all terms that are not *lambda*s into a *let* form. This will leave only those unassigned variables bound to *lambda* expressions that reference another variable within the *letrec*. This simplifies the *letrec* expression so that it only handles mutually recursive function definitions[136].

3.1.2. PRIMITIVES

There are no built-in primitives defined for this language. Instead, the goal is to implement all the standard primitive operations expected of a language as modules with compiler extensions. For example, the comparison operations on numbers (<, >, =) can be implemented by a module. The associated compiler extension can perform constant folding and range checking, just as an optimizing compiler would do. There are no primitive operations and, consequently, no built-in optimizations in the compiler.

Admittedly, the parser for the source language must be extended to read a wider range of constant types. Because the language defines the *if* form, it must include the boolean value #f. The parser also reads the standard Scheme constant types: symbols, strings, characters, integers, reals, and complex numbers. Though Sausage does not implement

an extensible parser, the Polyglot compiler[95] demonstrates its feasibility with PPG[22]. Since Scheme has a simple syntax, one need only modify the lexer to include new constant types.

3.1.3. MODULES

The source language does not explicitly define a module system. Instead, it assumes that all names are made distinct by a preprocessor. For example, the `foldl` function imported from a list library might be renamed to `list::foldl`. Issues concerning scoping, naming, linking, and versioning of modules are beyond the scope of this work. Indeed, these issues are orthogonal to the optimizer at the heart of Sausage.

The compiler does, however, rely on the module dependencies to establish a linear ordering for compiler extensions. The module dependencies must form an acyclic graph; therefore, the present system does not allow recursive module definitions[33], since this would create a cycle in the dependency graph. The final node in the graph is the language definition itself. This is explicitly declared by some Scheme systems and implied by others. Figure 3-2 illustrates a simple dependency graph, where the pattern matching and datatype libraries require Scheme.

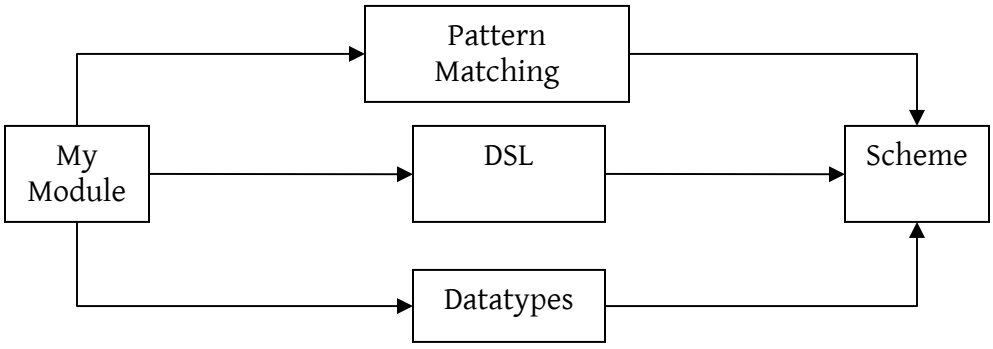

Figure 3-2. Module dependencies

There are times when it is appropriate to *not* import the base language definition, as shown by *My Module*. By importing from a domain-specific "language" implemented as a library, a programmer does not have access to the entire Scheme language. Instead, he is limited to the interface described by the DSL. A program written with the narrow syntax and semantics of a DSL are often easier to write and optimize. In this example, a DSL compiler extension associated with that library will be applied before the Scheme compiler.

3.2. Macro Preprocessor

Though the Core Scheme language is spare, most conventional programming language syntax can be added as derived forms using a macro system. The macro system is used to make programming more convenient by, for example, providing syntax for loops or

conditional expressions (e.g. a case statement). For the Core Scheme language, Sausage provides both Lisp-style macros and Scheme's define-syntax form[28]. Since these forms are expanded into Core Scheme before it is compiled by Sausage, any macro system, or lack thereof, is fine.

Though macros are intended for local syntax transformations, Lisp-style macros can perform arbitrary computations. The persistent success of Lisp and Scheme is due, in part, to macros. It allows programmers to quickly implement embedded DSLs. Macros offer the ease-of-use of a domain-specific programming notation; however, they generally cannot detect compile-time errors nor improve program performance.

Macros cannot perform global transformations on a program. This remains primarily the task of the compiler. However, the compiler does not have any semantic information about the modules. Typically, compilers optimize primitives using built-in transformation rules, while everything else, including semantically rich module interfaces, are compiled dumbly using the conservative transformation rules for the Core Scheme language. Sausage aims to bridge this gap by giving programmers more power beyond macros to optimize module interfaces.

3.3. Intermediate Language

While the Core Scheme source language is simple, it is not quite simple enough for writing compiler extensions. It allows terms to appear anywhere and everywhere. For example, an argument to a function application can be another complex let binding. It is difficult to match and transform code patterns at this level. Therefore, the Sausage compiler exposes a simpler intermediate language based on the A-normal forms (ANF)[49, 106]. This is a popular direct-style intermediate language, which is often used instead of one written in continuation-passing style (CPS). Though a CPS intermediate language makes control flow easier to follow[107], it can be more complex to write compiler extensions using CPS forms. On the other hand, ANF is a stylized way of writing programs in Core Scheme. Anyone familiar with Core Scheme will easily grasp ANF.

3.3.1. ANF

P	::=	(define x M)
M	::=	(let ((x N)) M)
	|	(letrec ((x_1 L_1) ... (x_n L_n)) M)
	|	N
N	::=	(V V_1 ... V_n)
	|	(if V M_1 M_2)
	|	(set! x V)
	|	L | V
L	::=	(lambda (x_1 ... x_n) M)
V	::=	x | c
x	∈	Variables
c	∈	Constants

Figure 3-3. Definition of ANF

The syntax of the ANF intermediate language, shown in Figure 3-3, is similar to Core Scheme. The difference is that terms are now divided into three syntactic classes to simplify some patterns. A *let* or *letrec* form is not allowed on the right-hand side of another *let* form. The conditional in an *if* term must be a constant or variable. All the terms in a function application must be constants or variables. Finally, all variables have been renamed by the α-conversion rule for lambda calculus. Otherwise, the language has the same semantics as Core Scheme.

The grammar presented here deviates a bit from the language defined by Sabry. First, Sabry's ANF language includes function application with primitives, but this ANF grammar only includes the *set!* primitive (assignment). Second, the *if* expression here is

also allowed on the right-hand side of a *let* and *letrec* binding form, which is actually how Sabry implements his ANF grammar. Figure 3-4 shows a transformation from Core Scheme to ANF where the *if* expression is lifted out of the *let* and the body of the *let* expression is duplicated for each arm of the conditional. With this grammar, the code can be left as it is on the left, thus avoiding the unnecessary code explosion. Finally, the *letrec* form is a derived form that makes it easier to detect recursive definitions. Figure 3-5 demonstrates how *letrec* can be derived from ANF; therefore, the semantics are still the same as Sabry's ANF.

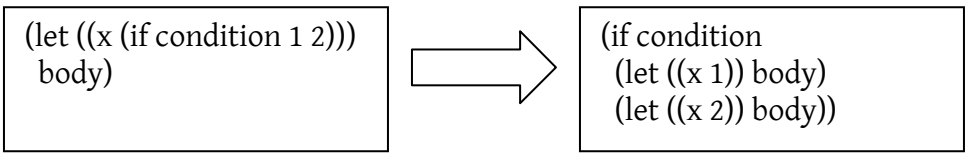

Figure 3-4. Code explosion when *if* expression is normalized.

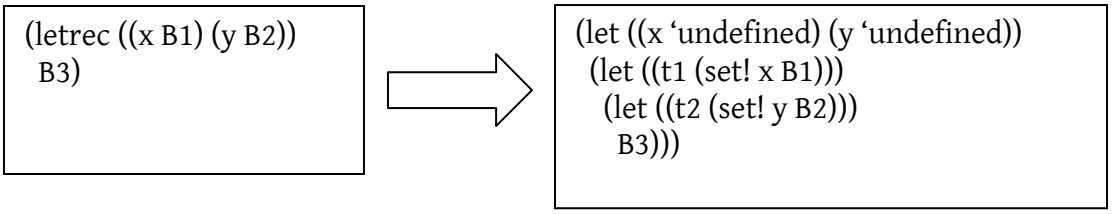

Figure 3-5. letrec is a derived form

3.3.2. A-REDUCTIONS

Core Scheme can be transformed to ANF by the A-reduction rules, shown in Figure 3-6. These rules merge code segments across let and letrec forms, and lifts all reducible expressions out of function applications and assigns them to temporary variables. This means ANF programs correspond to static-single assignment (SSA) form[70, 9], which is a widely used intermediate form for optimizing compiler. It simplifies the intermediate language and makes many transformations and analyses easier to perform[104].

The A-reductions perform a straight-forward translation of the source terms. The rules are described in a CPS style in Figure 3-6, where the normalizing function Θ takes an expression to reduce and a continuation functionκ. The left hand side of the rules matches on Core Scheme and transform it into ANF according to the reduction rules. The rule for *define* does not take a continuation argument; instead, it reduces the term T and passes in an identity function as the continuation. This means the result of the reduction will be inserted in place of T. The reduction rules for constants and variables simply forward the value into the continuation function. Similarly, the rule for *lambda* forwards itself into the continuation, but it also normalizes the body of the lambda. The rule for function application recursively normalizes each term T, then gathers all the results into a new function application and binds it to a new variable t, which is then

forwarded to the continuation. Since this is written in CPS style, the recursion is written inside-out as nested continuations. The same rule is applied explicitly to *set!*.

$\Theta[(\text{define x T})] \rightarrow (\text{define x } \Theta[T, \lambda M.M])$
$\Theta[(\text{let } ((x\ T_1))\ T_2), \kappa] \rightarrow \Theta[\ T_1, \lambda N.(\text{let } ((x\ N))\ \Theta[T_2, \kappa])]$
$\Theta[(\text{letrec } ((x_1\ L_1) \ldots (x_n\ L_n))\ T), \kappa] \rightarrow$
$\Theta[L_1, \lambda L_1. \ldots \Theta[L_n, \lambda L_n.(\text{letrec } ((x_1\ L_1) \ldots (x_n\ L_n))\ \Theta[T, \kappa])] \ldots]$
$\Theta[(\text{if } T_1\ T_2\ T_3), \kappa] \rightarrow \Theta[T_1, \lambda V.\kappa(\text{if V } \Theta[T_1, \lambda M_1.M_1]\ \Theta[T_2, \lambda M_2.M_2])]$
$\Theta[(T\ T_1 \ldots T_n), \kappa] \rightarrow \Theta[T, \lambda V.\Theta[T_1, \lambda V_1. \ldots \Theta[T_n, \lambda V_n.(\text{let } ((t\ (V\ V_1 \ldots V_n))) \kappa[t])] \ldots]]$
$\Theta[(\text{set! x T}), \kappa] \rightarrow \Theta[T, \lambda V.\kappa[(\text{set! x V})]]$
$\Theta[(\text{lambda } (x_1 \ldots x_n)\ T), \kappa] \rightarrow \kappa[(\text{lambda } (x_1 \ldots x_n)\ \Theta[T, \lambda M.M]]$
$\Theta[x, \kappa] \rightarrow \kappa[x]$
$\Theta[c, \kappa] \rightarrow \kappa[c]$

Figure 3-6. A-reduction rules

The rule for *if* normalizes the conditional term first, and then the continuation inserts the new *if* term and normalizes both branches. Both continuations in the branches are the identity function because they are not lifted out above the *if* term. The rule for *let* normalizes the term T_1 and passes the *let* form into the continuation. The continuation will then normalize T_2 and forward the result to the continuation. Finally, the *letrec* rule recursively normalizes all the *lambda* terms in the definition list, and rebuilds the *letrec* form by normalizing the body term T. All the rules are applied until the Core Scheme program is transformed into ANF.

The result of the A-reduction rules is equivalent to that of a CPS based compiler. A CPS compiler will convert Core Scheme to a CPS representation, and then perform β and η

reductions on the intermediate form. A β-reduction converts (λx.x)n to n by performing function application. A η-reduction converts (λx.f x) to f, which means it removes lambda abstractions where possible. By converting it from CPS back into a direct-style lambda calculus, one gets ANF. The A-reductions, therefore, produce a normalized direct-style program without the intermediate conversion to CPS form.

3.4. Alternatives

It is important to note that the compilation scheme described in this dissertation is not limited to Core Scheme or ANF. In fact, a wide variety of languages would be suitable for this approach. The choice of language determines the degree of analysis required to safely perform a transformation. For example, in Core Scheme it would be unsafe to reorder two function calls without ensuring that neither have side effects. In Java an interprocedural analysis is required to propagate the types from other classes to make type-safe transformations. Some languages may allow aggressive transformations, while others can only safely enable a small set of simpler transformations.

Though the languages described here are latently typed, it can easily support static typing. The semantic rule for *if* must be changed to only allow boolean types in the conditional expression. Then Core Scheme is similar to core Standard ML[110] and can be checked by the same typing rules, after which the types can be erased to yield Core Scheme. Of course, Scheme's base language libraries would need to be rewritten to be

type safe according to these type rules. This is easy to do because Sausage does not hard-code any primitives or optimization rules that are specific to Scheme. Since Sausage does not enforce a source language, it can support statically typed languages, too.

3.4.1. Typed Intermediate Language

A typed intermediate language (TIL) is a statically typed language that does not erase the types before compilation[122]. Instead, the types are preserved throughout the compilation phase. The type checker can ensure that each compiler pass is correct with respect to the type system. This is particularly useful for a meta-compilation system like Sausage, where the compiler passes are written by module writers. When a module writer implements a module and compiler extension, the compiler can check at compile-time that the compiler extension does not violate the type rules of the language or the module. This extra checking can help make meta-compilation more practical for real-world use.

3.4.2. GHC Core

The Glasgow Haskell Compiler (GHC)[64] for the Haskell language applies source-to-source transformations on a simple, core intermediate language. This intermediate language is similar to Core Scheme, though it also adds some terms for manipulating

types. Haskell is a lazy functional language, which means the program can be aggressively reordered because there is no fixed order of evaluation, i.e. evaluation occurs on demand. In addition, side effects are described explicitly by monads, which mean local transformations do not need to preface every transformation step with an effects analysis. The Haskell core language would be an ideal intermediate language for a meta-compilation system.

3.5. Conclusion

This dissertation will use the ANF intermediate language throughout to implement compiler extensions. This does not mean, however, that a module writer must necessarily write extensions using ANF. A specialized DSL can be layered atop the Sausage system which automatically translates rules written in Core Scheme to ANF. The benefit of such a translation is that it does not expose the internal structure of the compiler. If a compiler wishes to use a CPS intermediate language instead, the DSL can translate rules from Core Scheme to the CPS forms. This makes the compiler extensions portable across Scheme implementations, much like macros.

The Core Scheme and the ANF intermediate languages are flexible and powerful enough to implement a variety of programming language features. In fact, Scheme is an ideal laboratory for meta-compilation because it is so flexible. Everything is implemented by libraries; consequently, an optimizing compiler for Scheme is

relatively simple. On the other hand, since everything is implemented by libraries, Scheme compilers normally cannot optimize programs aggressively, especially in the presence of separate compilation. Sausage's meta-compilation, however, allows programmers to use Core Scheme and ANF intermediate languages to easily write complex compiler extensions to optimize library calls, be they standard Scheme procedures or domain-specific library procedures.

Chapter 4. Meta-Compilation

Compiler research has been a fruitful discipline for decades. Researchers have concentrated on individual optimization techniques and the intermediate language format; however, the overall organization of compilers has remained the same. There is only one static compiler that operates over all input sources. Programs are written in the primary language paradigm (object-oriented, functional, etc.), while other domains are crudely implemented by libraries and programming patterns. For example, many constraint programming libraries for Java[125] require programmers to tediously construct a constraint graph; indeed, the programmer manually *compiles* his problem into an abstract syntax tree. This inflexibility can be mitigated by macro preprocessors, but the main compiler does not change.

For the Sausage meta-compiler system, the motivating assumption is that a large application consists of many smaller domain-specific tasks[13]. Many of these tasks would benefit from domain-specific language syntax, error-checking and/or optimizations. The compiler vendor cannot add all requested domain-specific features into a compiler[103]; instead, the compiler needs an extension mechanism to allow others to add domain-specific knowledge. Macro systems for Lisp/Scheme have been used to add domain-specific language syntax and optimizations to a compiler via the preprocessor. In fact, aggressive macros turn out to be domain-specific compilers, often

re-implementing portions of the main compiler. The Sausage system views each source file as a potentially different domain-specific source that might benefit from a specialized domain-specific compiler.

The Sausage system provides a framework for programmers to write new domain-specific optimizations and inject them into the main compiler. For each input source, a custom compiler is automatically constructed and applied to that source. This design is actually a natural next step beyond the static, inflexible conventional compiler design. This chapter describes how the compiler pipeline is inferred and executed, but does not describe the implementation of the individual passes. The meta-compilation technique described for Sausage can be used for any programming language. It is not specific to Scheme or s-expressions or dynamic languages, though these certainly simplify the implementation. In fact, Sausage itself can operate over any language's AST because the Scheme specific knowledge is in the extensions, not the meta-compiler.

The Sausage system allows any programmer to add new extensions. In practice, however, a meta-compiler can restrict extensions to a select group and still derive practical benefits. Sausage could be used as a flexible framework for compiler researchers to mix and match compiler passes. It could be used to add specialized optimizers for a language's standard library, e.g. the base libraries provided for Java and .NET, the C Standard Library, the Scheme SRFI libraries[113], etc. It can be used by domain specialists to add optimization extensions for domain-specific libraries[66] like

networking, graphics, state-machines, pattern matching, etc. Or programmers can be trusted to use the extension mechanisms carefully and judiciously for their own code, like macros. The question of whom to allow the use of the meta-compilation features is a commercial concern, not a research matter. Sausage demonstrates that all are possible and useful.

4.1. Compilation Techniques

The basic organization of most compilers is as a pipeline of passes which analyze and transform an abstract syntax tree (AST). Most compiler research has focused on the format of the AST and the individual passes. However, relatively little work has focused on the pipeline which connects all these passes. The most significant step has been the move from conventional compilers, as described by most compiler textbooks, to the transformational compilers typically used by functional languages. In both cases, the pipeline remains rigid and inflexible. Even extensible compiler frameworks like SUIF[141] and Polyglot[95] require that the compiler's pipeline be programmatically modified and recompiled. The striking difference between conventional compilers and meta-compilers is that the pipeline is dynamically reconstructed for each invocation of the compiler, depending on the source input.

4.1.1. Conventional Compiler

Most traditional optimizing compilers have a similar structure[5]. The source code is simplified and converted into an AST by the parser. A carefully constructed pipeline of analyses and transformations optimize the AST, and finally generate some output. In fact, there are often several different types of ASTs within a compiler. For example, the various language parsers in GCC 4.x create a high-level language independent AST called GENERIC[89]. This is transformed into a family of ASTs called high-level GIMPLE, low-level GIMPLE and SSA-GIMPLE[94]. After each AST has been optimized, the tree is transformed into a very low-level AST suitable for code-generation called RTL (register transfer language). This final AST is used to generate machine code for a specific architecture. The optimizer is like a chain of smaller compilers for different ASTs (see Figure 4-1)[83]. Each *sub-compiler* reads an AST, optimizes it and produces a lower-level AST.

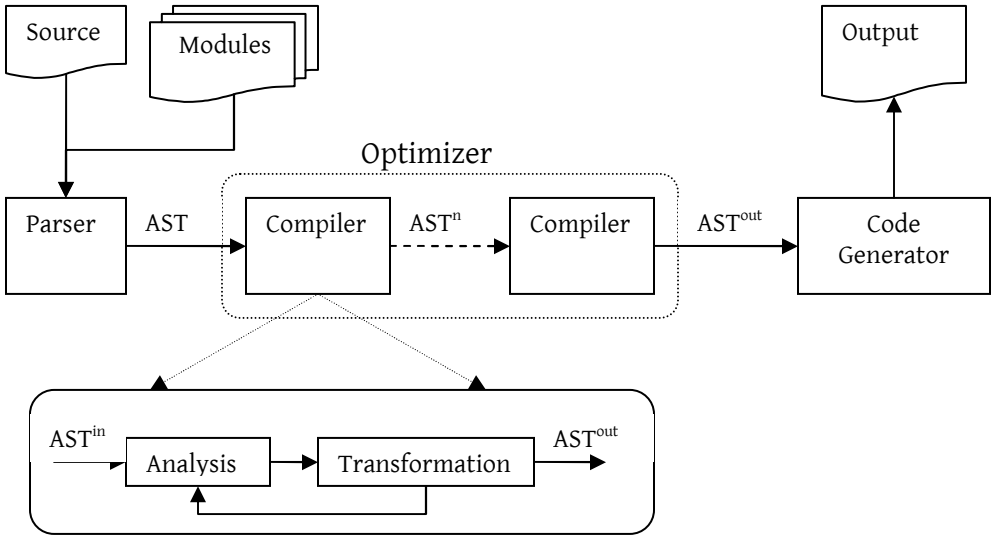

Figure 4-1. A conventional compiler

Each *sub-compiler* for an AST has a hard-coded sequence of optimization passes, where a pass may depend on transformations performed or information deduced by a previous pass. For example, a pass may require that all complex looping constructs (e.g. FOR, WHILE) be transformed into a canonical form. If a previous pass had not already done so, then this pass would not recognize many patterns in the AST. The passes are also mutually beneficial, where one pass may create an opportunity for another pass to optimize an expression, and vice versa. For example, constant folding and constant propagation are mutually beneficial[139]. Each time a constant is propagated throughout a program, it may create a new opportunity to compute a constant

expression at compile-time. Therefore, passes are often run iteratively until the AST no longer changes (fixed point) or some arbitrary cutoff has been reached.

4.1.2. TRANSFORMATIONAL COMPILER

The different ASTs in a conventional compiler represent real programming languages; they must be at least as powerful as the input language. Therefore, the optimizer can be implemented as depicted in Figure 4-2, where a series of smaller compilers operate over programming languages by giving the ASTs a concrete syntax. Just as with ASTs, a lower-level language simplifies some features found in the higher-level language, or rewrites them in a canonical form that is easier to work with. This transformation can continue all the way down to a simple RTL, which can then be printed in the appropriate output format, i.e. machine code. Basically, it duplicates a conventional compiler, but implements all the ASTs as real programming languages.

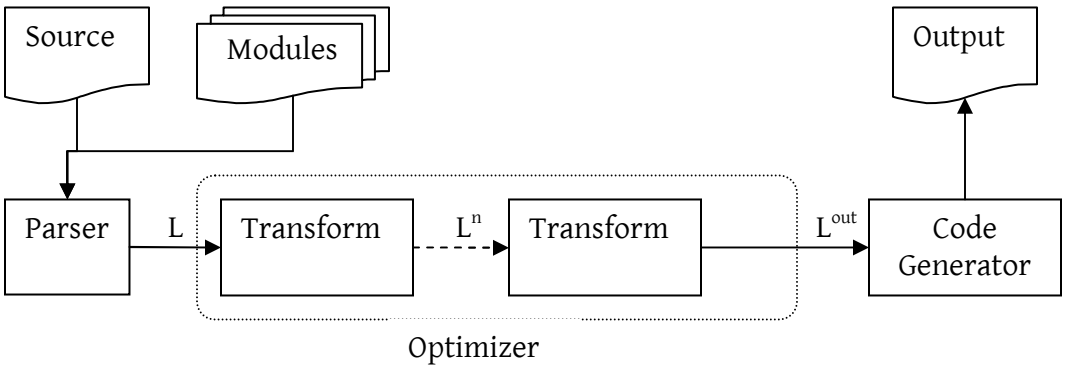

Figure 4-2. A transformational compiler

For many functional languages[10], these smaller compilers can be implemented as source-to-source transformation systems[69]. In these systems, a series of rewrite rules match and replace terms in the source program, both optimizing the code and transforming it into a lower-level language. The benefit of this technique is that the languages can be defined in terms of their denotational semantics rather than as an ad-hoc data structure. This aids in demonstrating the correctness of the languages, the transformations and, therefore, the compiler. It is also easier to add and remove rewrite rules in a transformation system, making it easier to experiment with new optimizations. Since transformation systems primarily perform local transformations, it is ideally suited for functional languages with few global side-effecting operations like assignments.

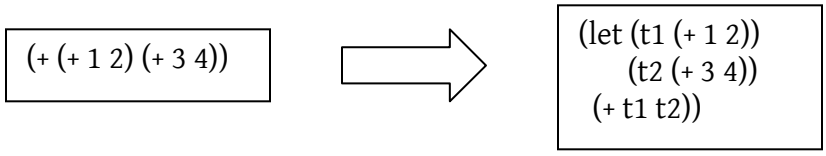

Figure 4-3. Lift complex expressions out of function applications

A few compilers use the same raw syntax to represent both the high-level and low-level languages For example, a Core Scheme program may be simplified by lifting all complex expressions out of function applications and replacing them with temporary variables, as shown in Figure 4-3. The result is a stylized Core Scheme program that is easier to manipulate, but all subsequent transformations must maintain this invariant.

That is, the language has been restricted to only accept variables as arguments to a function application. Though the languages are syntactically similar, the lower-level language is a subset of, and a valid program in, Core Scheme.

4.2. Meta-Compiler

The Sausage meta-compiler is primarily concerned with dynamically constructing the pipeline within a transformational compiler. It is motivated by the idea that a base language L extended by a module's interface is similar to a higher-level language[44]. The same idea motivating transformational compilers can be applied in this context to reduce the extended language to the base language L. The *sub-compiler* can be supplied by the author of the module, who is responsible for correctly optimizing a program extended by his module's interface. The Sausage system loads and applies each *sub-compiler* from each module referenced by the source program, as shown in Figure 4-4. The result is an optimized program written in the base language, which still may call functions in those modules' interfaces. The code generator could either produce machine code or, as in the current implementation of Sausage, an optimized Scheme program.

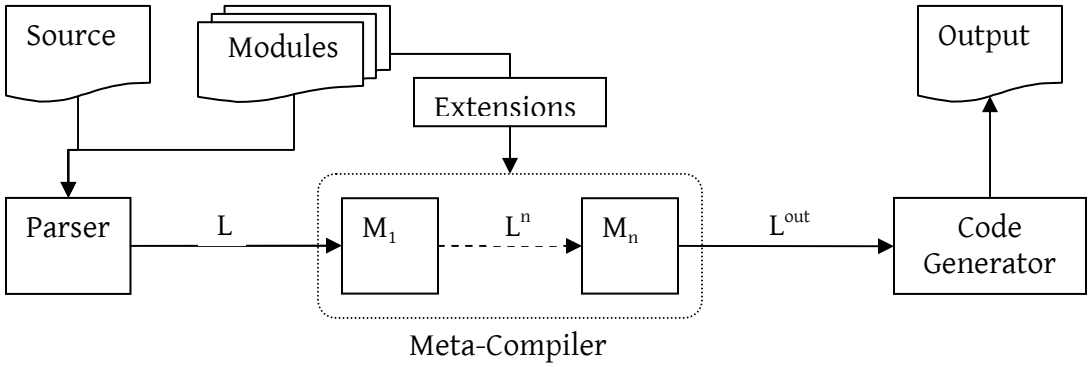

Figure 4-4. A meta-compiler

4.2.1. Interfaces As Languages

Scheme compilers already adopt the idea described here in a limited way. The Scheme language[68] is divided into two parts, a definition of the Core Scheme language and a set of standard library functions. A Scheme compiler only needs to implement the Core Scheme language to be correct, though its performance would be poor. To improve performance, most Scheme compilers have a built-in set of optimizations for the standard library functions[79]. For example, a Scheme compiler will optimize calls to arithmetic functions like +. Of course, a C compiler does this too, but the + operator is defined in the C language, not in the C standard library. This is the critical difference: Scheme compilers are optimizing calls to library functions. Since these functions are defined by the Scheme standard, compiler writers can hard-code a set of special

optimization rules into the compiler. Unfortunately, this privilege is not extended to the authors of other modules.

With Sausage, module writers have the same ability to add library specific optimizations and error checks as do compiler writers. With Scheme, there is no syntactic difference between a built-in operator and a procedure call. Therefore, a general mechanism for optimizing procedure calls is sufficient to handle any language extension (pattern matching, for example). For languages like Java, there is no way to add additional syntax to the language. Instead, all programmer extensions are modeled as method calls or, in the worst case, as code wrapped within strings (e.g. SQL, regular expressions). In this case, library writers can check for more complex errors in how the interface is used and optimize particular uses of the interface. More significantly, the library interface can be vastly simplified without sacrificing performance by adding a library-specific optimizer.

4.2.2. Compiler Construction

At a high-level, the implementation of the Sausage meta-compiler is straightforward. First, the language is parsed and represented in some simple intermediate format. Next, Sausage must dynamically construct a pipeline of *sub-compilers* to reduce a program down to a base language, which in this case is Core Scheme. The program is

transformed by this pipeline of sub-compilers iteratively until it reaches fixed point. The result is a compiled program.

To construct the pipeline, Sausage uses the source code's references to other modules to load and organize the passes. Most languages use a module system that encodes references to other required modules. C uses *#include* statements, Java uses the *import* statement, SML/NJ[91] and Scheme48 use a *structure* declaration, and SWI-Prolog[140] uses a *module* statement. Each program declares a set of required modules, and each of those refers to another set of required modules. This continues recursively to form a directed acyclic graph of module dependencies rooted at the source program and ending, either explicitly or implicitly, with the core language library. In Scheme48 it is the *scheme* structure; in Java it is the package *java.lang.**.

This directed acyclic dependency graph is linearized by performing a topological sort. Since the set of dependencies of a given module in Sausage is unordered, the topological sort could produce many valid orderings. Sausage selects an arbitrary valid ordering, which has proven sufficient thus far. Many languages load modules in a specific order. For example, Scheme48 loads the required structures in the order they are listed because this may affect top-level definitions with multiple bindings. Additional edges, or sorting constraints, can be added to the graph to force certain orderings; however, this may result in a failure to find an ordering if it inadvertently

creates a cycle in the graph. The result of this step is a valid linear order of modules beginning with the program and ending with the core language library.

Given this linear ordering, any compiler passes associated with a module are dynamically loaded into Sausage. For example, the arithmetic module may point to a specific optimization pass that handles arithmetic. Not all modules will have an optimization pass; indeed, most modules written by end-users will not. All those that do are organized into a pipeline of passes using the linear ordering inferred by the dependency graph. At this point, the generated compiler is similar to a transformational compiler.

4.2.3. Compilation

The main compiler pipeline has now been constructed and is ready to compile a module. In a conventional compiler, many passes are executed together iteratively because they are mutually beneficial. Constant folding and constant propagation are often merged together because one may create a new optimization opportunity for the other[27]. For example, the code in Figure 4-5 needs to propagate the value of x before constant folding can determine that y is 2. The entire expression can be replaced with the value 4 by executing both constant folding and propagation three times. Though the pair of optimizations could be run until fixed point, most compilers cap the number

of iterations to some arbitrary max (e.g. 10). The Sausage system emulates this behavior when executing its dynamic compiler pipeline.

```
(let* ((x 1)
       (y (+ x 1))
       (z (+ y 1)))
 (+ z 1))
```

Figure 4-5. Simple constant folding and propagation example

Sausage executes the pipeline of optimization passes in small steps. A pass makes only a small improvement to the AST and then returns to the beginning of the pipeline to start again. When a pass does not make any changes to the AST, Sausage then applies the next pass in the pipeline. When no pass in the pipeline makes any changes to the AST, it has reached fixed point and the compiler is finished. The pseudo code in Figure 4-6 illustrates the algorithm. When the loop finishes, the final tree will be assigned to the variable AST.

```
AST = program
pipeline = {p1 ... pn}
pass = 1
do
        NEWAST = apply pipeline[pass] to AST
        if (NEWAST == AST)
                pass++
        else
                pass = 1
        endif
        AST = NEWAST
until pass > n
```

Figure 4-6. Pseudo code for Sausage compilation model

The key to this algorithm is that the early passes have a higher priority than the later passes and must be offered an opportunity to optimize the new AST first. The reason for this is that a custom compiler extension may wish to extend or override the behavior of another extension. For example, a low-level extension may expand a structure accessor like (A-field struct) into a direct vector accessor like (vector-ref struct 1). However, a high-level extension may wish to first perform type-checking on the structure accessors. Therefore, any extension at any level that creates new structure accessors must return to the beginning of the pipeline to trigger the type-checking extension. When extensions manipulate the same expressions, the order in which extensions are applied becomes crucially important.

It is important to note that the current implementation of Sausage is written in a purely functional style. Therefore, checking for equality between the AST and NEWAST

is a simple pointer comparison. No compiler pass is allowed to make any side-effecting modifications to the tree. A pass that makes a small transformation must also regenerate the spine of the tree all the way back to the root[96]. Though this may seem expensive, modern copying garbage collectors for functional languages are adept at handling large amounts of garbage in the first generation[8]. In addition, some techniques for deforestation[137] may reduce the amount of garbage produced between passes. Regardless, the use of functional data structures simplifies the design of the Sausage compiler and easily enables the composition of compiler extensions.

4.2.4. Cleanup

When the main Sausage compiler has finally reached fixed point, each compiler pass is granted one last opportunity to clean up the AST. This is needed because a compiler extension may insert stubs in the AST to facilitate further optimizations. For example, a type checker may place annotations throughout the AST to cache type signatures. These annotations need to be removed from the AST before code generation. Each compiler pass may optionally register a finalizer with the Sausage system. When the main compilation loop is finished, each finalizer is executed once in the same order as the original pipeline. This is the last and only chance for a compiler extension to remove any stubs and cleanup any unoptimized code in the AST.

The final AST is then serialized into its final format. Generally, every compiler has a code generator that transforms the final AST into some other format such as a binary executable, a C program or an optimized version of the original program. Sausage does the latter by generating an optimized program in Core Scheme. This can be compiled into native code by a conventional Scheme compiler or executed by an interpreter. In fact, additional extensions could be added to transform the program into RTL just like a transformational compiler. Sausage does not support conversion into another language, i.e. Scheme to C, using the meta-compilation system. Instead, a separate code generation phase would perform this type of translation.

4.3. Properties

This implementation of Sausage aims for power over safety and provable correctness. The primary issue is that the individual transformations are allowed to do anything. They are simply transformation functions over ASTs. Therefore, it is possible for the compiler to diverge or fail to terminate. The transformation language can be constrained to help programmers write correct extensions, but this would limit the power of the system. There is no reason that different programmers could not use different DSLs for writing extensions: a powerful language for compiler writers and a simple one for the casual programmer. For now, the Sausage system suffers from the same issues as a macro system, which appears to be successful in practice.

4.3.1. Confluence

Confluence is usually an important property for a transformation system[38]. It states that a set of transformations applied in random order to an AST will always yield the same final result. Of course, confluence is not an issue for a traditional compiler because the order of transformations is fixed by the programmer. For a rewrite system, confluence is important because the runtime selects an applicable rule from a bag of valid rules with little input from the programmer. The result may be different if the runtime runs rule *A* followed by *B*, or *B* then *A*.

Sausage lies between a compiler and a rewrite system. Each library has a consistent partial ordering of extensions which an extension writer can depend on. The issue is when more than one library is referenced by a module, the different partial orderings are merged into a total ordering and there will likely be different extensions inserted into the expected ordering. For example, a library A refers to libraries X and Z. Another library B refers to W and Y. When a programmer refers to libraries A and B, the total ordering might be W, X, Y and Z. The programmer can depend on the fact that Z will always run after X, but other extensions may run between them. It is possible that running extensions X and Z will yield a different result than running W, X, Y and Z. The partial ordering is consistent, but the total ordering is derived by the runtime.

Sausage does not provide any guarantees for confluence between extensions. In practice, most extensions will only match on the public API exposed by its library. Therefore, the rules are likely to be non-overlapping, which means the rules across extensions are not unifiable[146]. In fact, there are times when confluence is not desired. Programmers may wish to override or extend the behavior of rules in other extensions. In this case, the partial ordering of extensions ensures that one rule must run before another. The DSL for transformations can be constrained to guarantee confluence, but it will significantly reduce the system's power.

4.3.2. Termination

Sausage may not terminate if a set of rules create a cycle. If a rule transforms x to y and another transforms y to x, this cycle will run forever. The simplest solution is to cap the number of transformations at some reasonably high number. Another solution is to detect cycles by periodically computing a hash for the AST to ensure the same AST does not reappear. Sausage does not statically guarantee that the collection of transformations will terminate.

A more aggressive solution is to ensure that all transformations are monotonically improving an AST[86]. That is, no rule is allowed to produce a *less optimized* AST, however optimization might be defined. For example, a constant folding rule for multiplication always rewrites "*x * y -> z*". It would be illegal, then, to write a rule that

transforms a negative constant number –N into "N * -1". Given a lattice with fixed height of incrementally optimized trees beginning with the input AST and ending in an optimized AST, no rule can go *down* the lattice; instead, all rules must move *up* the lattice to a slightly better tree. So long as all transformations are moving closer to the final result (a fixed height lattice implies a final result actually exists)[84], the system is guaranteed to terminate eventually.

In practice, this is likely to be less of an issue for Sausage. Macros in Lisp and Scheme have been used successfully by normal programmers without creating non-terminating systems. This is generally because the transformations reduce a high-level embedded DSL to a set of low-level library calls. This intuition matches the idea expressed in the previous paragraph of always transforming the tree towards the final result. Sausage should benefit from the long history of success with macro systems.

4.3.3. Complexity

At first glance, Sausage appears to be a very computationally expensive system. It iterates over the extensions in the pipeline repeatedly, only making a single transformation each time. Given a set of extensions E, each extension has a set of patterns P and transformations R. Assume the average time for a single transformation is r. In the worst case, every extension except the last will fail to transform the tree. The required running time is |E||P| + r. Now this will be repeated until the AST reaches

fixed point, which means there will be some number, t, of transformations applied to the tree. The total running time is therefore |E||P|t + tr. The |E||P| factor assumes every pattern is exhaustively applied to every node in the tree; however, this term can be substantially reduced by employing well known engineering tricks to optimize the matching engine. In that case, the running time will be dominated by the actual rewrites, not the checking of patterns that fail to match, and so compilation will be O(tr), the total cost of all tree transformations needed to produce an optimized tree. This is consistent with the runtime behavior of a traditional compiler.

4.3.4. Performance

The current implementation of Sausage has not been designed for efficiency. It uses pure functional data structures to ease development; however, this means every modification generates a new AST. In addition, the system runs on a Scheme interpreter, albeit a high-performance one. Nevertheless, compile-time has been adequate. On a 1.6GHz Pentium-M, it took 2.6 seconds to reduce a 713 line Scheme function to 460 lines by applying many of the same high-level optimizations also used by the Twobit Scheme compiler. It took 0.95 seconds to compile a comprehensive list library of 113 functions and approximately 1000 lines of code. In a more aggressive stress test of the system, it took 136 seconds to apply 10,000 transformations to the same large function, where each transformation rebuilds the spine of the tree. On many

Scheme source files, the compiler typically runs under 3 seconds despite its inefficient implementation.

With some modest design improvements and an optimizing compiler, Sausage can easily run many times faster than it does currently. A more substantial improvement would be to merge adjacent extensions into a single pass, when possible. The key is to avoid rebuilding intermediate ASTs unless they are actually modified. Other improvements can be made in how the individual extensions transform the AST. Compile time will likely be longer than a conventional compiler, but it would certainly be usable.

4.4. Related Work

Techniques similar to those described for the Sausage meta-compiler can be found in other systems. The Compilation Manager (CM) for SML/NJ is an automated build system for SML programs[18]. Rather than explicitly list the dependencies and build order for modules, as is done for the *make* or *ant* build systems, CM infers the build order from an initial list of source files[17]. It parses each source file and builds a dependency graph from the explicit imports listed in each source file. The imports are scanned recursively until all required files are found and the complete dependency graph is built. The graph is topologically sorted to get a feasible linear ordering for the compiler. The CM model is made slightly more complicated by the potential of multiple

top-level declarations of the same global binding. In addition, in SML a module can be dynamically opened within a program, thus complicating the exact dependency analysis. Both complications could occur in some Scheme systems too, but they are ignored for now by Sausage for simplicity. The point, however, is this technique for dependency analysis has been used successfully by SML/NJ and can be used by Sausage to infer the order for the compiler pipeline.

Two other compilers have developed ad-hoc methods to organize and apply extensible compiler passes or rules. The MAGIK compiler[43] solves the problem of ordering the extensions by requiring they be order independent. There are three kinds of extensions: (1) transformers, which modify the AST without global optimizations; (2) optimizers, which run until fixed point; (3) inspectors, which are transformers that run last in the pipeline. The MAGIK compiler loads all extensions specified in the source code by a special directive. It runs all the transformers once, runs the optimizers until fixed point, and finally runs all the inspectors once. Whereas most conventional compilers carefully organize the order of passes, the MAGIK compiler does not guarantee any order between passes, except for this coarse-grained ordering of extension types.

The GHC Haskell compiler has a distinctly different design. Rather than write a complete compiler pass, programmers write a set of transformation rules[100]. All these rules are loaded into the core rewrite engine within GHC and applied along with

the built-in transformation rules. Whenever a rule matches, the rewrite is applied to the AST. They implement a simple scheme to organize the rules into phases by having the programmer annotate the rule with a *phase number*. Each time the rewrite engine executes one pass over the AST, the phase number is incremented. Then all rules with the new phase number are added to the collection of transformation rules and the rewrite engine runs again until the phase number reaches some arbitrary cutoff. The order of rewrite rule application is still unordered, though a small degree of order can be forced through the phase numbers. Though this scheme is, as the designers admit, quite simplistic, it appears to work in practice.

4.5. Conclusion

The Sausage meta-compiler demonstrates a useful advance in compiler systems. For each invocation of the Sausage meta-compiler, a specialized compiler is dynamically constructed based on the source input. It uses the latent dependency information within the source code to infer the linear order of the compiler pipeline. It allows each compiler extension to make a small, incremental change to the AST and then restarts from the beginning of the pipeline again. This ensures that higher priority extensions see any new modifications to the AST before the others. When a compiler extension does not change the AST, it invokes the next extension until it reaches the end of the

pipeline. Finally, it applies each extension's finalizer to perform one last cleanup on the AST. The final AST is an optimized program.

The novelty of the Sausage meta-compilation system is that it allows a compiler to be extended and molded to fit a specific task without waiting for the next compiler release. It is impractical to expect a compiler vendor to add every feature one might want, particularly for features with a small customer base. With a meta-compiler, domain experts can encode their knowledge into a reusable library and compiler extension. Many of the techniques used now, from macros to DSLs, can be implemented more easily by a meta-compiler.

Chapter 5. Program Transformation

A conventional compiler improves a program by matching and replacing subtrees in an AST. Of course, the Sausage meta-compiler is no different. Compiler researchers have developed an array of tools and domain-specific languages (DSLs) to aid in the construction of compilers. Sausage borrows from these ideas to make it more convenient for programmers to develop optimizations. However, the differences between conventional compilers and meta-compilation influence how an individual optimization pass can be implemented.

The key difference between a conventional compiler and Sausage is in how much work an individual optimization pass performs. The former exploits every available optimization opportunity before passing the program to the next stage. Sausage, by contrast, only performs a single coherent optimization and then returns back to the beginning of the compiler pipeline. For example, a constant folding pass will only fold the first expression it finds, whereas a conventional compiler would fold every expression in the program. This difference implies that optimizations in Sausage must be *incremental.* An optimization makes a single change to a program, though that change may involve many transformations. Constant propagation, for example, replaces all uses of a single variable with a constant, which may involve many rewrites in the program. The result is a program that has been *incrementally* improved and the

compiler is restarted from the beginning of the pipeline with the modified AST. This gives every optimization pass higher up in the pipeline an opportunity to transform an expression, essentially overriding lower-level optimizations.

5.1. Prior Work

There have been many program transformation systems[12] and compiler-generators[7, 57, 31] developed over the years. Though they differ in power and ease-of-use, they are all designed to match and replace terms in some type of program representation. Most of these systems provide a convenient DSL for writing transformations, but most follow the conventional design of a compiler's pipeline as a static sequence of sub-compilers. These systems can be generally divided into two groups. The first are declarative rewrite systems[134], where the runtime applies the rewrite rules whenever it finds opportunities to transform the AST. The second are tree walking systems that are under the programmer's explicit control. These systems aim to improve programmer productivity by generating code for ASTs, pattern matching, tree walking operations and error detection. The programmer manually controls the order in which passes are applied since there is little or no support from the runtime.

The design for meta-compilation in Sausage is similar to Whirlwind[24], a runtime for applying declarative transformation rules to a program representation. Programmers write transformation and analysis rules in the Rhodium DSL[86], a declarative language

specialized for compiler optimizations which can be validated at compile-time. These can be written in separate modules and the Whirlwind system will compose these separate optimizations into a single *super-optimization*. While Sausage adopts a similar declarative rewrite system, it also maintains the ordering of these different optimizations by tracing the dependencies between libraries. Whirlwind is intended for building a single static compiler for use with all input programs; Sausage dynamically builds a specialized compiler for each input file.

The design for Sausage's transformation DSL most closely resembles the Stratego/XT program transformation suite[133, 37]. Stratego/XT offers a set of compiler tools, including parsers, code generators, and tree walking libraries, to ease the development of compilers. The Stratego language offers a convenient pattern matching syntax and a combinator library for building tree walking routines. Each optimization pass is written in Stratego and compiled into a stand-alone transformation program. A compiler, therefore, is a series of these stand-alone transformation programs connected by a Unix pipe. A parser converts a program into a text representation of its AST. Each transformation program reads the AST from standard input, applies its transformations, and then serializes the result onto standard output. The compiler pipeline is managed by a Unix script; therefore, there is no high-level organization to merge mutually beneficial transformation passes. Sausage embeds a DSL similar to Stratego into Scheme while maintaining control over how the rewrite rules will be

applied to the AST. Within each transformation rule programmers can use the DSL and arbitrary Scheme code to perform more aggressive transformations than usually allowed by other DSLs.

5.2. Pattern Matching

The main task of a compiler pass is to transform an AST into a "better" AST. To locate and replace a subtree often involves lots of tedious code to manipulate and construct ASTs. Most specialized tools for writing compiler passes include a DSL for pattern matching. Sausage, therefore, also includes an embedded DSL for pattern matching, both on the AST and the source programming language. The former is required for more precise pattern matching, whereas the latter is useful for quick and simple patterns written directly in the language's syntax. Pattern matching makes it easier to write transformations, but is not required for an extensible compiler.

5.2.1. Core Pattern Matching

Sausage provides pattern matching syntax to identify specific expressions within an AST. The pattern matching system is based on a popular macro package for Scheme[144]. The macro package matches against structures defined by its structure definition syntax. This section briefly reviews a subset of the pattern matching syntax required to understand the examples in this dissertation.

```
expr :- (match-lambda clause+)

clause :- (pat body)
       |  (pat (=> identifier) body)

pat :- identifier
    |  _
    |  ()
    |  #t
    |  #f
    |  string
    |  number
    |  character
    |  's-expression
    |  'symbol
    |  (pat+)
    |  (pat* pat …)
    |  ($ structure-name pat*)
    |  (? predicate pat*)
    |  (and pat+)
    |  (or pat+)
    |  (not pat+)
```

Figure 5-1. Pattern matching syntax

The syntax shown in Figure 5-1 is similar to pattern matchers found in functional languages like SML and Haskell. The *match-lambda* form creates a first-class function that takes only one argument: the value to match against. This value is bound within the body by a special variable named *this*. Each clause is a pattern followed by a Scheme expression to execute using the variables bound successfully by the pattern. The escape clause (=>) binds a continuation to an identifier for use within the scope of the body. If called, the current clause fails and control flow drops to the next clause, if any, in the match form. By default, if all the patterns fail to match, then the *match-lambda* form returns #f; otherwise, it returns the result of the body. If the body returns #f, then the *match-lambda* form exits without trying any other patterns.

The components of the patterns are quite simple. An identifier binds with any expression at that position, whereas the any (_) pattern matches but does not bind with the value at that position. All the standard Scheme constants will match if equal to the value found at that position. A list of patterns will match every item in the list. If the last pattern in a list is followed by three dots, the pattern is applied to every element remaining in the list. Any variables in the pattern accumulate a list of matching items. A dollar sign and structure type name is used to match a particular structure, and the remaining patterns match on the fields in the order in which they were declared. A predicate is any arbitrary function that takes one argument and returns a boolean, and any remaining patterns are applied to the same item. Finally, the boolean operators *and, or, not* apply a list of patterns to the same item and succeed if all, one or no patterns succeed, respectively. This pattern language is a useful DSL for writing clear and simple expressions for deconstructing ASTs.

5.2.2. Language Specific Pattern Matching

Since the core pattern matching syntax can sometimes be cumbersome, it is useful to offer a simpler language specific pattern matcher. In addition, a language specific pattern matcher can ensure that the patterns are valid expressions in the language. For Scheme, the patterns are similar to the syntax used for Scheme's *define-syntax* macros[77]. These patterns lack some power, but it is easy to write simpler patterns.

```
expr :- (match-scheme clause+)

clause :- (pat expr)

pat :- identifier
     |  ()
     |  #t
     |  #f
     |  string
     |  number
     |  character
     |  's-expression
     |  'symbol
     |  (pat+)
     |  (pat+ . pat)
     |  (pat* pat …)
```

Figure 5-2. Scheme-specific pattern matching

The syntax shown in Figure 5-2 is the same as the core pattern matching syntax, except the more powerful predicates have been removed. Of course, there are no structures in Scheme's syntax, only s-expressions. One difference between these patterns and *define-syntax* forms is that here the identifier in the function application position has not been declared; therefore, the identifier needs to be declared as a symbol. The example in Figure 5-3 shows that in Sausage the identifier *func* must be distinguished from pattern variables by making it a symbol; however, in Scheme it is defined by the *define-syntax* rule. Furthermore, in Sausage the identifier is not limited to the application position; it can appear anywhere, much like *syntax-id-rules* in DrScheme[48]. The expression after the pattern is, of course, any arbitrary Scheme expression. The modified expression is created via Scheme's quasiquote shorthand, which is then parsed into an internal AST representation.

```
In Scheme:

(define-syntax func
  (syntax-rules ()
    ((func x y) (func2 y x))
    ((func x y z) (func3 z y x))))

In Sausage:

(match-scheme
  ((‘func x y) `(func2 ,y ,x))
  ((‘func x y z) `(func3 ,z ,y ,x)))
```

Figure 5-3. define-syntax vs. Scheme patterns

Any language can add its own language specific pattern syntax; however, they must be converted into a function that manipulates an AST as described in the core pattern matcher. For example, a parser for Java expressions can translate a pattern such as “?x + ?y” into a matcher that destructures the Java AST’s plus structure and binds the variables *x* and *y*. For some languages, matching on the AST structures directly using the core pattern matching syntax may prove onerous and, thus, motivate someone to construct a language specific parser with more power than this Scheme parser. Since this would merely be another library and DSL layered on top of the core pattern matchers, Sausage can be used to ease the implementation effort.

5.3. Tree Traversal

The tree traversal combinator library is based on the Stratego compiler-generator suite[135]. The library consists of a few core primitives, which are used to construct an

assortment of complex tree traversal strategies. These strategies can later be combined with transformation rules to yield a rewrite function. For example, a top-down strategy applies a transformation rule to all the nodes in an AST, starting from the root and moving recursively down the tree. If the transformation rule succeeds on all nodes in the tree, it will return a new AST. Otherwise, it will return #f to indicate failure. It is absolutely important that the transformation rules do not make any side-effecting changes to the AST. Oftentimes a strategy may backup and attempt another set of transformations on the original tree; therefore, the original tree needs to be preserved. All rewrite rules in Sausage are purely functional.

5.3.1. Core Strategies

The core tree traversal primitives simply apply a strategy to the children of a node. A strategy either transforms a subtree or fails (returns #f). There are three core primitives: *one, some, all*. The *one* primitive stops iterating over the child nodes when it successfully applies the strategy to one and only one child node. The *some* primitive must successfully apply the strategy to one or more child nodes. The *all* primitive requires that the strategy succeed on every child node. When these primitives land on a leaf node (a node with no children), *all* succeeds, while *some* and *one* fail. These strategies return either a transformed subtree or fail. Note that these do not apply a strategy to the root node, only to the children.

Since these low-level primitives operate directly on the AST, they must be customized for each language. For Sausage, the language is currently Core Scheme in ANF form. However, Sausage is not limited to Scheme. To apply Sausage to a different language, one need only rewrite these primitives for that language's AST. Any language-specific implementation of these core primitives must apply a strategy to each child node. If the primitive succeeds, it should return a new node with the modified subnodes; otherwise, it should indicate failure with #f. The primitives must not directly modify the nodes, it must create a new node instead.

```
(define (one s)
  (match-lambda
   (($ Define V B a)
    (and-let* ((b1 (s B))) (make-Define V b1 a)))

   (($ Set V V2 a)
    (and-let* ((v1 (s V2))) (make-Set V v1 a)))

   (($ Lambda P* B a)
    (and-let* ((b1 (s B))) (make-Lambda P* b1 a)))

   (($ Begin B E a)
    (and-let* ((l (list E B)) (l1 (once-map s l)) (not (eqv? l l1)))
              (make-Begin (first l1) (second l1) a)))

   (($ Let V E B a)
    (and-let* ((l (list E B)) (l1 (once-map s l))
               ((not (eqv? l l1))))
              (make-Let V (first l1) (second l1) a)))

   (($ If C T E a)
    (and-let* ((l (list C T E)) (l1 (once-map s l))
               ((not (eqv? l l1))))
              (make-If (first l1) (second l1) (third l1) a)))

   (($ App F A* a)
    (and-let* ((l (cons F A*))  (l1 (once-map s l))
               ((not (eqv? l l1))))
              (make-App (car l1) (cdr l1) a)))

   ((? Var? e)        #f)
   ((? Value? e)      #f)))
```

Figure 5-4. Low-level combinator 'one'

The implementation of these primitives for ANF Scheme destructures a node and applies a strategy function to each child node. The difference between these primitives is in what each considers success. In Figure 5-4, the *one* strategy is careful to stop once a child node is successfully transformed by a strategy function. The function *once-map* applies a function to each member of a list. If the result is #f, it keeps the original list item; otherwise, it replaces the item with the new result and stops iterating down the

list. The *and-let** form binds variables to the results of expressions and ensures that no expression returns #f. It returns either a new list with one new item or the original list because the function failed on all items in the list. When this strategy meets a Var (for variable) or Value node, which has no child nodes, it simply fails.

```
(define (some s)
  (match-lambda
   (($ Define V B a)
    (and-let* ((b1 (s B))) (make-Define V b1 a)))

   (($ Set V V2 a)
    (and-let* ((v1 (s V2))) (make-Set V v1 a)))

   (($ Lambda P* B a)
    (and-let* ((b1 (s B))) (make-Lambda P* b1 a)))

   (($ Begin B E a)
    (and-let* ((l (list B E)) (l1 (or-map s l)) ((not (eqv? l l1))))
              (make-Begin (first l1) (second l1) a)))

   (($ Let V E B a)
    (and-let* ((l (list E B)) (l1 (or-map s l)) ((not (eqv? l l1))))
              (make-Let V (first l1) (second l1) a)))

   (($ If C T E a)
    (and-let* ((l (list C T E)) (l1 (or-map s l))
               ((not (eqv? l l1))))
              (make-If (first l1) (second l1) (third l1) a)))

   (($ App F A* a)
    (and-let* ((l (cons F A*)) (l1 (or-map s l)) ((not (eqv? l l1))))
              (make-App (car l1) (cdr l1) a)))

   ((? Var? e)        #f)
   ((? Value? e)      #f)))
```

Figure 5-5. Low-level combinator 'some'

The implementation of *some* and *all* operate in a similar way. In Figure 5-5, the *some* rule succeeds if a strategy is applied successfully to one or more child nodes. The function *or-map* applies a function to all items in a list. If the function returns #f on all items, it

returns the original list; otherwise, it returns a new list with the new items. The *all* strategy in Figure 5-6 only succeeds if the strategy succeeds on every child. It uses the function *and-map* to return a new list where every member has been transformed by the strategy; otherwise, it returns the original list. These rules apply a strategy to the children in the order of execution as defined by Scheme, i.e. first to the conditional of an *If* expression, then to the *then* and *else* expressions.

```
(define (all s)
  (match-lambda
   (($ Define V B a)
    (and-let* ((b1 (s B))) (make-Define V b1 a)))

   (($ Set V V2 a)
    (and-let* ((v1 (s V2))) (make-Set V v1 a)))

   (($ Lambda P* B a)
    (and-let* ((b1 (s B))) (make-Lambda P* b1 a)))

   (($ Begin B E a)
    (and-let* ((b1 (s B)) (e1 (s E))) (make-Begin b1 e1 a)))

   (($ Let V E B a)
    (and-let* ((e1 (s E)) (b1 (s B))) (make-Let V e1 b1 a)))

   (($ If C T E a)
    (and-let* ((c1 (s C)) (t1 (s T)) (e1 (s E)))
              (make-If c1 t1 e1 a)))

   (($ App F A* a)
    (and-let* ((f1 (s F)) (a1 (and-map s A*))) (make-App f1 a1 a)))

   ((? Var? e)        e)
   ((? Value? e)      e)))
```

Figure 5-6. Low-level combinator 'all'

5.3.2. HIGH-LEVEL STRATEGIES

The high-level tree traversal language combines the core strategies in different ways to produce useful tree traversal routines. That is, the high-level strategies are independent of the actual AST. In addition to the core strategies, the traversal language has a few additional features. The most important feature is recursion, without which these strategies could not move down a tree. Also, many transformation rules can be grouped together as a sequence of rules applied in order to a tree, where the result of one rule is fed into the next rule. All the rules must succeed for the group to succeed, like the *and* operation of most languages. Similarly, a group of rules can be applied to a tree such that the group succeeds if one rule succeeds, like a short-circuiting *or* operation in most languages. A higher-order strategy is one that takes one or more rules or strategies and returns a new strategy. For example, the core strategies *one, some* and *all* are higher-order strategies. These constructs, shown in Figure 5-7, are used to build high-level strategies.

```
root :- (strategy rule)
        (strategy (id …) rule)

rule :-  (rec name rule)
         (seq rule …)
         (choice rule …)
         (higher-order-strategy rule …)
         transform
```

Figure 5-7. Strategy language definition

This simple language is enough to describe a vast array of tree traversal strategies. The transform, which is the base term in the language, is a function that takes an AST and returns either a new AST or #f to indicate failure. Similarly, the strategy language produces a function that takes an AST and applies the rules using the prescribed high-level strategies, returning a new AST or #f. To easily write reusable strategies, the Scheme macro system is used to implement an embedded DSL for defining higher-order strategies, shown in Figure 5-8.

```
(define-syntax strategy
  (syntax-rules ()
    ((strategy ?rule)
     (lambda (var)
       (strategy-aux ?rule var)))

    ((strategy ?sig ?rule)
     (lambda ?sig
       (lambda (var)
       (strategy-aux ?rule var))))))

(define-syntax strategy-aux
  (syntax-rules (seq choice rec all one some)
    ((strategy-aux (seq ?rule) ?var)
     (strategy-aux ?rule ?var))

    ((strategy-aux (seq ?rule ?rules ...) ?var)
     (let ((t (strategy-aux ?rule ?var)))
       (and t (strategy-aux (seq ?rules ...) t))))

    ((strategy-aux (choice ?rule ...) ?var)
     (or (strategy-aux ?rule ?var) ...))

    ((strategy-aux (rec ?name ?rule) ?var)
     (letrec ((?name (lambda (var) (strategy-aux ?rule var))))
       (strategy-aux ?name ?var)))

    ((strategy-aux (?ho-strategy ?expr ...) ?var)
     ((?ho-strategy (lambda (var)
                      (strategy-aux ?expr var)) ...) ?var))

    ((strategy-aux ?atom ?var)
     (?atom ?var))))
```

Figure 5-8. Implementation of strategy language

Many conventional tree walking algorithms can be constructed from this language, as shown in Figure 5-9. These strategies are "higher-order" because, like a *map* function, it applies another function to each node as it traverses an AST. Programmers supply a transformation rule and allow the higher-order strategy to iterate over the tree and apply the rule. First, *fail* and *id* (for identity) represent false and true, respectively. The *try* strategy applies a transform *s* to an AST and returns a new AST or the original AST,

but it does not fail. The *repeat* strategy demonstrates recursion and higher-order strategies. It applies s to an AST and, if successful, applies the recursive strategy x to the new AST. If s fails, the *try* clause will return the previous AST. Obviously, this strategy never fails (i.e. never returns #f); instead, it will continue to apply s to the AST until it fails and then it will return the last valid AST. The *repeat1* strategy can fail if the first application of s in the sequence fails.

```
(define fail (lambda (term) #f))          ; unconditional failure
(define id (lambda (term) term))          ; unconditional success

;; simple strategies
(define try (strategy (s) (choice s id))) ; attempt s
(define repeat (strategy (s) (rec x (try (seq s x))))) ; apply s*
(define repeat1 (strategy (s) (seq s (repeat s)))) ; apply s+

;; The rule must succeed for the root and all children
(define bottomup (strategy (s) (rec x (seq (all x) s))))
(define topdown (strategy (s) (rec x (seq s (all x)))))

;; combine bottomup and topdown into one rule
(define downup (strategy (d u) (rec x (seq d (all x) u))))

;; The rule must succeed for the root or only one child
(define oncebu (strategy (s) (rec x (choice (one x) s))))
(define oncetd (strategy (s) (rec x (choice s (one x)))))

;; The rule must succeed for the root or some children.
(define somebu (strategy (s) (rec x (choice (some x) s))))
(define sometd (strategy (s) (rec x (choice s (some x)))))

;; The rule must succeed for the root or all children
(define allbu (strategy (s) (rec x (seq (all x) (try s)))))
(define alltd (strategy (s) (rec x (choice s (all x)))))

(define manybu (strategy (s)
                 (rec x (choice (seq (some x) (try s)) s))))
(define manytd (strategy (s)
                 (rec x (choice (seq s (all (try x))) (some x)))))

;; The rule is applied to the root, then to some children, and
;; finally to the root again.
(define manydownup
        (strategy (d u) (rec x (choice (seq d (all (try x)) (try u))
                                       (seq (some x) (try u))))))

(define outermost (strategy (s) (repeat (oncetd s))))
(define innermost (strategy (s) (repeat (oncebu s))))
(define reduce    (strategy (s) (repeat (manybu s))))
```

Figure 5-9. High-level strategies

The remaining rules in Figure 5-9 implement familiar tree-walking routines using the strategy language. The strategies for *bottomup* and *topdown* only succeed if s is successful on every node in the AST. The phrase *(all x)* is what triggers a recursive iteration over the tree. The strategy *downup* demonstrates a higher-order strategy that

takes more than one argument, i.e. one rule to apply on the way down the tree and another to apply on the way back up the tree. By the same token, *oncebu* and *oncetd* will traverse an AST (*bu* is bottom-up, *td* is top-down) until s succeeds once and only once. On the other hand, *somebu*, *sometd*, *allbu* and *alltd* succeed if s succeeds on the root node or on some or all of the children, respectively. For most programmers, *manybu* and *manytd* will be the most useful: the strategy succeeds if s succeeds on at least one node in the AST. Of course, *manydownup* applies d on the way down the tree and u back up, and succeeds if at least one node is transformed by d or u.

The final set of rules demonstrates that advanced strategies can be built from these high-level strategies. The strategy *outermost* applies *oncetd* repeatedly to an AST until it reaches fixed point. This means it only descends into a child node when the parent no longer changes. The converse is *innermost*, which transforms the leaf nodes until fixed point and then moves higher up into the tree. Finally, *reduce* applies a strategy repeatedly both up and down an AST until it reaches fixed point.

5.4. Complex Strategies

Since the pattern matching and strategy languages are implemented as embedded DSLs, programmers can write more complex strategies using Scheme's language features: lexical scoping, closures and continuations. In fact, any language with these advanced features can be used as the base implementation language for Sausage

instead of Scheme. The Stratego DSL[133], by contrast, implements many of these constructs directly. Though these complex strategies are only limited by programmers' imaginations, the following strategies have proven useful thus far. They are not the limit of what can be done, but merely examples of what is possible.

5.4.1. Anonymous Strategies

An anonymous strategy is one without a name, i.e. not bound to a top-level variable. This is frequently used for transformations defined within a larger rule. Since the strategies return a transformation function, it can be applied directly to an AST. For example, *((oncetd dosomething) tree)* will apply a strategy directly to a variable *tree* bound to some AST subtree. Rather than write *dosomething* elsewhere, it can be implemented anonymously as *((oncetd (match-lambda clauses+)) tree)*. So long as the signatures of the functions match what Sausage expects, the body can be any arbitrary mix of Sausage's transformation language and normal Scheme.

5.4.2. Conditional Strategies

A conditional rule is one that checks whether a strategy is successfully applied to an AST, but does not modify the tree. If the strategy fails, it returns #f; otherwise, it

returns the original tree. The code in Figure 5-10 implements *where* as a curryed[1] function that takes a strategy and a tree. Any changes made on the tree will be functional and, thus, will not affect the original tree. However, the strategy will still waste time making all those changes to the tree.

```
(define (where s)
  (lambda (tree)
    (and (s tree) tree)))

 (define throw
  (make-fluid (lambda (v)
                (error "Must call within where/cc strategy"))))

(define (return v) ((fluid throw) v)

 (define (where/cc s)
  (lambda (tree)
    (and
      (call/cc (lambda (exit)
                 (let-fluid throw exit
                            (lambda () (s tree)))))
     tree)))
```

Figure 5-10. Conditional strategies

[1] A function is curryed when it is wrapped by a new function where some of its arguments are bound to default values.

To quickly exit a strategy when a condition has been met, the *where/cc* conditional strategy installs a dynamically-scoped variable (e.g. a fluid[2]) to save the current continuation[3] exposed by *call/cc*. If a strategy is written to use *where/cc*, it can call *return* whenever a condition has been met to immediately jump back to the *call/cc* point. By using a fluid, nested conditionals will push their continuations correctly, such that any call to *return* will exit from the last *where/cc* strategy on the stack. If a programmer neglects to call *return* or the condition is never met, it behaves like the simpler *where*. This way, programmers can use the *where/cc* strategy instead of the *where* strategy in all cases. Implementation languages that do not have continuations can use the less efficient *where* strategy.

---

[2] A fluid is like a special variable in Lisp. It is a global variable that is shadowed by a *let-fluid* binding created on the call stack. All stack frames below see the new binding, but when the frame is released the previous binding is restored.

[3] A continuation represents the next step in a computation, or the current location on the stack. It can be called anytime to return to the point at which the continuation was captured. It is used here to implement non-local exit.

5.4.3. Dynamic Strategies

Oftentimes, the contents of a rule are not known until runtime. Rather than explicitly write rules to manipulate a specific variable, rules can be generated dynamically[21]. Since these rules and strategies are simply functions, it is trivial to write a rule generator. Simply wrap a rule in another function that binds the variables of interest using lexically scoped variables. The example in Figure 5-11 demonstrates a dynamic rule that searches an AST for any assignment to a specific variable v. This is used quite often to make sure transformations are aware of possible side-effects to a variable. The function *is-var?* is used as a predicate within the pattern to look for any use of *set!* on v. If it is found, it calls the *return* function to immediately exit the rule, otherwise the rule returns #f. The strategy on the next line uses *where/cc* to install the correct *return* handler and *oncetd* to traverse the tree using the rule generated by *search-set*.

```
(define (search-set v)
  (define (is-var? n) (equal? v n))
  (match-lambda
   (($ Set ($ Var (? is-var?))) (return #t))))

(where/cc (oncetd (search-set 'x)))
```

Figure 5-11. Dynamic strategy: searching for assignments

5.5. Examples

The ubiquitous compiler optimizations constant folding and propagation serve as useful examples of the Sausage meta-compilation system[97]. These optimizations are mutually beneficial: constant propagation may create an expression that can be folded, yielding another constant that can be propagated. This leads to the phase ordering problem, where any ordering that only runs the optimizations once may lead to a sub-optimal optimization. Many compilers solve this problem by applying both optimizations iteratively until fixed point or some arbitrary cutoff has been reached (i.e. loop 10 times). A more aggressive solution is to merge both optimizations into a super-analysis where both run at the same time[26]. In most such implementations the two are merged manually by the compiler writer. This technique does not work for a meta-compiler where new extensions are dynamically loaded into the compiler. The examples below demonstrate how Sausage solves this problem.

Recall that the Sausage runtime loads and applies each rule until it successfully transforms the AST only once. The runtime itself can be described as using the following strategy:

```
(repeat (choice (oncetd loaded_strategy1) ...))
```

The loaded strategies are determined by sorting the graph of library dependencies. Each is applied top-down to the AST until it succeeds only once. If it fails it moves to the next strategy; otherwise, it starts all over again from the beginning of the *choice*

term. This continues until all the strategies finally fail. Though this appears to be expensive, various techniques can be used to speed this up.

5.5.1. Constant Folding

Constant folding is an important task for any compiler and serves as an easy illustration of strategies and transformation rules. For brevity, the code in Figure 5-12 only applies to the Scheme '+' function. The function *constant-fold/+* is a transformation rule that optimizes the addition operation at compile-time. The first two rules perform the obvious simplifications for addition. The last rule matches every use of + with more than one argument. It partitions the arguments into the set of constants and variables, which are the only legal nodes within a function application in ANF form. If there is more than one constant, it sums them and replaces the expression with a new function. Otherwise, it returns #f to indicate failure. This rule does not need to use a strategy, since the runtime will implicitly load and apply this rewrite rule until it makes one successful transformation to an AST.

```
(define constant-fold/+
  (match-scheme
    ((‘+) (make-Value 0 ‘()))
    ((‘+ arg) arg)
    ((‘+ args …)
     (let-values (((constants variables) (l:partition Value? args)))
                 (and (> (length constants) 1)
                      (let ((v (apply + (map Value-data constants))))
                           (make-App f (cons (make-Value v ‘())
                                             variables)
                             '()))))))))
```

Figure 5-12. Constant folding for +[4]

Typically, constant folding rules for arithmetic would be hard-coded into the compiler. However, in Sausage these rules would exist outside the main compiler. Constant propagation rules for arithmetic operations would be associated with the library that implements those operations. When a programmer adds a reference to this library, the compiler will load this rule along with the library. With this extension mechanism, programmers can easily add constant folding optimizations for their own libraries.

5.5.2. Constant Propagation

Without constant propagation, a compiler would not be able to take advantage of the constants produced by the previous folding phase. Constant propagation copies a constant bound to a local variable to all uses of that variable[139]. Since ANF ensures

[4] Notes on code: *partition* splits a list into two lists by comparing items with the *value?* predicate. *let-values* binds multiple return values from *partition* to two variables.

that all variables have been uniquely renamed, this rule does not need to check for name collisions. However, if the variable is assigned by *set!* within the *let*'s body, this rule should not propagate the constant. Finally, if the variable is not used anywhere in the body, it is useless and the *let* binding is removed. Propagating constants into the program can expose more opportunities for the constant folding rule.

```
(define constant-propagation
  (match-lambda
   (($ Let ($ Var name) (? Value? C) B) (=> fail)
    (let* ((name? (lambda (s) (equal? name s)))
           (propagate
             (try (manytd (match-lambda
                            (($ Var (? name?)) C)
                            (($ Set ($ Var (? name?))) (fail)))))
      (propagate B)))))
```

Figure 5-13. Simple constant propagation

The implementation of this constant propagation rule in Figure 5-13 is fairly straightforward. The first pattern matches any *let* expression where a variable is bound to a constant value. It also captures a *fail* function to immediately abort this clause. Next, the dynamic rule *propagate* is created to transform the body of the *let*. Every use of the variable is replaced by the constant C found in the *let* binding. If the rule discovers an assignment to that variable, it immediately aborts the entire rule by calling *fail* and returns #f, dropping any changes made by the inner rule. If there was no use of or assignment to the variable in the body then the rule would return #f. Therefore, the rule is wrapped in a *try* strategy so it will return the original tree if *manytd* fails. This

means the variable is useless and the *let* binding should be removed. This simple constant propagation rule is installed as a core optimization in the Sausage compiler and bound to the *core scheme* library; therefore, it is among the last optimizations in the compiler pipeline because the *core scheme* library will always come last.

5.5.3. Combined Optimization

This and many other specialized constant folding transformations are associated with the libraries that implement the operation. An infinite precision arithmetic library, e.g. bignum, would have its own set of constant propagation rules. The constant propagation rule is associated with the core Sausage runtime, so it runs last in the pipeline. The pipeline for just these two rules would effectively behave like the following: *(repeat (choice (oncetd constant-fold/+) (oncetd constant-propagation)))*. It would traverse the tree top-down looking to apply either rewrite rule. If either succeeds, it transforms the expression and restarts from the top of the new AST. This continues until both rules cannot find any more optimization opportunities.

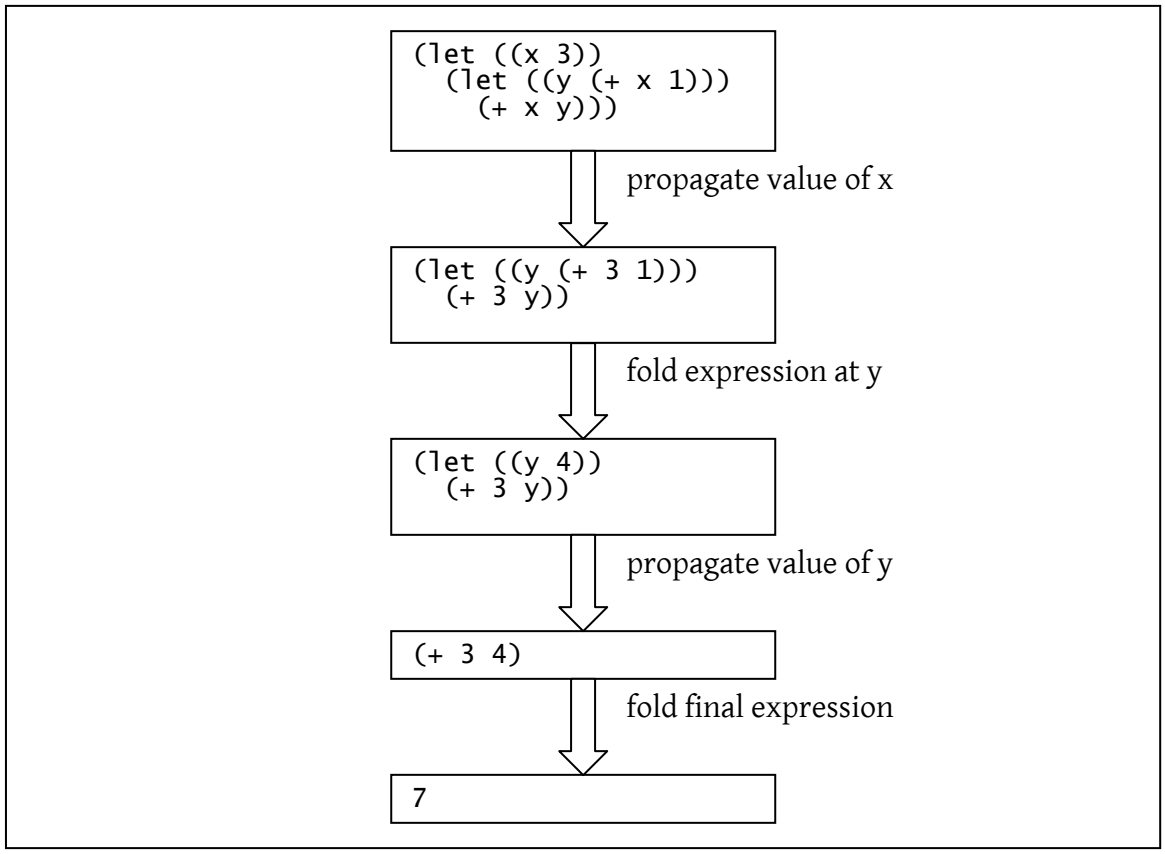

Figure 5-14. Example of combining optimizations

The example in Figure 5-14 illustrates how Sausage would run these two optimizations. Whenever a rewrite rule executes successfully, Sausage repeats the strategy from the beginning with the incrementally improved AST. Each rule is executed whenever an opportunity is created by the other rule. Sausage must restart from the beginning because some other rule may be interested in a larger chunk of the AST, not just the subtree that was modified. For example, if a rule eliminates some dead code that contained an assignment to a variable, this constant propagation rule could now run

successfully for that variable, too. These optimizations rules may be written by different programmers and dynamically loaded by Sausage, but the result is similar to a manually combined super-analysis.

5.6. Future Work

The ideas presented here for writing program transformations are very low-level. That is, there is no error checking at all, which will inevitably cause programmers to generate incorrect code. Though this is undesirable, it has not been too much of a problem for Lisp and Scheme programmers. However, the current system can easily be improved by adopting some techniques used in other systems. First, a static type system can catch a vast number of errors when programmers write patterns and create new ASTs. This means Sausage would need to be implemented in a language like SML or Haskell. Second, a typed intermediate language can ensure that the generated AST is always consistent, which is an additional static check for programmer errors. Of course, this will only work for a statically typed input language, not a latently typed language like Scheme. Third, the technique employed by Rhodium can be integrated into Sausage with a custom compiler extension. Every rule could be statically analyzed to ensure that the rules are generating an AST that has the same behavior as the original. This would require that the extension language be significantly constrained to allow for such a static analysis. Fortunately, it can be layered on top of the existing system by

implementing a Rhodium DSL extension. Sausage is designed to provide maximum power without concern for safety or correctness; however, it can be redesigned to provide safety and correctness at some cost to power.

5.7. Conclusion

The Sausage meta-compilation system offers programmers a declarative rewrite language for top-level rules and a tree walking DSL for writing complex rules. The pattern language and tree combinator libraries are, in truth, merely a more convenient and productive set of tools for programmers. In the end, they are compiled into functions that make incremental improvements in an AST. Higher-level DSLs can be implemented on top of this system to further ease development of compiler extensions. In fact, many of the tools described previously can be implemented on top of Sausage, from Lisp macros to C++ templates to Haskell rewrite rules. Therefore, many of the program transformations developed by those systems can be duplicated in Sausage. The Sausage meta-compilation system provides sufficiently low-level access to the AST that many powerful transformations can now be easily implemented.

Though Sausage is influenced by Whirlwind and Stratego/XT, the design had to be modified to allow programmers without access to a compiler's internal design to augment and extend its optimizations. After Sausage traces the dependencies between libraries used by the input program, it generates a compiler pipeline that maintains the

order, or priority, between compiler extensions. Like Whirlwind, Sausage loads and applies the rewrite rules found in these extensions; however, it only allows one transformation to take place per iteration over the AST. Like Stratego, Sausage allows programmers to use complex tree walking strategies to make more aggressive changes to an AST. However, it does not allow programmers to explicitly manage how those transformations will be applied to the AST. With just the program transformation techniques described in this chapter, a reasonably effective Scheme compiler has been implemented, including special optimizations for the Scheme standard library. As other systems have demonstrated, programmers should have the ability to transform their programs by writing little compilers. Sausage spares them the tedium of re-implementing yet another compiler by allowing them to extend an existing compiler, and gives them unrestricted power to do as they please.

Chapter 6. Program Analysis

Any sophisticated compiler must use program analyses to drive more powerful program transformations. Though purely transformational compilers work fairly well for functional languages with few side-effects, they do not perform as well on most other languages. These languages need a mechanism for gathering information scattered throughout an AST and delivering it when required by a local transformation. Sausage provides a simple and, hopefully, adequate analysis infrastructure.

The degree of analysis possible with Sausage is limited by the special requirements of an extensible meta-compiler. The intermediate representation in Sausage is an AST that closely resembles the programming language, rather than a control[6] or dataflow graph as favored by most compilers[5]. It is more difficult to efficiently analyze an AST because many valid control and data flow paths between program points are obscured. Also, since Sausage acts incrementally, it would be wasteful to perform many elaborate analyses only to trigger a single transformation and restart from the beginning of the compiler's pipeline of passes. Therefore, Sausage must perform incremental or lazy (i.e. demand-driven) analyses on an AST. The analyses in Sausage are less powerful than conventional compilers because AGs are less powerful than analyses over graphs. However, Sausage can implement any analysis performed by those compilers that use AGs.

Finally, a successful analysis pass does not cause Sausage to restart the compiler pipeline. An analysis pass does not modify the tree; it only distributes information around the tree. In keeping with the functional design of Sausage, an analysis does, in fact, produce a new tree. When an analysis is used by itself in Sausage, it must be registered as an analysis so the compiler knows to continue down the pipeline to the next pass. When an analysis is used in conjunction with a transformation, or within a transformation, then it does not need to be registered.

6.1. Attributes

A popular technique for computing information about a program is to use attribute grammars[76, 98]. An attribute grammar (AG) is a specification for annotating AST nodes with values. These systems are more commonly found in parsers, where attribute and syntax specifications are commingled. In Sausage the source code has already been parsed; therefore, the AG is defined in terms of the AST instead. Defining a set of rules for an AG is a more convenient way for programmers to analyze programs, especially when the AST so closely matches the source language. Fortunately, the attribute system can be built on top of the general program transformation system.

6.1.1. Attribute API

There is a trivial interface for manipulating attributes. The attributes are stored in a functional table that maps attribute names to any datatype[16]. Each AST node has an extra field to point to the attribute table for that node. The function "*(get-attribute node attribute)*" fetches the value of an attribute attached to a node. The function "*(put-attribute node attribute value)*" generates a copy of the node with an updated attribute table mapping the attribute to the value. After all, neither the AST nor the attribute tables should be modified directly.

6.1.2. Synthesized Attributes

Synthesized attributes propagate information bottom-up in a tree. For each node in the tree, the information is merged from the child nodes and stored in the parent node. The code in Figure 6-1 generates a function that merges information for a particular attribute for each AST node[46]. Note that the two functions *get* and *put* are curryed[5] versions of the respective attribute functions (the symbol <> leaves a hole in the argument list). Not all the nodes are represented, however. The two leaf nodes in ANF

---

[5] The operation *cut* binds some function arguments and creates a new function that expects arguments for those parameters marked by <>.

Scheme for variables and constant values are not here. Instead, another rule must be written to assign values for the base cases. A *merge* function is passed in to merge the attributes from the children. The function must be associative and commutative, since the argument order is not controlled by the programmer. The result is a tree with a synthesized attribute propagated at every node.

```
(define (synthesized-attribute merge attr)
  (lambda (tree)
      (let ((get (cut get-attribute <> attr))
            (put (cut put-attribute tree attr <>)))
        (match tree
             (($ Define V B)
              (put (merge (get V) (get B))))

             (($ Lambda P* B)
              (put (l:fold merge (get B)
                           (map (lambda (P) (get P)) P*))))

             (($ Set V V2)
              (put (merge (get V) (get V2))))

             (($ If C T E)
              (put (merge (get C) (get T) (get E))))

             (($ Begin S E)
              (put (merge (get S) (get E))))

             (($ Let V E B)
              (put (merge (get V) (get E) (get B))))

             (($ Rec D+ B)
              (let ((val (map (match-lambda
                                ((V E) (merge (get V) (get E))))
                              D+)))
                (put (merge (get B) val))))

             (($ App F A*)
              (let ((val (l:fold merge (get F)
                                 (map (lambda (A) (get A)) A*))))
                (put val)))))))
```

Figure 6-1. Driver code for synthesized attributes

This rule must be combined with another rule to supply values for the base cases. The code in Figure 6-2 is a simple example which locates the smallest constant number in an AST. The function *find-min* assigns any constant number to the *value* attribute; otherwise, it simply assigns some maximum value to the attribute. This rule must be combined with *synthesized-attribute*, which operates on the *value* attribute and uses the *min* (minimum value) function to merge the children's attribute values. The *choice* strategy means that it will apply the *find-min* function and, if successful, rewrite the node with the new attribute value; otherwise, it will use the *synthesized-attribute* function to propagate values up the other nodes. Finally, the rule is executed *bottomup* to generate the attributes from the leaves to the root of the AST.

```
(define find-min
  (match-lambda
   (($ value (? number? v)) (put-attribute this 'min-value v))
   ((or (? var?) (? value))
    (put-attribute this 'min-value max-value))))

(define find-min-analysis
  (strategy
    (bottomup (choice find-min
                      (synthesized-attribute min 'min-value)))))
```

Figure 6-2. Find minimum number in a tree

The *synthesized-attribute* function provides the default propagation behavior for bottom-up variables. Custom attribute rules are added as an additional function combined with the previous function using a transformation strategy. All synthesized attributes are, by definition, computed bottom-up. This rule will be installed into the

Sausage system as an analysis pass; however, its success will not restart the compiler pipeline. Sausage will simply move on the to the next optimization pass.

Synthesized attribute functions can also be called from within a normal transformation rule. By doing so, the synthesized attribute can be computed on demand. Synthesized attributes have the nice property that they are, in fact, merely a recursive function operating over a subtree. This can be more efficient for rules that only need one or more synthesized attributes, but no inherited attributes.

6.1.3. Inherited Attributes

Inherited attributes propagate values top-down in a tree. By default the value of an inherited attribute is simply copied to each child node. This default behavior is implemented by the fairly verbose transformation rule shown in Figure 6-3. First, the patterns match against 8 of the 10 node types defined for ANF Scheme, with the exception of the two leaf nodes for variables and constant values. After extracting the attribute value from the input tree (accessed by *this*), the rule assigns it to each child node using *put*. It then creates a fresh instance of the current node with *make-** and returns it. These rules do not make any side-effects on the tree; it generates a new tree. The function *inherited-attribute* generates a function specialized for the provided attribute. This function performs the default propagation behavior for inherited attributes.

```
(define (inherited-attribute attr)
   (lambda (tree)
      (let* ((in (get-attribute this attr))
             (put (cut put-attribute <> attr in)))
        (match tree
             (($ Define V B)
              (make-Define (put V) (put B) in))

             (($ Lambda P* B)
              (make-Lambda (map (lambda (P) (put P)) P*)
                           (put B)
                           in))

             (($ Set V V2)
              (make-Set (put V) (put V2) in))

             (($ If C T E)
              (make-If (put C) (put T) (put E) in))

             (($ Begin S E)
              (make-Begin (put S) (put E) in))

             (($ Let V E B)
              (make-Let (put V) (put E) (put B) in))

             (($ Rec D+ B)
              (let ((newDs
                   (map (match-lambda
                          ((V E) (list (put V) (put E))))
                        D+)))
                (make-Rec newDs (put B) in)))

             (($ App F A*)
              (make-App (put F)
                        (map (lambda (A) (put A)) A*)
                        in))))))
```

Figure 6-3. Driver code for inherited attributes

To demonstrate how to use inherited attributes, Figure 6-4 shows the canonical RepMin example[14], which replaces all constants in a tree with the minimum value discovered in the tree. First, the previous rule *find-min* from Figure 6-2 must be executed, which propagates the minimum number to the top of the AST. The *inherited-attribute* function used here will reuse the attribute name used by *find-min*, which means it will find the

minimum value at the root of the AST. The value is propagated *topdown* and, if it reaches a constant value node, it replaces the constant with the minimum value found in the tree.

```
(define rep-min
  (match-lambda
    (($ value (? number? v))
     (make-value (get-attribute this 'min-value)))))

(define rep-min-rule
  (strategy
    (topdown (seq (inherited-attribute 'min-value) (try rep-min)))))
```

Figure 6-4. Replace with the minimum number

Though simple, this example demonstrates that synthesized and inherited attributes can be combined to compute an ordered attribute grammar. Such an AG has no cycles in the dependency graph between inherited and synthesized attribute. Currently, one must manually order the attributes and arrange them in a strategy. In addition, inherited attributes can be computed incrementally along with a transformation rule. If the two are paired in a strategy, the inherited attribute will only be propagated down the tree until the transformation rule finally fires, perhaps with the *oncetd* rule. This can be a more efficient way to combine attributes and transformations.

6.2. Global Analysis

To facilitate interprocedural analysis and separate compilation, Sausage allows a compiler pass to persist attribute information. For example, Java class files and Haskell

interface files contain type information and, in Haskell's case, strictness information required to compile further code that depend on this module. The SML/NJ compiler implements cross-module inlining by storing a representation of functions in the object code file[19]. Many compilers use an intermediate file to store compiler-generated information such that it can be reused when compiling another file.

Sausage differs from most existing systems by allowing programmers to store arbitrary data in this intermediate store. This is similar to the approach used to implement .NET attributes and Java annotations. In both of these systems, there are a set of predefined attributes/annotations that are used by the runtime. For example, the annotation Override in Java informs the javac compiler that a method is intended to override another in the base class. The .NET attribute Serializable directs the runtime's just-in-time compiler to generate specialized methods for serializing and deserializing an object. These annotations serve as a type of compiler pragma to request additional services.

Programmers can define and use custom attributes and annotations in .NET and Java, but the compiler and runtime ignore the information. Instead, programmers must write a compiler and/or runtime services using reflection to process code that uses custom attributes. AspectJ 5[1], for example, allows programmers to use aspect-oriented programming features in Java code by using custom annotations. A special load-time service generates custom code by reading the annotations in the classes. The

goal of custom attributes and annotations is to allow programmers to extend the compiler and runtime with custom code; however, programmers cannot reuse the compilers provided by .NET and Java.

Sausage allows programmers to write extensions that generate and make use of persistent information. The implementation is simply a dictionary that is serialized to a file after compilation is complete. The current interface allows a few standard operations on dictionaries. To add, fetch and remove information, use *(put-global key value dict), (get-global key dict)* and *(remove-global key dict)*, respectively, where the key is a symbol type and dict is a dictionary name. To iterate over the dictionary, use *(fold func init dict)*, where the function *func* accepts a key and value parameter. Programmers can use

Unfortunately, the current implementation of Sausage does not support a module system for the source Scheme code. If a module refers to several libraries, then the associated dictionaries are added to a global variable. Programmers can iterate over the dictionaries in the global variable to locate information about the referenced libraries. Clearly, this interface must be improved for a more robust implementation. For now the implementation demonstrates that programmers can write useful extensions with a persistent store.

6.3. Examples

The attribute grammar system can be used to implement a variety of analyses. The most obvious are those that propagate information about variables throughout an AST. The two examples below can be used to generate code to manipulate free variables. Additional analyses are easy to write, since most of the work is done by the default routines for inherited and synthesized attributes.

6.3.1. Bound variables

It would be useful to annotate an AST with the set of local variables that have been bound at a higher lexical scope. The analysis rule shown in Figure 6-5 matches on the three AST nodes that generate new variable names: Lambda, Rec (i.e. letrec) and Let. At each node, the rules add all the names to a set bound to the attribute *bound-variables*. It relies on the *inherited-attribute* rule to copy the attribute to all the other node types.

```
(define bound-variable-analysis-helper
  (match-lambda
   (($ Lambda Ps B)
    (let ((bounded (get-attribute this 'bound-variables))
          (params (map Var-name Ps)))
      (let ((newB (put-attribute B 'bound-variables
                                 (set:union params bounded))))
      (make-Lambda Ps newB (attributes this)))))

   (($ Rec Fs B)
    (let* ((variables (map (match-lambda ((($ Var V) _) V)) Fs))
           (bounded (get-attribute this 'bound-variables))
           (newBounded (set:union variables bounded)))
      (let ((newFs (map (match-lambda
                          ((V E)
                           (list V (put-attribute E 'bound-variables
                                                  newBounded))))
                        Fs)))
      (make-Rec newFs (put-attribute B 'bound-variables
                                     newBounded)))))

   (($ Let (and ($ Var N) V) E B)
    (let ((bounded (get-attribute this 'bound-variables)))
      (let ((newB (put-attribute B 'bound-variables
                                 (set:union (set:new-set N) bounded))))
      (make-Lambda Ps newB (attributes this)))))))

(define bound-variable-analysis
  (strategy
    (topdown (choice bound-variable-analysis-helper
                     (inherited-attribute 'bound-variables)))))
```

Figure 6-5. Collect bound variables

The rule *bound-variable-analysis* rule puts the analysis together. It is a strategy that operates top-down on all the AST nodes. It applies the previous helper rule and, if successful, moves on to the other nodes. If it does not match, then it tries the *inherited-attribute* rule, which will succeed. This proceeds all the way down the tree, adding the set of all local variable names to each AST node.

6.3.2. Free Variables

It would also be useful to identify all free variables in an AST. Free variables are those not declared within a lambda, i.e. they exist outside a function definition. By reusing the information calculated by the rule *bound-variable-analysis*, finding the free variables is quite easy. The rule *free-variable-analysis* in Figure 6-6 only matches on the two leaf nodes in the AST. For every variable, it checks whether it exists in the *bound-variable* attribute. If not, it must be a free variable and it is added within a new set to the *free-variables* attribute. Constants and bound variables simply add an empty set to the attribute.

```
(define free-variables
  (match-lambda
   (($ var V)
    (let ((bvars (get-attribute this 'bound-variables)))
      (put-attribute this 'free-variables
                     (if (set:member V bvars)
                       (set:new-set V)
                       (set:new-set)))))

   ((? value?) (put-attribute this 'free-variables (set:new-set)))))

(define free-variables-analysis
  (strategy
   (bottomup (choice free-variables
                     (synthesized-attribute set:union
                                            'free-variables)))))
```

Figure 6-6. Collect free variables

The analysis rule *free-variable-analysis* propagates information bottom-up in the tree. It first tries the *free-variables* rule, which should initialize the leaf nodes in the AST. For

the remaining nodes, it will add the union of the attributes found in the child nodes. Therefore, every Lambda node will have attached to it the set of variables that escape the closure.

6.3.3. Type Inference

The attribute system can be used to infer types within a program. Since Scheme is latently typed, it is difficult to implement a strict type system for it[143]. Sausage can be used to infer types for a language with a static type system. In fact, the inference rules, or rule schemas, can be implemented by programmers as extensions to a base language. Furthermore, those inference rules can be extended by other programmers, thus implementing user-extensible type systems[109, 25].

As a simple example, consider implementing a type system for the simply typed lambda calculus extended with booleans and numbers[101]. In the lambda calculus, *let*, *letrec* and *begin* can be implemented in terms of the other language facilities, and *set!* is not allowed; therefore, the source language looks like ANF without those terms. The code in Figure 6-7 stores the relationships between types and terms in a type environment. Oftentimes, the types are still unknown and a type variable is stored instead. The code maintains two synthesized attributes, the type of the specific term and the type environment for the subtree. When the rule is finished, the type environment at the root term contains the relationship between all the types in program. The types of

functions in other modules can be accessed by using *get-global* and added to the type environment. As the types for each function is inferred they can stored in the global space with *put-global*, which can be persisted to a file for other modules to reference later. Once all the types have been unified, another rule similar to RepMin can distribute the types to all the terms in the AST.

```
(define (simple-type-inference this)
  (let ((put-type (cut put-attribute <> <> 'type))
        (put-tenv (cut put-attribute <> <> 'type-environment))
        (get-type (cut get-attribute <> 'type))
        (get-tenv (cut get-attribute <> 'type-environment)))
    (match this
          (($ Var X)
           (let ((T (new-binding)))
             (put-type
              (put-tenv this (merge-tenv (new-tenv) (unify X T)))
              T)))

          (($ Value (? boolean?))
           (put-type (put-tenv this (new-tenv)) 'Bool))

          (($ Value (? number?))
           (put-type (put-tenv this (new-tenv)) 'Num))

          (($ If C T E)
           (let ((tenv (merge-tenv (get-tenv C)
                                   (get-tenv T) (get-tenv E)
                                   (unify (get-type C) 'Bool)
                                   (unify (get-type T) (get-type E)))))
             (put-type
              (put-tenv this tenv)
              (get-type T))))

          (($ App F As)
           (let* ((T (new-binding))
                  (tenv1 (foldr merge-tenv (get-tenv F)
                                (map get-tenv As)))
                  (tenv (merge-tenv tenv1
                                    (unify (get-type F)
                                           (apply func-type T
                                                  (map get-type As))))))
             (put-type
              (put-tenv this tenv)
              T)))

          (($ Lambda Ps B)
           (let* ((ftype (apply func-type (get-type B)
                                (map get-type Ps)))
                  (tenv (foldr merge-tenv (get-tenv B)
                               (map get-tenv Ps))))
             (put-type
              (put-tenv this tenv)
              ftype))))))

(define simple-type-inference-rule
  (strategy (bottomup simple-type-inference)))
```

Figure 6-7. Type inference rules for simply typed lambda calculus

The type inference rules for the language terms are simple. If a constant is found, it is either a Bool or Num type. Variables add a relationship between the variable name and a new type variable T into the type environment. A properly typed *if* expression requires the condition be a Bool type. Both the *then* and *else* clause plus the return type for the *if* expression must be the same type. A function application adds a function type to the type environment that takes the types of all the arguments and returns a type variable. A *lambda* term also adds a function type that takes the types of all its parameters and returns the type of the body. These inference rules will fill the type environment with all the type information required to find an exact type for every term, or discover an inconsistency in the program.

The type inference rules can be extended by other programmers to handle new types. For example, a library that implements structures can add extra inference rules to handle structure access and assignment. Furthermore, new type systems can be implemented for DSLs by using the same technique. Finally, the source language can be extended with an embedded DSL to allow programmers to provide explicit type declarations, which is how most typed languages solve ambiguities when computing types. Type systems are just another analysis that can be inserted into a compiler. Though it is an integral part of the source language, it does not need to be a built-in part of the compiler.

6.4. Limitations

There has been a tremendous amount of research in program analysis. Most advanced analyses are performed using a complex graph representation of the program, e.g. control-flow or data-flow. Compilers strive to perform many program transformations together to amortize the cost of constructing the graph. Since Sausage only performs one transformation per pass, it could be too expensive to reconstruct an elaborate program graph every time. Therefore, Sausage should be limited to those analyses that can be computed quickly or incrementally. Of course, any analysis can be simulated with AGs and extra support code, so the line between what is and is not practical with AGs is a matter of efficiency, not computability.

Many powerful analyses for higher-order languages cannot easily be implemented with AGs. In particular, higher-order control flow analysis[111], i.e. tracking the use of first-class functions, could not be modeled with AGs efficiently. This makes it difficult to perform aggressive constant propagation, function inlining, type recovery, etc. Nevertheless, the AG framework can be used to create a simpler control flow graph by threading the AST. Building an AG that computes attributes efficiently and incrementally on a graph overlaying the AST would be an interesting area for future research.

6.5. Conclusion

The analysis system in Sausage is driven by a simple framework based on attribute grammars. Though limited, it provides sufficient power for many interesting analyses typically found in a compiler. The attribute grammar technique makes it easier for programmers to write a new analysis, since it remains true to the underlying AST. The system is built on top of the program transformation system, which allows an analysis and transformation to be executed together. Annotating an AST with information is a useful way to drive more powerful transformations.

Interprocedural and intermodule analyses and transformation can be implemented by using a persistent store. A variety of languages already use this technique, demonstrating its utility. Sausage extends the conventional implementation to allow arbitrary attributes in the persistent store. This, in conjunction with the compiler extension mechanism, allows Sausage to implement more powerful optimizations than previous user-extensible compilers.

Chapter 7. Examples

The Sausage meta-compilation system can be used to perform a wide variety of transformations, from common compiler passes to user-defined extensions. The examples in this section attempt to demonstrate the transformations possible with Sausage, while remaining clear and simple. More complex optimizations will, of course, be more complex than the rules described here. These examples demonstrate that significant program transformations can be performed with small, simple rules.

Though all the rules are designed for the Scheme programming language, it is important to keep in mind that Sausage is language agnostic. The rules are easy to write for Scheme (easier still for Haskell), but rules can also be written for imperative or object-oriented languages. The patterns and AST will certainly be more complex because those languages tend to be unwieldy. Nevertheless, other transformation systems demonstrate that it can be done. Sausage makes it easier to write modular rules that are integrated by the runtime, which should make it easier to write more complex transformations. Remember, Sausage operates over arbitrary trees. Define an AST for any language and Sausage can manipulate it.

Finally, these examples elide the issue of integrating these rules into a larger build system. How will the compiler find these extensions? Each compiler and language may choose to pass this information into the compiler in different ways. Since Sausage is

still a fragile prototype, the associations are listed manually within the Scheme interpreter that runs Sausage. In Scheme48 it might be integrated directly into its module declarations as a distinct attribute. In SML, it might be part of its compilation manager. In Java or C#, the extensions may be listed in a separate XML file that maps JARs or assemblies to extensions. This issue is certainly important for a real compiler, but can be ignored for a simple prototype with no users.

7.1. Optimizations

The following are examples of some common compiler optimizations for functional languages used by the Sausage meta-compiler for Scheme[29]. All of these optimizations are associated with the core Scheme system; therefore, they are executed last in the compiler pipeline. These optimizations are not executed in isolation; in fact, the interaction between these rules makes them more powerful. When beta reduction inlines a closure, other rules can fold and propagate constants, eliminate useless variables and remove dead code. The final example, conditional constant propagation, demonstrates how the interaction between these rules can improve a simpler optimization.

7.1.1. Beta Reduction

A core optimization for a functional language is to apply functions immediately where possible. This eliminates temporary closures and allows other optimizations to further reduce the term. In ANF the program will already be in beta normal form (i.e. no more beta reductions are possible); however, other optimizations, particularly inlining, may create new opportunities for beta reduction. The rule in Figure 7-1 demonstrates beta reduction for *let*-bound closures; however, a proper, and more complex, implementation of this rule must be careful about code explosion and, in degenerate cases, infinite beta reduction.

```
(define beta-reduction/let
  (match-lambda
   (($ Let V (and ($ Lambda Ps E) L) B) (=> fail)
    (let* ((V? (lambda (s) (equal? V s))))
           (newB
            (oncetd (match-lambda
                      (($ Set (? V?)) (fail))
                      (($ App (? V?) As)
                       (fold-right make-Let E Ps As)))
                    B)))
      (and newB (make-Let V L (normalize newB)))))))
```

Figure 7-1. Simple beta reduction

Though this rule has been simplified, it serves to illustrate the concept of beta reduction within Sausage. The pattern matches any variable bound to a *lambda* expression. It replaces one application of V with a set of *let* bindings that map the parameter names to the application's arguments, ending with the body of the *lambda*. If

it ever finds a side-effect on V, it aborts this rule. If the closure has been inlined somewhere in B, the code needs to converted back into correct ANF form by calling *normalize*. This ensures the AST is correct and performs alpha-conversion (variable renaming) to prevent name collisions. Notice that this rule does not eliminate the original *let* binding because there may be more uses of V elsewhere in B. If there are no more uses, then other optimization rules will eliminate the *let*.

7.1.2. Useless Variable Elimination

Many optimizations create temporary variables and intermediate *let* expressions that clutter the AST and may confuse other optimizations. It would be useful to eliminate any variables bound to another variable. In a functional language this is easy to do. So long as there are no assignments to either variable, the former can be replaced by the latter. The first clause in the rule shown in Figure 7-2 does just this. If either variable X or Y in the pattern is assigned to within the body, the rule immediately fails. Otherwise, it replaces all uses of X with Y. If there are no uses of X, then it simply returns B with no changes.

```
(define useless-variable-elimination
  (match-lambda
    (($ Let ($ Var X) (and nodeY ($ Var Y)) B) (=> fail)
     (let ((X? (lambda (s) (equal? X s)))
           (Y? (lambda (s) (equal? Y s))))
       (try (manytd (match-lambda
                      (($ Var (? X?)) nodeY)
                      (($ Set (or (? X?) (? Y?))) (fail)))
                    B))))
    (($ Let ($ Var X) E B)
     (let ((X? (lambda (s) (equal? X s))))
       (and (not (where/cc
                   (oncetd (match-lambda
                             (($ Var (? X?)) (return #t))))))
            (make-Begin E B))))))
```

Figure 7-2. Useless variable elimination rule

The second clause eliminates a *let* expression when there is no use of the variable X. If the variable is found just once in the body by *oncetd*, then this rule jumps out of the *where/cc* strategy by returning #t. If the variable is not used in the body, *oncetd* will return #f, which the *not* turns into #t, and it wraps E and B in a new *begin* expression. The *where/cc* strategy is used here because it only needs to check for the existence of the X variable, but not actually do anything with it. The result eliminates the variable and the *let*-binding.

7.1.3. Dead Code Elimination

A simple yet useful optimization is to eliminate any code that cannot affect the result of the program. The intermediate language ANF Scheme is already simplified enough that detecting dead code is easy. First, an *if* expression with a constant value of #f is replaced

by the *else* clause; otherwise, it is replaced by the *then* clause. This is a simple form of unreachable code elimination. Second, a *begin* form consists of a statement followed by an expression. If the statement is a constant, variable, closure or a pure function, then it can be removed. This is called unused code elimination. Together, the rules perform what is commonly referred to as dead code elimination.

```
(define dead-code-elimination
  (match-lambda
   (($ If ($ Value #f) T E) E)
   (($ If ($ Value val) T E) T)
   (($ Begin (or (? Value?) (? Var?) (? Lambda?)
                 ($ App (? pure-function?)))
             E)
    E)))
```

Figure 7-3. Dead code elimination rule

The implementation of the predicate *pure-function?* checks the attribute list attached to the function name. Any library can set this attribute for functions that have no side-effects (most Scheme functions, but not *display*, for example). Since a *begin* statement will ignore the result of a function, there is no reason to execute the code unless there might be a side-effect.

7.1.4. Conditional Constant Propagation

More aggressive constant propagation algorithms integrate dead code elimination and useless variable elimination and execute over an intermediate SSA form. This enables the algorithm to run more efficiently and to discover more opportunities to propagate

constants. Fortunately, ANF form is already similar to SSA form. By executing a simple constant propagation rule with the previous two rules, plus additional rules to fold Boolean expressions, Sausage automatically integrates all these rules to optimize an AST. It can eliminate dead code which may contain assignments that interfere with the simple constant propagation algorithm. All these optimizations are mutually beneficial and do a better job of optimizing the AST.

7.2. Libraries

The primary motivation for developing the Sausage system was to allow programmers to add special optimizations for their own libraries. Conventional compilers do add library specific optimizations, but only in limited and ad-hoc ways. Important libraries like MPI[142] or a standard list library will be optimized by some compilers because they are worth the compiler writer's time to implement. But for everything else, there is no compiler support. Programmers could add optimizations directly to an open-source compiler, but they may fork the source code substantially, making it difficult to integrate updates from the main compiler into their implementation. The Sausage system gives programmers the ability to add extensions easily. The following examples are simple yet powerful in practice.

7.2.1. DEFORESTATION

Many functional languages compose small functions together to perform more powerful operations by communicating via intermediate data structures. This is why the list data type is ubiquitous in many functional languages. For example, in the expression (sum (iota 0 9)), *iota* produces a list of numbers from 0 to 9, which *sum* consumes to produce the value 45. Generating the intermediate list of numbers is inefficient since it will be destroyed immediately. Deforestation is a compiler optimization that eliminates these intermediate data structures[137]. Common Lisp provides an elaborate macro package – really an embedded DSL compiler – to deforest list processing expressions wrapped in a *loop* macro[138]. However, it cannot optimize across different *loop* macros or functions. Some compilers for functional languages use a cheaper variant of this technique called *short-cut deforestation*[53]. The idea is to transform existing functions into a form that explicitly indicates how they produce and consume intermediate data structures. Once these have been inlined, it is easy to merge a list producer and consumer into one that does not produce an intermediate list at all. In the Glasgow Haskell compiler, deforestation reduced allocation in a benchmark suite by an average of only 5%, but these benchmarks were already written to avoid generating intermediate lists in most cases anyway. In another example with rose trees, GHC reduced allocation by 96%. If deforestation is implemented in a compiler, it

may encourage programmers to write in a more modular, functional style, rather than hand-optimize list processing functions themselves.

```
Standard definitions:
(define (sum lst) (fold-right + 0 lst))
(define (iota n m) (if (> n m) '() (cons n (iota (+ n 1) m))))

Expose cata-build functions:
(define (sum lst) (cata + 0 lst))
(define (iota' n m)
  (lambda (kons nil)
    (if (> n m) nil (kons n (iota' (+ n 1) m)))))
(define (iota n m) (build (iota' n m)))

Demonstrate deforestion:
(sum (iota n m))
→ (cata + 0 (build (iota' n m)))
→ ((iota' n m) + 0)
```

Figure 7-4. Short-cut deforestation example

The code in Figure 7-4 demonstrates the steps to performing short-cut deforestation. The standard definitions for *sum* and *iota* embed the requirement for an intermediate list into their implementation. In *iota*, the code explicitly uses *cons* and '() to construct a list. The definition of *sum* is rewritten to use *cata* (e.g. catamorphism), which is a generalization of *fold-right* that operates over any tree-like data structure. The function *iota* is transformed into a curried function *iota'* that takes an implementation of *kons* and *nil*. The *build* function simply instantiates a function with the standard *cons* and '() (empty list) it would have used anyway. Short-cut deforestation first inlines the definitions of *sum* and *iota*. Then the cata-build transformation rule matches pairs of *cata* and *build*, and replaces *build* with the constructor and nil arguments from *cata*. The

final expression using *iota'* will add together elements of the list rather than construct the list with *cons*.

Though there are elaborate transformations that can automatically deduce the cata-build form for many user defined functions, most functional compilers implement list functions in this special cata-build style. For brevity, assume that Sausage inlines the cata-build implementations for *sum* and *iota* into a program, as shown in Figure 7-5. The *cata-build-rule* looks for a use of *build* in a *let* form. The argument P is a variable bound to a list producing function. Within the body of the *let*, the first part of the rule transforms one use of *cata* on the variable V into a form that eliminates the *cata* form altogether.

The second portion of this rule is a bit more complex. The *cata-build-rule* should not replace more than one use of that particular *build* form, because it will execute it twice. For example, if the result from *iota* was used in two different places, then the intermediate list is needed so it can be consumed twice (say for *sum* and *filter*). The second portion of this rule checks if there are any remaining uses of the variable V. If so, the *and* expression fails and the rule returns #f; no changes are made to the AST. If not, then it can safely return the modified body that removes the *cata*.

```
(define inline-example
  (match-scheme
   (('sum a) `(cata + 0 ,a))
   (('iota n m) `(build (iota' ,n ,m)))))

(define cata-build-rule
  (match-lambda
   (($ Let ($ Var V) ($ App ($ Var 'build) P) B)
    (let ((V? (lambda (s) (equal? s V))))
      (let ((B'
             ((oncetd (match-lambda
                        (($ App ($ Var 'cata) (C N ($ Var (? V?))))
                         (make-App P (list C N)))))
              B)))
        (and B'
             (not ((where/cc
                    (oncetd (match-lambda (($ Var (? V?)) (return #t)))))
                   B'))
             B'))))))

(define final-cleanup-build-rule
  (match-scheme
   (('build P) `(,P cons ()))
   ((‘cata C N L) `(fold-right ,C ,N ,L))))
```

Figure 7-5. Cata-Build deforestation rule

After all the rules have run and the tree has reached fixed point, Sausage executes any rules registered as finalizers. These rules have one last opportunity to modify the AST without triggering the other rules. Here, the *final-cleanup-build-rule* rewrites all leftover uses of *build* to call the curried function P with *cons* and ‘(). It also replaces uses of *cata* with the function *fold-right*, which in this case does the same thing. The final program will have no mention of the *cata* and *build* special functions in the code at all. And it will merge many, though not all, pairs of list producing and consuming functions to eliminate the wasteful generation of intermediate lists.

7.2.2. FFT

Writing efficient code is difficult, tedious and error-prone, but it must be done. The Fast Fourier Transform (FFT) is an example of a routine that can be made vastly more efficient by specializing it for the desired input vector size. The FFTW system[50] automatically generates code for specific input sizes, probing the hardware to select an ideal implementation. The graph in Figure 7-6 demonstrates that the special optimizations implemented by FFTW allow it to exceed all but Intel's special-purpose FFT library; in addition, FFTW beats all implementations for non-power of two FFTs. A code generator written in MetaOCaml has duplicated these results[75]. However, both have had to write domain specific optimizers to produce quality code. This means they had to duplicate some of the functionality that already existed in their compiler just so they could add a few domain-specific optimizations. For FFTW, the entire DSL compiler is over 8000 lines of O'Caml code. It took 600 lines of MetaOCaml code written in a convoluted monadic style, though it only implements one FFT algorithm. With Sausage, one can simply add the domain-specific optimizations to the existing compiler.

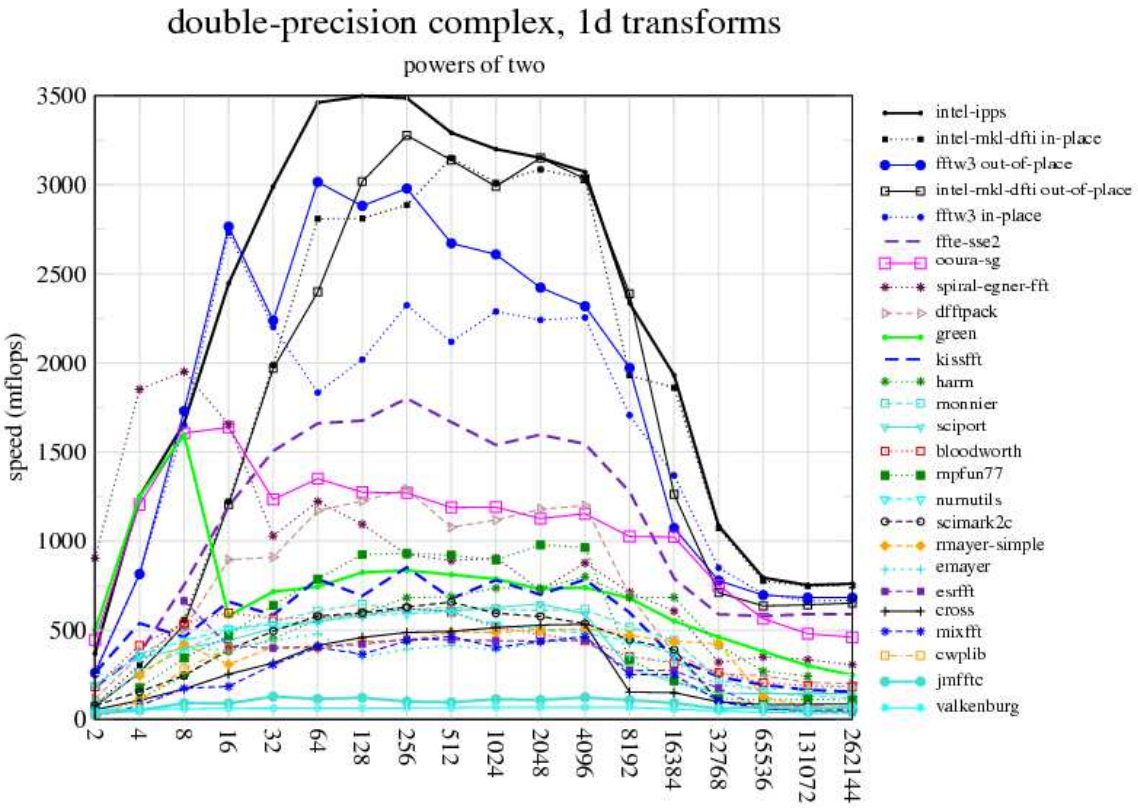

Figure 7-6. Radix-2 FFT on 2.4 GHz Pentium 4 w/ Intel compilers

The rules in Figure 7-7 implement a simple and inefficient recursive FFT algorithm for vectors whose size is a power of two[32], i.e. radix-2. Of course, additional FFT algorithms can easily be added for different input sizes, as well as a default algorithm when the input size is not known at compile-time. The generated code takes as input a vector of input values and sets the final FFT results back into the same vector. The main rule *fft-rule* extracts the values from the array and assigns them to temporary variables,

generated by gensym. It then applies the function *recursive-fft* to the input variables. It uses Scheme's *call-with-values* to fetch the results back from *recursive-fft*, which it then assigns into the original array.

```
(define fft-rule
  (match-scheme
   (('fft-gen n)
    (let* ((indices (l:iota n))
           (inputs (map gensym indices))
           (outputs (map gensym indices)))
      (let ((input-code
             (map (lambda (v i)
                    `(,v (vector-ref a ,i)))
                  inputs indices))
            (output-code
             (map (lambda (v i)
                    `(vector-set! a ,i ,v))
                  outputs indices)))
        `(lambda (a)
           (let ,input-code
             (call-with-values (lambda () (recursive-fft ,@inputs))
               (lambda ,outputs
               ,@output-code)))))))))

(define recursive-fft-rule
  (match-scheme
   (('recursive-fft input) `(values ,input))
   (('recursive-fft inputs ...)
    (let ((n (length inputs))
          (outputs (map gensym inputs)))
      `(call-with-values (lambda () (recursive-fft ,@(evens inputs)))
      (lambda ,(l:take outputs (/ n 2))
        (call-with-values (lambda () (recursive-fft ,@(odds inputs)))
          (lambda ,(l:drop outputs (/ n 2))
            (merge-fft ,@outputs)))))))))

(define merge-fft-rule
  (match-scheme
   (('merge inputs)
    (let ((n (length inputs))
          (y0 (l:take inputs (/ n 2)))
          (y1 (l:drop inputs (/ n 2)))
          (outputs (map gensym inputs)))
      `(values
      ,@(map (lambda (y0 y1 k)
               `(+ ,y0 (* (unity-root ,n ,k) ,y1)))
             y0 y1 (l:iota (/ n 2)))
      ,@(map (lambda (y0 y1 k)
               `(- ,y0 (* (unity-root ,n ,k) ,y1)))
             y0 y1 (l:iota (/ n 2))))))))
```

Figure 7-7. Recursive FFT implementation

The rule *recursive-fft-rule* expands into an implementation of a single stage of the FFT algorithm. It splits the inputs into an even and odd set and calls *recursive-fft* on both. It then merges the results of both recursive calls. In the base case, the rule simply returns the input value. Finally, the *merge-fft-rule* arranges the actual computation by multiplying the complex nth root of unity with the results of the latter *recursive-fft*, and combining them with the results from the first *recursive-fft*. It uses the place holder *unity-root* to represent the complex nth root of unity taken to the kth power. The generated code for an FFT of size 4, in Figure 7-8, still contains many redundant and expensive floating point multiplications and additions, particularly with *unity-root*. All the uses of *call-with-values* have been replaced by another rule that recognizes the degenerate case:

```
(call-with-values (lambda () (values args ...))
    (lambda (params ...) body))
```

It binds each expression in *args* with the correct parameter from *params* with a let-form, and removes *call-with-values*, *values*, and both lambda forms.

```
(lambda (a)
  (let ((a0 (vector-ref a 0))
        (a1 (vector-ref a 1))
        (a2 (vector-ref a 2))
        (a3 (vector-ref a 3)))
    (let ((y0 (+ a0 (* (unity-root 2 0) a2)))
          (y1 (- a0 (* (unity-root 2 0) a2)))
          (y2 (+ a1 (* (unity-root 2 0) a3)))
          (y3 (- a1 (* (unity-root 2 0) a3))))
      (let ((r0 (+ y0 (* (unity-root 4 0) y2)))
            (r1 (+ y1 (* (unity-root 4 1) y3)))
            (r2 (- y0 (* (unity-root 4 0) y2)))
            (r3 (- y1 (* (unity-root 4 1) y3))))
        (vector-set! a 0 r0)
        (vector-set! a 1 r1)
        (vector-set! a 2 r2)
        (vector-set! a 3 r3)))))
```

Figure 7-8. FFT for input vector of size 4

To further reduce this code, special rules for reducing the complex roots of unity are implemented in *unity-root-rule*, shown in Figure 7-9. The problem is that computing these values by constant propagation will result in a very small rounding error; therefore, the values will not be precisely 1, 1i and -1 as desired. By implementing domain-specific constant-folding rules, these precise values can trigger a cascade of additional code improving transformations. The next two patterns implement a pair of well-known simplifications, the cancellation and halving lemma. The problem is that for larger FFTs there will be many more distinct *unity-root* computations, but they are really the same value. For example, (unity-root 8 4) is equivalent to (unity-root 2 1) which is -1. These two lemmas reduce the total number of distinct *unity-root* forms and allow common sub-expression elimination to replace redundant computations. The

final pattern computes the value of any remaining *unity-root* forms; therefore, none will be left in the final code.

```
(define unity-root-rule
  (match-scheme
   (('unity-root N 0) 1)
   (('unity-root 2 1) -1)
   (('unity-root 4 1) 0+1i)

   ;; cancellation lemma
   (('unity-root N K) (=> fail)
    (if (and (even? N) (<= K (/ N 2)))
       `(negate (unity-root ,N ,(- K (/ N 2))))
      (fail)))

   ;; halving lemma
   (('unity-root N K) (=> fail)
    (if (zero? K) (fail)
      (let ((d (gcd N K)))
        (if (> d 1)
            `(unity-root ,(/ N d) ,(/ K d))
            (fail))))))

   ;; compute unity root
   (('unity-root N K) (compute-unity-root N K))))

(define final-negate-rule
  (match-scheme
    ((‘negate V) `(- ,V))))
```

Figure 7-9. Simplify complex roots of unity

The optimized code in Figure 7-10 shows there are only two remaining multiplications for complex numbers. All the other *unity-root* forms have been replaced with constants and the code has been simplified. Only two complex multiplications remain; however, a code generator that converts this code to C can safely eliminate both operations. In C, complex numbers might be represented by a structure that explicitly separates the real and imaginary components of a complex number. Therefore, these multiply operations

will be expanded into a floating point multiply with 0 and 1, which can be trivially folded away. Sausage could do this, too, but Scheme already represents complex numbers nicely.

```
(lambda (a)
  (let ((a0 (vector-ref a 0))
        (a1 (vector-ref a 1))
        (a2 (vector-ref a 2))
        (a3 (vector-ref a 3)))
    (let ((y0 (+ a0 a2))
          (y1 (- a0 a2))
          (y2 (+ a1 a3))
          (y3 (- a1 a3)))
      (let ((r0 (+ y0 y2))
            (r1 (+ y1 (* 0+1i y3)))
            (r2 (- y0 y2))
            (r3 (- y1 (* 0+1i y3))))
        (vector-set! a 0 r0)
        (vector-set! a 1 r1)
        (vector-set! a 2 r2)
        (vector-set! a 3 r3)))))
```

Figure 7-10. Optimized FFT code for size 4

When all the rules have finished executing, there is one finalizer rule that must be run. The *final-negate-rule* in Figure 7-9 implements a curious domain-specific optimization discovered by the FFTW system. The cancellation lemma for complex roots of unity will negate the value of a simpler *unity-root* form. Most compilers will compute this at compile-time, producing a negative constant. It turns out that for large FFTs it is inefficient to store negative values in a register because it complicates register allocation. Instead, it is cheaper to only store the positive values and calculate the negative number at runtime. Therefore, this rule uses *negate* to prevent other constant

folding optimizations from computing a negative number. When all the rules are finished, the *final-negate-rule* turns it back into a "–" operation.

Size	Direct Staging	FFTW	Sausage	Compile time
4	32/32	0/16	0/16	0.08s
8	96/96	4/52	4/52	0.28s
16	256/256	28/150	28/150	0.62s
32	640/640	108/398	108/398	2.17s
64	1536/1536	332/998	332/998	10.04s
128	3584/3584	908/2406	908/2406	53.53s
256	8192/8192	2016/5184	2016/5184	550.68s

Figure 7-11. Floating point addition/multiplication for FFT

The key idea demonstrated by this example is that a compiler extension can easily generate optimized code at compile time. The implementation uses rewrite rules to simulate the behavior of a simple, recursive FFT algorithm. The domain-specific rules for the complex nth roots of unity actually transform the code into the efficient parallel FFT described in textbooks. Most temporary variables have been eliminated. The extra closures generated to use *call-with-values* have been eliminated. Common subexpression elimination will ensure that every computation will only occur once. As shown in Figure 7-11, Sausage reduces the number of floating point operations in the naïve implementation of the FFT to match that of FFTW. Also shown are the compile times (in seconds) for Sausage, which grows outrageously high as the code size increases because no work has been done to optimize performance. This

implementation is simpler than others because it relies on the existing compiler to do the bulk of the work; indeed, many of the optimizations can be reused in other domains.

7.3. DSLs

Most programmers are already using DSLs in their existing programs. The most popular include format strings, regular expressions and SQL queries. These DSLs are wrapped in strings and sent to a language processor at runtime. The main language compiler does not know anything about the structure of these languages; therefore, it cannot provide any assistance in error checking these little DSLs. Though syntax macros can implement many of these directly in Scheme and Lisp[61], there are still limitations. Macros do not know the context in which a call is made and cannot rely on other compiler optimizations to simplify the program. Sausage provides a bit more power than available with macros, which enables more interesting transformations.

7.3.1. Format Strings

Many languages adopt the convenient formatting directives used by C's printf statement. For example, the expression in Figure 7-12 expects month and date to be a string and number, respectively. It converts the latter into a string and concatenates everything together. To do this, the format function interprets the format string at

runtime to choose the appropriate type conversion routines. If a language provides static type checking, it is impossible to ensure that this code is type safe. If month is really a symbol type, the compiler cannot warn the programmers that the format string is incorrect.

```
(format “Date: ~S ~D, 2005” month date)
→
(string-append “Date: “ month “ “
               (number->string date) “, 2005”)
```

Figure 7-12. Format string example

The solution is to partially evaluate the format string with respect to its arguments. The first rule in Figure 7-13 parses the format string and generates a list of strings and special *format/one* functions, which take a single directive and an argument. The entire list is appended together into a single string. This breaks apart the complex format string into a sequence of low-level format calls. This allows other library authors to dynamically extend the format string with new directives for special data types. The next two rules might be implemented in different libraries. Each converts *format/one* with a specific directive into the appropriate string conversion operation. If a programmer wants to add new directives for complex numbers or records, he simply adds a new *format/one* rule to replace a different format directive (maybe “C” and “R”). The format function can be partially evaluated and extended with directives.

```
(define evaluate-format-string
  (match-lambda
   (($ App ($ Var 'format) (($ Value F) As ...))
    (normalize
     (parse
      (cons 'string-append
          (letrec next ((Fs (parse-format-string F))
                        (As As))
                (match (car Fs)
                     ((? string? S)
                      (cons S (next (cdr Fs) As)))
                     (('directive D)
                       `((format/one ,D ,(car As))
                         ,@(next (cdr Fs) (cdr As))))))))))))

(define format/one-string
  (match-scheme
   (('format/one "S" A) A)))

(define format/one-number
  (match-scheme
   (('format/one "D" A) `(number->string ,A))))
```

Figure 7-13. Partially evaluate format strings

Since the *format* and *format/one* functions will be replaced by type-specific operations, a language that performs type checking can now check that each argument is used correctly. When it generates the expression (number->string date), the type checker can ensure that date is a number type. In addition, these rules can benefit from constant folding. If the value of date is know at compile-time, it can be propagated into that position and converted into a string at compile-time. Furthermore, a folding rule for string-append can merge adjacent strings in its argument list into a single string.

This same technique can be used to lift other DSLs out of strings and into the language, where the compiler can perform more aggressive checks and optimizations on them.

The interpreter for a DSL should be written as a set of calls to a library. A rule can then be written to partially evaluate the interpreter with the DSL code found within a string. However, the derivation should be done in stages to allow programmers to add extensions. In the example, the rule for *format* could have transformed the directive directly to their type specific operations and skipped the intermediate transformation into *format/one*. But this would not allow other programmers to extend the format string with new types. Transforming a DSL in stages is an important step in writing extensible systems.

7.3.2. Pattern Matching

The pattern matching macro used by Sausage suffers from a frustrating limitation. It cannot detect lexically bound variables in the patterns; instead, it assumes all identifiers are new pattern variables. In many of the examples, a pattern matches on a specific variable name in the AST which it wishes to use in a sub-pattern. In Figure 7-14, the first pattern matches a variable named `V` in a *let* form. The next line creates a predicate *V?* to locate variables with the same name in the sub-patterns. It would be convenient if a pattern that uses a variable bound in a higher scope would test for equality automatically. That is, if the sub-pattern was `($ var V)`, the pattern matcher should recognize that `V` is bound in the previous pattern and test for equality with the value it matches at that position. A Lisp-style macro system, however, cannot

determine if V has already been bound unless all the surrounding code has been wrapped by a macro that then walks the code.

```
(define match-variable-rule
  (match-lambda
   (('match-lambda clauses ...)
    (let ((bounded (get-attribute this 'bound-variables)))
      `(match-lambda-aux ,bounded ,@clauses)))))
```

Figure 7-14. Discovering lexically bound variables

To find all lexically bound variables, run the *bound-variable-analysis* rule described in the previous chapter. It adds the names of all local variables created in a higher scope to the *bound-variables* attribute. In Figure 7-15, the *match-variable-rule* simply attaches the set of bound variable names to the *match-lambda-aux* expression. Another rule, presumably, will use this information to generate better pattern matching. code.

```
Original:

(define (example p)
  (match-lambda
    (($ Var V)
      (match-lambda
        (($ Var V) dosomething)))))

First Transformation:

(define (example p)
  (lambda (tmp1)
    (and (Var? tmp1)
         (let ((V (Var-name tmp1))) ;; bind V
           (match-lambda-aux (V)  ;; add bound variables
             (($ Var V) dosomething))))))

Final Transformation:

(define (example p)
  (lambda (tmp1)
    (and (Var? tmp1)
         (let ((V (Var-name tmp1)))
           (lambda (tmp2)
             (and (Var? tmp2)
                  (equal? (Var-name tmp2) V)) ;; test equality
                  dosomething))))))
```

Figure 7-15. Match-lambda transformation example

This example also illustrates why it is important to perform transformations incrementally. The rules described above do not explicitly locate new variables created by a pattern. For example, the original code in Figure 7-1 has two nested *match-lambda* forms that match on a *Var* structure and bind its name with V. In the existing implementation of pattern matchers, the nested pattern will create a fresh variable V even though it will be bound by the first pattern. The rules in this example will, instead, transform the first *match-lambda* form into explicit matching code, which binds the variable V with the name within the *Var* structure (first transformation). This will

restart the compiler pipeline on the new AST. The second time through the *bound-variable-analysis* will find the variable V used in a *let* form. This information will be passed into the nested *match-lambda* form. When it is transformed into explicit matching code, it will test whether the value within *Var-name* is equal to the value bound to V.

This would be far more difficult to emulate with macros. And it would be impossible if either *match-lambda* used *example*'s parameter *p* within a pattern. Even a Lisp-style macro could not discover that *p* is bound in a higher lexical scope without shadowing the definition of *define*, but Sausage can do it easily. More importantly, multiple embedded DSLs may be expanded at the same and their interplay may reveal more variables. For example, a variable bound within a regular expression will not be discovered until it is expanded by partial evaluation. Only then can this rule discover the variables hidden inside a regular expression. The order in which forms are transformed can be important for a rule like this.

7.4. Conclusion

The examples here demonstrate that a variety of program transformations can easily be implemented within the Sausage meta-compilation system. These examples range from typical compiler optimizations to domain specific transformations. It should be clear that, at a minimum, the Sausage system is at least as powerful as Lisp-style

macros, which is equal to or more powerful than most program transformation systems. Therefore, Sausage can easily duplicate active libraries implemented by C++ templates[129, 112], deforestation on rose trees as implemented by Haskell rewrite rules[100], and compile-time staging annotations as in MetaML[121]. Sausage can easily duplicate Catacomb's code transformations[115] because it, too, is embedded within the compiler. Sausage goes beyond these other systems by allowing programmers to write more complex rules that can operate over large portions of a program at once. In addition, it provides some analysis features that allow programmers to write specialized analyses to help drive their transformations. The Sausage system provides all the functionality found in earlier systems and more.

Chapter 8. Conclusion

The Sausage meta-compilation system replaces the conventional static, black-box compiler with a flexible and extensible transformation tool. Often, macros, templates and many program transformation tools are too limited because they do not have access to the transformational power and, more importantly, the wealth of information available within a compiler. Meta-compilation makes these features easily accessible to programmers, allowing them to mold compilers into specialized programming tools.

There appears to be growing interest in program transformation tools and domain specific languages. One sees hints of these ideas in other systems, but they are often one-off hacks to solve a particular problem. Meta-compilation offers a broad solution that can be applied to many programming problems. An implementation of a meta-compiler does not need to be as open and powerful as Sausage; instead, it can offer programmers a more limited set of extensions tools. In fact, there is still much more to be done before meta-compilation can be used in a commercial software environment. The prototype described here is limited, but there are many ideas explored by other researchers that can be incorporated into a more practical implementation of Sausage. The idea of meta-compilation is important. Sausage is merely a demonstration.

8.1. Summary

Sausage allows programmers to dynamically extend a compiler with arbitrary analysis and transformation passes. The extensions are associated with code libraries, and a group of core compiler optimizations are associated with the language (thus always lowest priority). When a module is compiled, Sausage determines an ordering for compiler extensions by analyzing the dependency graph between the libraries reached by that module. The extensions are loaded into a compiler pipeline. A parser turns the module into an AST. The compiler extensions are then applied to the AST. When an extension successfully transforms the AST, the new AST is sent to the beginning of the pipeline. If the extension could not find any optimization opportunities within the AST, it fails and sends the AST to the next extension in the pipeline. When every extension in the pipeline fails to change the AST, it restarts once more from the beginning of the pipeline and executes any transformation rules registered as finalizers. These do not restart the pipeline if successful; each extension modifies the tree one last time. When all are finished, the final AST can be sent to a code generator.

Sausage provides a convenient DSL for writing compiler extensions. Sausage itself is language independent. To apply Sausage to another input language, the AST and a small set of primitive tree walking combinators need to be defined. The transformation rules use a pattern-matching language to easily match against the AST. More complex tree-walking routines can be combined to perform sophisticated transformations on

the AST. In addition, a set of services running concurrently with the extensions compute attributes either bottom-up or top-down in the AST. These features allow programmers to dynamically extend a compiler with new transformations while reusing the existing compiler.

8.2. Trends

As of this writing, there have been several recent trends in programming languages and compilers that suggest that meta-compilation may not be so far-fetched. The idea of exposing raw access to a compiler may disturb some researchers, but programmers are demanding more control over their tools. The following examples do not provide full access to a compiler; instead, people are providing small hacks and hooks to make their tools more flexible. Sausage provides a superset of most of these features.

8.2.1. Reflection

Various systems today use runtime reflection to automatically enable services for the programmer. The Java and .NET runtime systems use reflection to offer services such as object proxies, serialization, and extra security features. The Ruby on Rails system[124] uses reflection to generate code to automatically store and retrieve objects from a relational database, i.e. a form of automatic object persistence. A similar tool for Java, Versant's FastObjects[132], rewrites the bytecodes in a class file to support object

persistence. Reflection is fast becoming a powerful way for programmers to dynamically extend the services offered by a runtime or program.

In fact, the upcoming version of Java, code-named Mustang, has an extensible javac compiler to allow programmers to write annotation processors[35]. This allows programmers to intercept an AST before the javac compiler executes and analyze the code and annotations. Unfortunately, programmers cannot modify the AST; however, they can generate other code and data files. It is being used as a way to generate additional code, not modify existing code. Regardless, the trend is towards opening the compiler and allowing programmers to write code generators and static checkers.

8.2.2. Static Checking

There has been renewed interest in static code checking tools. Long ago C programmers once used *lint* to perform a large set of static checks on a program before compiling. However, the C compilers have since hard-coded most of the checks performed by *lint*, thus making it obsolete. FxCop from Microsoft is an extensible static checking tool that resembles the venerable *lint* tool. Similar tools like Splint[45, 36] and xg++[42, 60] allows programmers to add custom checking rules for libraries. The latter has found thousands of errors in Linux and OpenBSD. Extensible static checking tools are an important way to discover more errors at compile-time.

8.2.3. Library Specific Compilation

There are some interesting cases where compilers have been tweaked to give it extra information about common libraries. For example, the Java and .NET runtime systems have optimizations specialized for array manipulation[20]. In particular, a loop of the form "for (int i=0; i < myarray.Length; i++) { ... }" would be expensive because it calls the Length method each time it checks the condition. Instead, these compilers optimize this special case by calling Length just once and removing, in most cases, array bounds checking within the body of the loop. This code can be optimized because the special properties of arrays are hard-wired into the compiler.

In the Java command line compiler javac, there is a special optimization for string concatenation[54]. Concatenating strings using the "+" operator is expensive because the contents from the two strings are copied into a newly allocated string. Instead, the Java compiler transforms concatenation into uses of the StringBuilder library, which concatenates strings much more efficiently. The expression "s1 + s2" is transformed into "new StringBuffer().append(s1).append(s2).toString()". Obviously, programmers should use the StringBuilder class to efficiently concatenate strings. Since most fail to do so, this library specific optimization was added to convert an inefficient operation into a more efficient library call.

In another case, the Objective Caml compiler[87] has a *printf* function in the standard library that uses a format string similar to C. The compiler must statically ensure that the types declared within the format string match the types of the data passed in. For example, for the line "Printf.printf "%d" i" the compiler must ensure that *i* is an integer type. To do so, the compiler's type checker has a special extension for format strings that discovers the type specifiers and compares them with the function arguments. Without this library specific type checking rule, it would be impossible to type check an expression when information is hidden within a format string.

8.2.4. Domain Specific Languages

There has been a surge of interest in DSLs and code generators recently. Many of the new XML based tools like XPath, XSLT and XQuery are DSLs for matching, transforming and querying XML data, respectively. The Cg language[47] from Nvidia is a specialized DSL for writing programs for a GPU. Microsoft has released a DSL designer for the Visual Studio 2005 IDE, which allows individuals to design and use custom DSLs for their problem domains. The Meta-Programming System[40] from JetBrains, another IDE vendor, will also allow programmers to design and use custom DSLs. Not to be outdone, the Eclipse IDE also provides an API for manipulating Java ASTs and generating code[145]. More recently, the Ruby community has become more enthusiastic about writing embedded DSLs within Ruby.

8.3. Application

The meta-compilation technique implemented by Sausage can be used in a variety of ways to improve programming language design, compiler construction, and software engineering. These techniques are not limited to Scheme or functional languages. They can be applied in varying degrees to any language that supports some extension mechanism. Meta-compilation can be used to make it easier to build compilers, to encode special optimizations for standard language libraries, to extend the language for new domains, or to enable programmers to rapidly add any extension they need. Meta-compilation can be constrained to allow only a few domain specialists to add extensions. Or it can be exposed publicly to allow anyone to add program transformations, as macros are used today. Compilers are generally useful tools and can be used in many domains, if only one easily could get access to the internals.

8.3.1. Compiler Construction

Compilers today are often a collection of optimizations carefully orchestrated to efficiently analyze and transform a program. They are monolithic black boxes that cannot be easily extended by even the compiler writers themselves. The order of different optimizations and analyses must be manually constructed and the compiler must be rebuilt. Even compiler frameworks that purport to be extensible are, in fact, well written software with clear interfaces that allow knowledgeable compiler writers

to manually add new passes. A programmer must invest a significant amount of time learning the AST, libraries and the dependencies between existing passes to properly integrate a new compiler pass. More importantly, the compiler must be recompiled with all the new optimizations before it can be used.

Sausage offers compiler writers an easier way to integrate new optimizations into an existing compiler. As other systems show, a flexible framework allows compiler researchers a quicker way to test new optimizations. More importantly, compiler vendors can quickly add special optimizations and error-checking rules for individual customers. Sausage can also serve as a framework for students to study modern compiler design by adding small optimizations and measuring the effect on the generated code[108]. The Sausage system allows compiler writers to construct compilers and other programming tools in a piecemeal fashion.

8.3.2. Library Support

Though a compiler is designed to exploit the semantics of a language, it is ignorant of the semantics of the libraries. When designing a library API, there is a trade-off between simplicity and power; most opt for power. Since a more powerful API is usually low-level, it forces the programmer to "manually compile" his high-level ideas into opaque code that efficiently uses the more complex, low-level library APIs. Obviously, it would be better if the compiler could do this instead[128, 131]. In most

cases, the library writer has a more intimate understanding of how the library can be used most efficiently. With Sausage, a library writer can expose a simple API and provide an extension that transforms it into the more efficient low-level implementation. In the rare case that a programmer decides that the transformation is not good enough, he could always use the low-level API directly; much like C programmers who occasionally write assembly code. In addition, library writers can perform more aggressive static checks if programmers use the simpler, high-level API. Thus, programmers get extended static checking and better optimizations through library specific compiler extensions.

Even if a compiler vendor chooses not to expose meta-compilation to the masses, they can still provide explicit library support for the standard libraries. Haskell does this by providing deforestation rules for their standard list processing library. Java and .NET offer enormous standard libraries that could benefit from extended checking and optimizations. The meta-compilation system can be used to allow the library writers within the same organization to quickly add new library specific extensions. These extensions can go through the same quality control as the compiler to ensure correctness. Since most programs rely heavily on the standard libraries, these extensions should be a boon to programmers.

8.3.3. Language Extensions

In many cases, it would be useful to add explicit syntax in a language for a specific task. Unfortunately, most language syntaxes are not flexible at all. Currently, programmers write programs in DSLs within strings and send these programs to another language processor. Regular expressions and SQL queries are examples of popular DSLs. Conventional compilers cannot even alert programmers to syntax errors in the DSL, but Sausage can be used to explicitly check the code, if only for syntactic correctness, within the strings to statically ensure they are correct. For example, an XPath expression can be compared against a XSD specification at compile-time to ensure it is valid. Furthermore, these DSLs can be partially evaluated and optimized at compile-time.

For example, the language C#[3] v2.0 adds a new syntactic form called *yield* to ease the burden of implementing iterators. In the previous version, programmers had to implement the IEnumerable interface and maintain the state of the iterator manually. This proved difficult for many programmers and was rarely used. The new *yield* statement implements the interface and rewrites the class to save state by replacing some local variables with instance fields. It is a straightforward program transformation that can not be performed by a conventional macro system. However, a system like Sausage could implement this transformation with a compiler extension. It

simply needs to capture the class context and execute a sub-transformation that rewrites all *yield* statements within the class. With C#, as with any language, programmers had to wait several years before the compiler was finally upgraded to support this transformation.

In Scheme and Lisp, extending the language is trivial because the syntax is simply an s-expression. It has been said that Lisp programmers first implement a DSL for their problem using macros, and then write the program in that DSL. Many languages are now offering extensible annotations to provide a bit of syntactic flexibility, but it is not really enough. A compiler could offer an extensible parser[78, 22], but it would be difficult to merge the syntactic changes from a random collection of libraries. Extending the syntax with either pattern matching or state machines is fine, but what happens when one wishes to use both syntax extensions at the same time? To give programmers the syntactic convenience of an embedded DSL, languages will need to offer more flexible syntax forms.

8.3.4. User Extensions

Ultimately, programmers should be allowed to develop their own compiler extensions. By democratizing languages and compilers, new innovations will no doubt be discovered in entirely new domains. For example, advances in biocomputing may inspire new DSLs for querying DNA patterns. Oil services companies may develop DSLs

for geologists to analyze data from prospective fields. No doubt there is still room for innovation within traditional computer science domains. New hardware, new services, and new computing paradigms may inspire new DSLs and library extensions. Frankly, almost every elaborate library API is an opportunity to apply meta-compilation to simplify the API and/or design an embedded DSL.

8.4. Future Work

The Sausage prototype would need to be improved markedly before it could be used by anyone else. The current implementation is driven by small scripts within the Scheme48 interpreter that coordinate various libraries to transform Scheme into optimized Scheme. Obviously, a more practical system would be the primary goal of any future work. In addition to the possible future work listed in each chapter, there are some broad research goals that would be interesting to explore.

8.4.1. Statically Typed Implementation

The most significant next step is to reimplement Sausage in a statically typed language (STL). Also, the input language should be a STL and the AST should be a typed intermediate language[93]. Static type systems are immensely useful in quickly locating errors, especially across module boundaries. Curiously, Sausage did not suffer from many type errors, but it is small and has only one author. For larger systems with many

cooperating programmers, STLs are invaluable. Unfortunately, no STL is currently implemented as s-expressions; therefore, it is difficult to design useful embedded DSLs for writing compiler extensions. Nevertheless, it would be a significant step towards developing a production quality implementation of a meta-compiler.

8.4.2. Performance

The current implementation is designed to be simple. The use of functional data structures throughout ensures there are no subtle errors due to global side-effects. However, this can be a serious drag on program performance. One way to improve performance is to use structure sharing, where trees are reused whenever possible. This will reduce memory allocation and the overall memory footprint.

Since most rewrite rules are pure functions, it should also be possible to memoize some rules. Whenever it finds the same patterns in the AST, it can simply replace them all with the same result. More importantly, it should be possible to know when a rule has already failed on some tree. A significant drag on performance is that the same rules are applied again and again to the tree, failing nearly all the time. If the failures can be memoized, the system can avoid applying all those rules.

Finally, there are many existing rewrite systems that have added tweaks to improve performance[92]. One standard technique is to partition the rules by their first condition: all rules that apply to *let* nodes are separated from other node types. For

each node in the AST, it can dispatch straight to the appropriate set of rules. The rules can be partitioned again by their second condition to further improve performance. In fact, all the rules in an extension can be transformed into a giant decision tree to quickly locate the applicable set of rules. This technique can dramatically reduce the number of rules that are attempted at each node.

8.4.3. Sausage DSLs

Sausage could be applied to itself. The tree walking combinators might be optimized by a library specific extension. The compiler pipeline might use deforestation techniques on trees to merge some rules and eliminate intermediate ASTs. An extension for the analysis system could automatically choose efficient data representations[65] for the analyses: bit vectors, sparse sets, hashtables, etc. The caveat is most of these optimizations would need to occur at runtime, not compile-time, because that is when the pipeline is constructed. A DSL can make this easier to discover at runtime by transforming the code at compile-time into easily identifiable standard forms.

The pattern matching language can be implemented as an embedded DSL, which would give Sausage many opportunities to improve performance and usability. And the pattern language would be extensible, perhaps making it easier to match against nodes with specific attributes. The analysis system can also be rewritten as a DSL modeled on other analysis frameworks. In particular, it must support cycles between attributes like

other, more powerful attribute grammars. All these could be done with a more robust implementation of Sausage.

8.4.4. Conventional Source Languages

For this technique to gain a broader audience, it must be applied to popular programming languages like Java, Python, and C++. Most popular languages today are object-oriented imperative languages, which tend to be burdened by many language features. This can be more difficult to transform effectively within the Sausage framework because the syntax trees are so verbose. The bigger problem is that a more powerful analysis system is required to understand the web of relationships between different points in the program. For example, a method may affect the state of instance or static variables beyond its current scope. To track these side-effects, the analysis must be applied to an entire class. Any transformations on a subcomponent of a class will force the class to be reanalyzed. An incremental analysis may make this more feasible.

A previous attempt to duplicate Sausage within an existing compiler framework failed. Both the Polyglot compiler for Java and Microsoft Research's Phoenix compiler for .NET appeared to be good candidates for molding into a dynamically extensible compiler, but were difficult to work with in practice. First, the ASTs are complex and verbose, which makes it difficult to match against and construct new subtrees. Second, the analysis

framework is expensive and requires a great deal of effort to extend. Third, there are many dependencies between existing compiler passes that complicate where new passes can be inserted. All these problems are surmountable, but they make it far more difficult for a programmer without compiler experience to easily write and integrate a new compiler pass.

8.5. Final Words

Programming has become too difficult. Applications have grown large and complex. Programmers use a variety of libraries with elaborate and confusing APIs. To write a reasonable application today, programmers must be experts in networking, graphics, XML, security, databases and on and on. Most programmers can not become experts in every domain. Instead, they cut and paste sample code they find on the web. They pore over stacks of programming books. They do the best they can in the face of a bewildering array of computer technology. And when they have finally understood how to properly use a complex software system, a new version or technology or paradigm sends them right back where they started.

The solution is to simplify the task of programming. Library APIs can be reduced to a clear, consistent and simple set of operations. An extensible compiler can ensure that the API is used correctly and transform the code into a more efficient implementation using the *raw*, low-level APIs. In some cases, the API can be converted into an

embedded DSL, which an extensible compiler can translate into code which uses the API. Finally, in a few cases a DSL, which does not expose any features of the source language, may be the right choice, particularly for non-programmers.

The key is to reduce the amount of tedious, low-level details required to properly use a library. Let the compiler do what it is good at: translating human readable code into efficient, low-level code. Programmers became more productive by moving to higher-level languages. The Sausage meta-compilation system allows programmers to develop new, domain-specific, high-level tools to improve their own productivity. Hopefully, this dissertation has presented a compelling argument that the need exists for DSLs and library extensions. The Sausage meta-compilation system is one attempt to answer that need.

References

[1] Aspect-Oriented Software Development.

[2] Fortran 90 Standard, 1991.

[3] ISO/IEC 23270: C# Language Specification, 2003.

[4] OpenMP: Simple, Portable, Scalable SMP Programming, OpenMP Architecture Review Board, 2005.

[5] A. V. Aho, R. Sethi and J. D. Ullman, Compiler: Principles, Techniques and Tools, Addison-Wesley, 1986.

[6] F. E. Allen, Control flow analysis, Proceedings of a symposium on Compiler optimization, Urbana-Champaign, Illinois, 1970, pp. 1-19.

[7] A. Appel, J. Davidson and N. Ramsey, The Zephyr Compiler Infrastructure, University of Virginia, 1998.

[8] A. W. Appel, Garbage Collection can be Faster than Stack Allocation, Information Processing Letters, 25 (1987), pp. 275-279.

[9] A. W. Appel, SSA is functional programming, ACM SIGPLAN Notices, 33 (1998), pp. 17-20.

[10] A. W. Appel and D. B. MacQueen, A Standard ML Compiler, Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Springer, Portland, Oregon, 1987, pp. 301-324.

[11] I. Bagrak and O. Shivers, trx: Regular-tree expressions, now in Scheme, Proceedings of the Fifth Workshop on Scheme and Functional Programming, Snowbird, Utah, 2004, pp. 21-32.

[12] I. Baxter, Transformation Technology Bibliography, Semantic Designs, Austin, TX, 1998.

[13] J. L. Bentley, Programming pearls: Little languages, Communications of the ACM, 29 (1986), pp. 711-721.

[14] R. Bird, Using circular programs to eliminate multiple traversals of data, Acta Informatica, 21 (1984), pp. 239-250.

[15] L. Birkedal and M. Welinder, Partial Evaluation of Standard ML, DIKU, University of Copenhagen, Copenhagen, 1993.

[16] G. E. Blelloch and M. Reid-Miller, Fast set operations using treaps, ACM Symposium on Parallel Algorithms and Architectures, Puerto Vallarta, Mexico, 1998, pp. 16-26.

[17] M. Blume, Dependency Analysis for Standard ML, ACM Transactions on Programming Languages and Systems, 21 (1999).

[18] M. Blume, Hierarchical Modularity and Intermodule Optimization, Computer Science, Princeton University, Princeton, 1997.

[19] M. Blume and A. W. Appel, Lambda-splitting: a higher-order approach to cross-module optimizations, Proceedings of the second ACM SIGPLAN international conference on Functional programming, ACM Press, Amsterdam, The Netherlands, 1997.

[20] R. Bodik, R. Gupta and V. Sarka, ABCD: Eliminating Array-Bounds Checks on Demand, Conference on Programming Language Design and Implementation, 2000, pp. 321-333.

[21] M. Bravenboer, A. van Dam, K. Olmos and E. Visser, Program Transformation with Scoped Dynamic Rewrite Rules, Fundamenta Informaticae, 69 (2005), pp. 1-56.

[22] M. Brukman and A. C. Myers, PPG, pp. A parser generator for extensible grammars.

[23] C. Calcagno, W. Taha, L. Huang and X. Leroy, Implementing Multi-stage Languages using ASTs, gensym, and reflection, Generative Programming and Component Engineering, 2003.

[24] C. Chambers, Staged compilation, Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, ACM Press, Portland, Oregon, 2002.

[25] B. Chin, S. Markstrum and T. Millstein, Semantic type qualifiers, SIGPLAN Notices, 40 (2005), pp. 85-95.

[26] C. Click, Combining Analyses, Combining Optimizations, Computer Science, Rice University, 1995.

[27] C. Click and K. D. Cooper, Combining Analyses, Combining Optimizations, ACM Transactions on Programming Languages and Systems, 17 (1995), pp. 181–196.

[28] W. Clinger, Macros that work, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Orlando, Florida, 1991.

[29] W. D. Clinger and L. T. Hansen, Lambda, the ultimate label, or a simple optimizing compiler for Scheme, Proceedings of the 1994 ACM conference on LISP and Functional Programming, 1994.

[30] J. Cocke, Global Common Subexpression Elimination, Symposium on Compiler Construction, 1970, pp. 850-856.

[31] J. R. Cordy, TXL - A Language for Programming Language Tools and Applications, Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications, Barcelona, Spain, 2004, pp. 1-27.

[32] T. H. Cormen, C. E. Leiserson and R. L. Rivest, Introduction to Algorithms (1st Ed.), McGraw-Hill, 2001.

[33] K. Crary, R. Harper and S. Puri, What is a Recursive Module?, SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp. 50-63.

[34] L. Damas and R. Milner, Principle type schemes for functional languages, ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, 1982, pp. 207-212.

[35] J. Darcy, JSR 269: Pluggable Annotation Processing API, Java Specification Requests, 2005.

[36] D. L. David Evans, Improving Security Using Extensible Lightweight Static Analysis, IEEE Software, 19 (2002), pp. 42-51.

[37] M. de Jonge, E. Visser and J. Visser, XT: a bundle of program transformation tools, in M. V. D. Brand and D. Perigot, eds., Workshop on Language Descriptions, Tools and Applications (LDTA'01), Elsevier Science Publishers, 2001.

[38] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B) (1990), pp. 243–320.

[39] A. v. Deursen and P. Klint, Little languages: Little maintenance?, Journal of Software Maintenance, 10 (1998), pp. 75-92.

[40] S. Dmitriev, Language Oriented Programming: The Next Programming Paradigm, JetBrains.

[41] S. Egner, SRFI 42: Eager Comprehensions, 2003.

[42] D. Engler, B. Chelf, A. Chou and S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions, Proceedings Operating Systems Design and Implementation (OSDI), 2000.

[43] D. R. Engler, Incorporating application semantics and control into compilation, USENIX Conference on Domain-Specific Languages, 1997.

[44] D. R. Engler, Interface Compilation: Steps Toward Compiling Program Interfaces as Languages, IEEE Trans. Softw. Eng., 25 (1999), pp. 387-400.

[45] D. Evans, J. Guttag, J. Horning and Y. Tan., LCLint: a Tool for Using Specifications to Check Code, in D. Wile, ed., Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering, New Orleans, USA, 1994.

[46] R. Farrow, T. J. Marlowe and D. M. Yellin, Composable attribute grammars: support for modularity in translator design and implementation, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, New Mexico, 1992, pp. 223-234.

[47] R. Fernando and M. J. Kilgard, The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley Professional.

[48] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen, DrScheme: A Progamming Environment for Scheme, Journal of Functional Programming, 12 (2002), pp. 159-182.

[49] C. Flanagan, A. Sabry, B. Duba and M. Felleisen, The essence of compiling with continuations, Programming Language Design and Implementation, 28 (1993), pp. 237-247.

[50] M. Frigo, Portable High-Performance Programs, Computer Science, MIT, Boston, MA, 1999.

[51] Y. Futamura, Partial evaluation of computation process - an approach to a compiler-compiler, Systems, Computers, Controls, 2 (1971), pp. 45–50.

[52] S. E. Ganz, A. Sabry and W. Taha, Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML, ACM SIGPLAN International Conference on Functional Programming, ACM Press, 2001.

[53] A. J. Gill and S. L. P. Jones, Cheap deforestation in practice: An optimiser for Haskell, in B. Pehrson and I. Simon, eds., Proceedings of the IFIP 13th World Computer Congress, Elsevier Science Publishers, Amsterdam, The Netherlands, 1994, pp. 581–586.

[54] J. Gosling, B. Joy, G. Steele and G. Bracha, The Java Language Specification, Addison Wesley, 2000.

[55] P. Graham, On LISP: Advanced Techniques for Common LISP, Prentice Hall, 1993.

[56] W. G. Griswold, R. Wolski, S. B. Baden, S. J. Fink and S. R. Kohn, Programming language requirements for the next millennium, ACM Comput. Surv., 28 (1996), pp. 194.

[57] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, LNCS, 477 (1990), pp. 106-116.

[58] S. Z. Guyer, Incorporating Domain-Specific Information into the Compilation Process, Computer Science, University of Texas, Austin, Austin, 2003.

[59] S. Z. Guyer and C. Lin, An annotation language for optimizing software libraries, Proceedings of the 2nd conference on Domain-specific languages, ACM Press, Austin, Texas, United States, 1999.

[60] S. Hallem, B. Chelf, Y. Xie and D. Engler, A system and language for building system-specific, static analyses, Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, ACM Press, Berlin, Germany, 2002.

[61] D. Herman and P. Meunier, Improving the Static Analysis of Embedded Languages via Partial Evaluation, International Conference on Functional Programming, 2004.

[62] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys, 28 (1996), pp. 196–196.

[63] N. D. Jones, An introduction to partial evaluation, ACM Computing Surveys, 28 (1996), pp. 480–503.

[64] S. L. P. Jones, The Implementation of Functional Programming Languages, Prentice Hall, 1987.

[65] M. Jourdan and D. Parigot, Internals and Externals of the FNC-2 Attribute Grammar System, in H. Alblas and B. Melichar, eds., Attribute Grammars, Applications and Systems, Springer-Verlag, Prague, 1991, pp. 485-504.

[66] S. Kamin, in S. Kamin, ed., DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, Paris, France, 1997.

[67] R. Kelsey, SRFI 9: Defining Record Types, 1999.

[68] R. Kelsey, W. Clinger and J. Rees, Revised^5 report on the algorithmic language Scheme, ACM SIGPLAN Notices, 33 (1998), pp. 26–76.

[69] R. Kelsey and P. Hudak, Realistic Compilation by Program Transformation, 16th Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, 1989, pp. 281--292.

[70] R. A. Kelsey, A correspondence between continuation passing style and static single assignment form, ACM SIGPLAN Notices, 30 (1995), pp. 13-22.

[71] B. Kernighan and D. Ritchie, The C Programming Language, Prentice Hall, 1978.

[72] G. Kiczales, The Art of the Metaobject Protocol MIT Press, 1991.

[73] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, An Overview of AspectJ, in J. L. Knudsen, ed., ECOOP 2001 — Object Oriented Programming 15th European Conference, Springer-Verlag, Budapest, Hungary, 2001, pp. 327-355.

[74] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtie and J. Irwin, Aspect-Oriented Programming, Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 1997.

[75] O. Kiselyov, K. N. Swadi and W. Taha, A methodology for generating verified combinatorial circuits, International Workshop on Embedded Software, Pisa, Italy, 2004.

[76] D. E. Knuth, The genesis of attribute grammars, Proceedings of the international conference on Attribute grammars and their applications, Paris, France, 1990, pp. 1-12.

[77] E. E. Kohlbecker, D. P. Friedman, M. Felleisen and B. F. Duba, Hygienic macro expansion, Proceedings of the 1986 ACM Conference on LISP and Functional Programming, 1986.

[78] D. Kolbly, Extensible Language Implementation, CS, University of Texas at Austin, Austin, 2002.

[79] D. Kranz, R. Kelsey, J. A. Rees, P. Hudak, J. Philbin and N. I. Adams, Orbit: An optimizing compiler for Scheme, Proc. SIGPLAN '86 Symp. Compiler Construction, 1986, pp. 219-233.

[80] S. Krishnamurthi, Linguistic Reuse, Computer Science, Rice University, Houston, 2001.

[81] S. Krishnamurthi, M. Felleisen and B. F. Duba, From Macros to Reusable Generative Programming, Lecture Notes in Computer Science (2000), pp. 105-120.

[82] B. Lampson, Hints for computer system design, ACM Operating Systems Review, 1983, pp. 33-48.

[83] C. A. Lattner, Architecture for a Next Generation GCC, Proceedings of the 2003 GCC Developers Summit, Ottawa, Ontario, 2003, pp. 121-134.

[84] S. Lerner, D. Grove and C. Chambers, Composing dataflow analyses and transformations, Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, Portland, Oregon, 2002.

[85] S. Lerner, T. Millstein and C. Chambers, Automatically proving the correctness of compiler optimizations, PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, 2003, pp. 231.

[86] S. Lerner, T. Millstein, E. Rice and C. Chambers, Automated soundness proofs for dataflow analyses and transformations via local rules, POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2005, pp. 377.

[87] X. Leroy, The Objective Caml System.

[88] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta and H. Takagi, OpenJIT A Reflective Java JIT compiler, OOPSLA '98, 1998.

[89] J. Merrill, GENERIC and GIMPLE: A new tree representation for entire functions, Proceedings of the 2003 GCC Developers Summit, Ottawa, Ontario, 2003, pp. 171-180.

[90] R. Milner, A theory of polymorphism in programming, Journal of Computer and System Science, 17 (1978), pp. 348-375.

[91] R. Milner, M. Tofte, R. Harper and D. B. MacQueen, The Definition of Standard ML, MIT Press, 1997.

[92] P.-E. Moreau, C. Ringeissen and M. Vittek, A Pattern Matching Compiler for Multiple Target Languages, in G. Hedin, ed., 12th Conference on Compiler Construction, Springer-Verlag, Warsaw, Poland, 2003, pp. 61-76.

[93] G. Morrisett, Compiling with Types, Computer Science, CMU, Pittsburgh, 1995.

[94] D. Novillo, Tree SSA – A New Optimization Infrastructure for GCC, Proceedings of the 2003 GCC Developers Summit, Ottawa, Ontario, 2003, pp. 181-193.

[95] N. Nystrom, M. R. Clarkson and A. C. Myers, Polyglot: An Extensible Compiler Framework for Java, 12th International Conference on Compiler Construction, Lecture Notes in Computer Science Warsaw, Poland, 2003, pp. 138-152.

[96] C. Okasaki, Purely Functional Data Structures, Cambridge University Press, Cambridge, UK, 1998.

[97] K. Olmos and E. Visser, Strategies for Source-to-Source Constant Propagation, Workshop on Reduction Strategies (WRS'02), Elsevier Science Publishers, Copenhagen, Denmark, 2002, pp. 20.

[98] J. Paakki, Attribute grammar paradigms - a high-level methodology in language implementation, ACM Computing Survey, 27 (1995), pp. 196-255.

[99] S. L. Peyton Jones and A. L. M. Santos, A transformation-based optimiser for Haskell, Science of Computer Programming, 32 (1998), pp. 3-47.

[100] S. L. Peyton Jones, A. Tolmach and T. Hoare, Playing by the rules: rewriting as a practical optimisation technique in GHC, Haskell Workshop, 2001, pp. 203-233.

[101] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[102] T. J. P. a. R. W. Quong, ANTLR: A Predicated-LL(k) Parser Generator, Software -- Practice and Experience, 25 (1995), pp. 789-810.

[103] A. D. Robison, Impact of economics on compiler optimization, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, ACM Press, Palo Alto, California, United States, 2001.

[104] B. K. Rosen, M. N. Wegman and F. K. Zadeck, Global value numbers and redundant computations, 15th ACM Symposuim on Principles of Programming Languages, ACM, 1988, pp. 12-27.

[105] G. v. Rossum and J. Fred L. Drake, The Python Language Reference Manual, Network Theory Ltd., 2003.

[106] A. Sabry, The Formal Relationship between Direct and Continuation-passing Style Optimizing Compilers: A Synthesis of Two Paradigms, Computer Science, Rice University, Houston, 1994.

[107] A. Sabry and M. Felleisen, Is Continuation-Passing Useful for Data Flow Analysis?, Programming Language Design and Implementation, 1994, pp. 1-12.

[108] D. Sarkar, O. Waddell and R. K. Dybvig, A nanopass infrastructure for compiler education, ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming, 2004, pp. 212.

[109] S. Schupp, D. Gregor, D. R. Musser and S.-M. Liu, User-Extensible Simplification and Type-Based Optimizer Generators, Proceedings of the 10th International Conference on Compiler Construction, Springer-Verlag, 2001.

[110] Z. Shao and A. W. Appel, A type-based compiler for standard ML, ACM SIGPLAN 1995 conference on Programming language design and implementation, La Jolla, California, 1995, pp. 116-129.

[111] O. Shivers, The semantics of Scheme control-flow analysis, Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, New Haven, Connecticut, 1991, pp. 190-198.

[112] J. G. Siek and A. Lumsdaine, The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra, International Symposium on Computing in Object-Oriented Parallel Environments, 1998, pp. 59–70.

[113] M. Sperber, F. Solsona, D. V. Horn and D. Kolbly, SRFI: Scheme Request For Implementation.

[114] G. L. Steele, Common LISP the Language, Second Edition, Digital Press, 1984.

[115] J. M. Stichnoth, Generating Code for High-Level Operations through Code Composition, Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[116] J. M. Stichnoth and T. Gross, Code Composition as an Implementation Language for Compilers, USENIX: Conference on Domain-Specific Languages, 1997, pp. 119–132.

[117] J. M. Stichnoth, D. O'Hallaron and T. Gross, Generating communication for array statements: Design, implementation, and evaluation, Journal of Parallel and Distributed Computing, 21 (1994), pp. 150-159.

[118] R. E. Strom and S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE Transactions on Software Engineering, 12 (1986), pp. 157-171.

[119] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, Reading, MA, USA, 1994.

[120] W. Taha, Multi-Stage Programming: Its Theory and Applications, Computer Science, Oregon Graduate Institute of Science and Technology, 1999.

[121] W. Taha and T. Sheard, MetaML and multi-stage programming with explicit annotations, Theoretical Computer Science, 248 (2000), pp. 211-242.

[122] D. Tarditi, G. Morrisett and P. Cheng, TIL: A Type-Directed Optimizing Compiler for ML, Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, 1995, pp. 181-192.

[123] P. Thiemann, Aspects of the PGG System: Specialization for Standard Scheme, Partial Evaluation: Practice and Theory (DIKU International Summer School 1998) (1998).

[124] D. Thomas, D. Hansson, L. Breedt, M. Clark, T. Fuchs and A. Schwarz, Agile Web Development with Rails : A Pragmatic Guide, Pragmatic Bookshelf, 2005.

[125] M. Torrens, R. Weigel and B. Faltings, Java Constraint Library: bringing constraint technology on the Internet using the Java language, Constraints and Agents: Papers from the 1997 AAAI Workshop, AAAI Press, Menlo Park, California, 1997, pp. 21-25.

[126] M. T. Vandevoorde, Exploiting specifications to improve program performance, CS, MIT, Cambridge, 1993.

[127] M. T. Vandevoorde and J. V. Guttag, Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity, Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, New Orleans, Louisiana, 1994, pp. 121-127.

[128] T. L. Veldhuizen, Blitz++: The library that thinks it is a compiler, in E. Arge, A. M. Bruaset and H. P. Langtangen, eds., Modern Software Tools for Scientific Computing, Birkhauser (Springer-Verlag), Boston, 1997.

[129] T. L. Veldhuizen, C++ templates as partial evaluation, in O. Danvy, ed., ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical report BRICS-NS-99-1, University of Aarhus, San Antonio, Texas, 1999, pp. 13–18.

[130] T. L. Veldhuizen, Using C++ template metaprograms, C++ Report, 7 (1995), pp. 36-43.

[131] T. L. Veldhuizen and D. Gannon, Active libraries: Rethinking the roles of compilers and libraries, Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98), SIAM Press, 1999.

[132] Versant, FastObjects at *http://www.versant.com/products/fastobjects*.

[133] E. Visser, Stratego: A language for program transformation based on rewriting strategies, Rewriting Techniques and Applications (RTA'01), 2051 (2001), pp. 357-361.

[134] E. Visser, A survey of rewriting strategies in program transformation systems, in B. Gramlich and S. Lucas, eds., Workshop on Reduction Strategies in Rewriting and Programming (WRS'01) Elsevier Science Publishers, Utrecht, The Netherlands, 2001.

[135] E. Visser, Z.-e.-A. Benaissa and A. Tolmach, Building program optimizers with rewriting strategies, Proceedings of the third ACM SIGPLAN international conference on Functional programming, ACM Press, Baltimore, Maryland, United States, 1998.

[136] O. Waddell, D. Sarkar and R. K. Dybvig, Robust and effective transformation of letrec, Workshop on Scheme and Functional Programming 2002.

[137] P. Wadler, Deforestation: Transforming Programs to Eliminate Trees, Theoretical Computer Science, 73 (1990), pp. 231-248.

[138] R. C. Waters, Automatic transformation of series expressions into loops, ACM Trans. Program. Lang. Syst., 13 (1991), pp. 52-98.

[139] M. N. Wegman and F. K. Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems, 13 (1991), pp. 181-210.

[140] J. Wielemaker, SWI-Prolog Reference Manual, University of Amsterdam, The Netherlands, Amsterdam, 1997.

[141] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam and J. L. Hennessy, SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers, SIGPLAN Notices, 29 (1994), pp. 31-37.

[142] M. J. Wolfe, High Performance Compilers for Parallel Computing, Addison Wesley, Reading, Mass., 1996.

[143] A. K. Wright and R. Cartwright, A Practical Soft Type System for Scheme, ACM Transactions on Programming Languages and Systems, 19 (1997), pp. 87-152.

[144] A. K. Wright and B. F. Duba, Pattern Matching for Scheme, Rice University, Houston, TX, 1995.

[145] C. G. Wu, Modeling Rule-Based Systems with EMF, 2004.

[146] T. Yamada, Confluence and Termination of Simply Typed Term Rewriting Systems, in A. Middeldorp, ed., Rewriting Techniques and Applications: 12th International Conference, RTA 2001, Springer-Verlag GmbH, Utrecht, The Netherlands, pp. 338

Appendix A. Scheme

The Scheme programming language is a simple, latently-typed, higher-order dialect of Lisp. Though many are put off by its lack of rigid syntax and language features, these are precisely its strengths. Many conventional language features can be simulated by layering a library on top of the core language. Therefore, Scheme is explicitly designed to be an extensible language. The introduction to the Scheme report describes this idea eloquently:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is extensible enough to support most of the major programming paradigms in use today.

Unlike many languages, the core Scheme language is extremely small, since it is merely an implementation of the untyped lambda calculus. Most conventional language features are implemented using macros and libraries. The following is a brief description of some features found in Scheme that are relevant to this dissertation. Of course, the full range of language features that can be implemented in Scheme is unbounded, from object-oriented, to logic, to constraint-based language features.

Syntax

Syntax is a superficial concern for programming languages; programmers can easily adapt to different syntax forms. Extensibility, however, is a critical feature for language

syntax. To promote extensibility, Scheme uses S-expressions just as does Lisp: a prefix notation bound within parentheses. Therefore, every expression is of the form "(operator operands ...)". This allows programs to be manipulated as lists, which is the primary data structure used in Scheme and Lisp. Conveniently, the reserved syntactic terms in Scheme are also in prefix form, i.e. *define, lambda, set!.* In a sense, functions and "syntax" are indistinguishable. This is why Scheme is an extensible language.

Macros

Scheme macros are a rewriting preprocessor for S-expressions. It is a set of patterns and replacements that are applied to source code until it reaches fixed point. It is quite easy to write macros that do not terminate, though, in practice, this problem is easy to avoid. Scheme macros do not allow one to execute arbitrary Scheme code as in Lisp. Instead, the patterns are replaced in a more controlled manner. The main benefit of this style is hygiene, meaning new variable names are guaranteed not to inadvertently capture existing variables. Scheme macros can only match s-expressions where a specific identifier is in the operator position.

```
(define-syntax fib-macro
  (syntax-rules ()
    ((fib-macro) 1)
    ((fib-macro 1) 1)
    ((fib-macro 1 1 args ...)
     (+ (fib-macro 1 args ...) (fib-macro args ...)))))

(fib-macro 1 1 1 1 1)
➔
(+ (+ (+ (+ (+ 1 1) 1) (+ 1 1)) (+ (+ 1 1) 1))
   (+ (+ (+ 1 1) 1) (+ 1 1)))
```

Figure A-1. Syntax-macro example

The example in Figure A-1 demonstrates how macros can be used as a simple rewrite system. Since macros cannot perform arithmetic, the number of 1's in the argument list to *fib-macro* is the argument to the Fibonacci rules. If there are zero or one 1's in the argument list, Fibonacci returns a 1. Otherwise, the last rule matches the first two 1's and collects the remaining, if any, 1's into the pattern variable *args*. It replaces the pattern with the sum of two different *fib-macro* rules. These two calls to *fib-macro* are expanded recursively to generate the large summation shown for Fibonacci of 5. Scheme's syntax-macros have a few more features, but this example illustrates the main idea.

Lists

The primary data structure in Scheme is the *cons* cell, a structure with two untyped slots. A collection of cons cells can be used to build many different data structures, but it is most commonly used to construct singly-linked lists. These linked lists are ubiquitous in Scheme code; however, they can best be described as the lowest common

denominator among more complex data structures. Large Scheme programs will undoubtedly use trees, hashtables, sets, etc. wherever appropriate. Linked lists can be used to glue together disparate data structures. It is the *lingua franca* of data structures.

The two most basic operations on lists are the *map* and *fold* operations. A *map* operation applies a function to each element of the list and returns a new list containing the results, in order. The order in which it operates over the list's elements is undefined, giving the compiler more flexibility to implement it efficiently. A *fold* operation recursively applies a function to an element and the result of the previous application, beginning with an initial value. For example, "(fold + 0 '(1 2 3))" will sum the values in the list by generating the following code: "(+ (+ (+ 0 1) 2) 3)". To explicitly specify the order of evaluation, implementations typically provide the functions *fold-left* and *fold-right*, which operate over the input list either left or right element-wise, respectively. Most other list operations can be described in terms of these two basic functions.

Multiple Values

Though functions typically return a single value, Scheme allows functions to return more than one value in a controlled manner. The function *values* declares that more than one value will be returned, i.e. "(values 1 2)" returns two values. Since the caller is expecting only one value, one must use the function *call-with-values* to explicitly capture more than one return value. The function that returns multiple values is

wrapped in a thunk, a closure with no arguments. The code that uses the return values is wrapped in a function that accepts the expected number of arguments.

```
(define (partition pred input)
  (if (null? input)
      (values '() '()) ;; base case
      (call-with-values
        (lambda () (partition pred (cdr input)))
        (lambda (in out)
          (if (pred (car input))
              (values (cons (car input) in) out)
              (values in (cons (car input) out)))))))
```

Figure A-2. Using multiple values

The code in Figure A-2 will split a list into two using a predicate to filter the elements. It recursively builds two different lists by appending each element to either list. It then passes both lists back as multiple return values. The *call-with-values* function takes two arguments: a thunk that produces multiple return values and a closure that accepts the correct number of arguments. Though it is verbose, macros can make this easier to use.

Fluids

A fluid is a data object used to simulate a dynamically scoped variable, similar to a special variable in Lisp. In Scheme, as in many other languages, variables are accessed by lexical scoping. This means that the use of a variable refers to the last binding of that variable in the text of the program. A dynamic variable, on the other hand, tracks the last binding in the dynamic execution path of the program. Since this is implemented by a library in Scheme, it is much more verbose to use than in Lisp.

```
(define x (make-fluid 1))

(define (f) (fluid x))

(define (g)
  (let-fluid x 2
    (lambda () (f))))

(define (h)
  (let ((x 2))
    (f)))

(g)
➔ 2

(h)
➔ 1
```

Figure A-3. Dynamically vs. Lexically scoped variables

The example in Figure A-3 illustrates the difference between lexically and dynamically scoped variables. A global variable *x* is bound to a fluid structure initialized to 1. The current value of *x* can be accessed with the function *fluid*. In *g* the dynamic variable *x* is bound to 2. When *f* is called, it retrieves the last value assigned to *x*, which is 2. In *h* a local variable *x* is bound to 2. When *f* is called again, it retrieves the value from the global variable *x*, not the local variable bound in *h*, thus returning 1. With fluids, one can make a global change to a variable in a controlled manner.

Continuations

A continuation is a function that leads to the *next* step in a computation. The nested addition in Figure A-4 first adds 2 and 3, and then adds that sum to 1. It can be rewritten in *continuation passing style* (CPS) with a modified + operation. In this case, it

adds two numbers and calls the function (third argument) with the sum. This function, called the continuation, makes explicit the control flow in a program. In Scheme, the function *call-with-current-continuation* (or *call/cc* for short) can access the continuation that is generally hidden in a normal program. Since the continuation is just a function, it can be passed around as a first-class value and it can be called more than once. By manipulating this continuation, programmers can manipulate the control flow of their program.

```
(+ 1 (+ 2 3))

In CPS form:
(+ 2 3 (lambda (v) (+ 1 v k)))

Iterator with continuations:
(define (iterator n k)
  (let next ((i 0))
    (if (> i n)
      i
      (next (call/cc ;; entry point for count
             (lambda (yield) ;; jump to count's let
               (k (cons i (lambda () (yield (+ 1 i))))))))))))

(define (count n)
  (let ((v (call/cc  ;; entry point for iterator
             (lambda (return)
               (iterator n return)))))
    (if (pair? v)
      (begin (display (car v))
             ((cdr v)))  ;; jump to the loop within iterator
      (display "done"))))

(count 9)
➔
0123456789done
```

Figure A-4. Continuations

The iterator example in Figure captures and manipulates the normal control flow in the two functions *iterator* and *count*. When *count* calls *iterator*, it gets back a cons cell that contains the initial counter value and a continuation that jumps back into the iterator body (noted in the code comments). After *count* prints the current value, it calls the continuation (wrapped in a thunk). This jumps straight back into the *iterator* right where the continuation is caught. When *iterator* calls the continuation k, it jumps back into *count*'s let expression with a new cons cell. This continues until iterator fails to return a cons cell. Though continuations can get complicated, they are enormously useful. All known control flow operations in a single threaded program can be implemented using continuations: backtracking, coroutines, exception handling, etc.