Type-Safe Domain-Specific Code Graph Construction Using Scala

Type-Safe Domain-Specific Code Graph Construction Using Scala

By
Simon C. Broadhead B.A.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2015) McMaster University

COMPUTER SCIENCE Hamilton, Ontario

TITLE: Type-Safe Domain-Specific Code Graph Construction Using Scala

AUTHOR: Simon C. Broadhead B.A.Sc. (McMaster University)

SUPERVISOR: Dr. Christopher K. Anand

NUMBER OF PAGES: viii, 70

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

As an extension to the ongoing Coconut (*COde COnstructing User Tool*) project at McMaster University, we present a Scala library for constructing type-safe domain-specific languages that uses Coconut's hypergraph-based representation of code (*code graphs*) as the intermediate representation. Our library automatically produces strongly typed, deeply embedded DSLs given only a minimal specification of the DSL's value types and primitives. We make extensive use of path-dependent types and implicit argument lookup to construct a type-safe interface on-the-fly, rather than requiring DSL designers to explicitly create a type-safe interface.

In this thesis we present our library and demonstrate its utility as both a general-purpose DSL framework and as a suitable platform for continued research on the Coconut project. By giving practical examples of the library in use, we demonstrate both its general utility, and the striking swiftness with which new DSLs may be constructed, especially compared to the previously laborious Haskell DSLs of Coconut.

Acknowledgments

Thank you to my advisor Dr. Christopher Anand for all of the support, assistance, and opportunities he has offered me over the past six years. Thank you to my parents, Lori and Steve Broadhead, for supporting me in everything I have ever done. And thank you to my amazing girlfriend Kirstie Gooding for her love and support throughout the past two years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The Coconut project being researched at McMaster University is a collection of related tools and libraries, primarily written in the Haskell programming language, designed to aid in the generation of efficient code from provably correct domain-specific models [KAC06, ACK+04, AK07b, AK07a]. These models are often specified in the form of type-safe domain-specific languages (DSLs) embedded within Haskell. See Chapter 2.

Although highly effective (see [AK09]), these tools have been crafted in a somewhat ad-hoc fashion over the course of many years. The Glasgow Haskell Compiler (GHC), itself an active research project, has evolved greatly since the inception of Coconut, and as a result, much of the old code that would be useful today is no longer compilable. Although Coconut's aging code base is becoming increasingly obsolete, much of its functionality is still highly useful.

Rather than attempt to patch up the aging code (an assuredly temporary measure) or rewrite it from scratch in Haskell, we view this as an opportunity to look at the goals of Coconut from a completely new perspective. The Haskell code base exists primarily, among other things, as an academic research tool. There have been efforts to take it even further in that direction, such as [Kah11], which brings Coconut's core data structures into the realm of automated proof checking. This has powerful implications for produc-

ing correct code, but lacks mainstream applicability and does not promote rapid experimentation and implementation.

The research presented in this thesis represents an effort to realize the goals of Coconut from a practical standpoint rather than a purely academic or theoretical one. Using the emerging programming language Scala, our goal is to provide a reusable toolkit for rapid development of, and experimentation with, domain-specific languages suitable for practical use, while also easily facilitating the kind of research tasks Coconut specializes in.

The Scala programming language is a general-purpose language that compiles to Java Virtual Machine (JVM) bytecode [OAC+04]. Its type system is a generalization of the underlying Java type system, and so features the full object-oriented class system of that language—however, it adds many extra features onto Java's type system, such as co/contravariant generic parameters, class composition via traits (mixins), first class singletons, and, most importantly, dependent types. The balance between extreme type safety and platform ubiquity (due to being hosted by the JVM) makes Scala an obvious choice in trying to bring the Coconut project's specialized functionality and inherently type-safe interface into the realm of general mainstream applicability.

1.2 Prior Work and Contribution

The Yin-Yang library for Scala [JSS+14] provides a macro-based translator to provide a deep embedding for a shallowly embedded DSL interface. See Chapter 2 for an overview of deep and shallow embedding.

The Delite project is, like Coconut, a performance-oriented library focused on efficient code generation. Delite is a compiler framework and runtime that facilitates the implementation of highly performant, parallel domain-specific languages from within Scala [BSL+11, RSL+11, SRB+13].

Our research is intended to bring the past work of the Coconut project to the Scala language, primarily in order to make use of its extremely powerful type system. Its goal is to provide a reusable platform for the rapid development and use of deeply embedded domain-specific languages. The underlying data structures, *code graphs* (see Chapter 3) are well-studied, and

by using them explicitly within our DSL library, we facilitate the furtherance of the interests of Coconut.

1.3 Organization

The remainder of this thesis is organized as follows:

- Chapter 2 looks at embedded domain-specific languages in general. It serves to motivate our desire to use DSLs and demonstrates ways in which type-safe DSLs can prevent problems that would otherwise occur in untyped environments
- Chapter 3 gives a brief overview of the Coconut project, code graphs, and the strongly typed domain-specific language Coconut uses for code graph construction. This lets us clearly see later how the syntax and semantics of the existing tools can easily be adapted (and extended) in the Scala environment.
- Chapter 4 presents our Scala library for constructing domain-specific code graphs using rapidly developed DSLs.
- Chapter 5 relates past Coconut work to our modern Scala code base and demonstrates the implementation of the exponential function—a function typical of those implemented using Coconut—using a rapidly constructed DSL.
- Chater 6 presents the conclusions of this thesis.

Chapter 2

Embedded Domain-Specific Languages

2.1 Domain-specific Languages

Domain-specific language (DSL) describes a class of languages whose semantics are strictly relevant to a specific domain. Such languages often partially or completely capture the conventions and jargon of their respective domains, and allow domain experts to use familiar words and idioms to interact with a system without being burdened by the semantic overhead of a traditional API bound to a general-purpose language. For example, a typical configuration DSL may have only simple keywords for configuring each facet of the subject domain and nothing else. The same configuration expressed in a general-purpose language would necessarily be subject to at least some overhead from the language, such as the need for string literals and the lack of domain-specific keywords. This overhead can obscure domain semantics and increase the programmer's cognitive load.

There is clearly a trade-off present when it comes to capturing domain-specific semantics: a general-purpose language offers many features (i.e., control flow, complex function evaluations) not present in a typical domain-specific language, while a domain-specific language has a much lower complexity floor and can be more easily and reliably used by non-programmers.

Varying audiences may require varying levels of complexity (and expressivity) from domain-specific programming interfaces. Advanced users, such as

those with a strong programming background, may desire full imperative control over their domain operations and so benefit from having access the power of a general-purpose language, while others may require only basic access to a fixed set of features and would benefit from the simplicity of a DSL.

Luckily, the goals of general-purpose languages and domain-specific languages are not inherently mutually exclusive. Many modern programming languages have syntaxes sparse and extensible enough to support establishing a DSL directly within the language without imposing too much semantic or syntactic overhead of its own. A DSL hosted inside a general-purpose language in this way is called an *embedded domain-specific language*.

An embedded DSL is typically a library written in a suitable host language whose functionality is expressed through a system of domain-specific primitives that partially obscure or abstract the underlying semantics of the language. These primitives form a sub-language embedded within the host language, allowing access to the full power of the host language while still providing rich domain-specific abstractions.

```
// Using Python
servers['MyServer']['HOSTNAME'] = '10.0.0.1'
servers['MyServer']['PORT'] = 7080
servers['MyServer']['ALIASES'] = ['foo', 'bar']

// Using a domain-specific language
server MyServer {
    at 10.0.0.1:7080
    aliases foo bar
}
```

Listing 2.1: A short, contrived example illustrating how a traditional domain-specific language can eliminate much of the syntactic and semantic overhead (such as string literals and dictionary semantics) of a general purpose language like Python. Also note the use of the canonical *`hostname`*`:`*`port`* syntax, which is familiar to domain experts.

Real-Life Example: Sinatra

The HTTP protocol is surrounded by many domain-specific terms and conventions, which makes it a prime subject candidate for a domain-specific language. One compelling aspect of HTTP is its use of *verbs*, such as `GET` and `POST`, as units of communication. Such clear divisions of work and unambiguous terminology leads to obvious abstractions of the sort that makes a good embedded DSL.

The Sinatra library for Ruby provides (among other things) a DSL for creating HTTP applications. Ruby is a natural choice for creating an embedded DSL because it has a clean and extensible syntax—core language constructs are effectively indistinguishable from user-defined ones, and there is very little syntactic noise.

Listing 2.2 shows a minimal Sinatra application. The only construct present in the code is a `get` block, which is not a primitive construct in Ruby. Indeed, at first glance, it may not be clear exactly how the `get` construct is implemented in Ruby, but the semantics are fairly self-evident to anyone familiar with even the most basic HTTP applications (i.e., a domain expert).

```
require 'sinatra'

get '/' do
  'Hello world!'
end
```

Listing 2.2: A minimal Sinatra application.

That distinction nicely highlights the typical characteristics of embedded DSLs: their syntaxes should be free from overhead generated by the host language and they should hide as many of the irrelevant semantics of the host language as possible while cleanly and concisely capturing the semantics of the subject domain.

In the case of Sinatra, `get` is a domain-specific primitive that captures the semantics of one of the core functions of the library—routing a `GET` request—while completely hiding the underlying Ruby semantics (like the fact that an implicitly constructed `Application` object is having its `routes` variable mutated behind the scenes, none of which is relevant to the domain).

2.2 Shallow Embedding and Deep Embedding

Sinatra is implemented by mapping the domain-specific primitives (such as `get` and `post`) directly onto corresponding semantics within Ruby (such as mutating the underlying list of HTTP routes). This kind of embedding—mapping domain primitives directly onto corresponding host language semantics—is known as a *shallow embedding*. Figure 2.1 gives a simple overview of the way a shallowly embedded domain-specific interacts with its host language.

Many utility DSLs are implemented this way, as they provide a natural alternative to clumsy, verbose, or otherwise inhospitable interfaces for many tasks. This type of DSL has many practical applications, but for use-cases that extend beyond wrapping existing functionality in a nice syntax, it is likely to be impractical or impossible to map the domain primitives directly

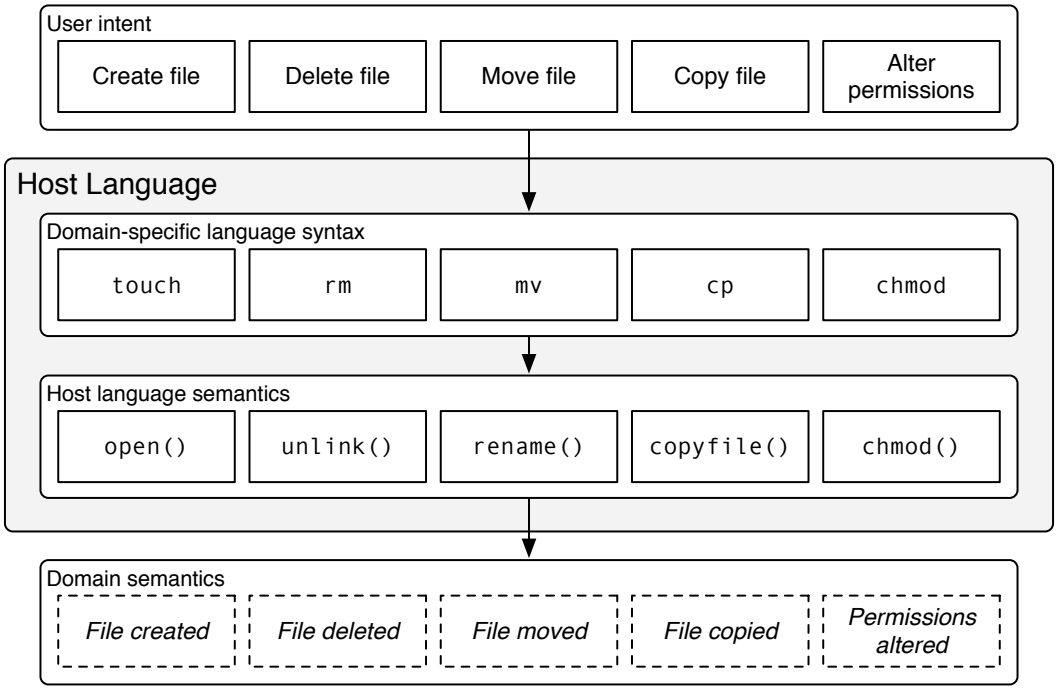


Figure 2.1: A simple overview of a shallowly embedded DSL.

onto the host language semantics. For these cases, a *deep embedding* may be used, where the domain primitives are not mapped directly onto language semantics, but rather onto intermediate, often recursive data structures—i.e., an abstract syntax tree (AST)—which can then be interpreted. Figure 2.2 (page 17) shows how an embedded DSL interacts with its host language. Note that unlike a shallow embedding, a deep embedding allows for the domain-specific constructs to be stored, inspected, and transformed, rather than only directly executed.

This kind of embedding allows for a much richer language representation, at the expense of a potentially more hostile user experience—rather than using familiar programming constructs with real-time side effects, users must (directly or indirectly) manipulate data structures.

Languages that support algebraic data types and pattern matching, such as Haskell, are well suited to deeply embedded DSLs; algebraic data types provide a seamless way to integrate the abstract syntax tree (AST) of the DSL into the type system of the host language, while pattern matching provides a seamless way to decompose and evaluate these structures.

2.3 FILEDSL: A Simple DSL in Haskell

In this section we present a simple deeply embedded DSL called FILEDSL for performing Unix-style file operations in Haskell. It serves to reify the concepts talked about so far in this chapter. This section is written in the Literate Haskell style, meaning that this section is interspersed with the entire executable Haskell code for this DSL.

```
module FILEDSL where

import Control.Monad (mapM_, mplus)
import Data.Char (ord, chr)
```

2.3.1 Data Types

We start by defining our data types. We use *Filename* as a semantic alias for *String*.

```
type Filename = String
```

Permission encodes the 8 possible octal values used by the Unix permission system. The *toOctal* function converts it to its corresponding octal digit. See Section 2.4 for a short description of how `chmod` works.

```
data Permission
   = Deny  | Execute     | Write     | WriteExecute
   | Read  | ReadExecute | ReadWrite | All
  deriving (Show, Enum)

toOctal    :: Permission → Char
toOctal p  = chr $ (fromEnum p) + (ord '0')
```

Perms encodes an individual file's permissions.

```
data Perms = MkPerms
   { p_owner  :: Permission
   , p_group  :: Permission
   , p_world  :: Permission
   }
  deriving (Show)
```

Since this is a deeply embedded DSL, we need to define a type to encode our AST. Since this is such a simple DSL, our AST will just be a usual Haskell list of operations, which are represented by the *FileOp* type. We alias the list and call it *FileAST* to obscure Haskell's underlying list semantics from our DSL's syntax.

```
data FileOp
   = CreateFile  Filename
   | DeleteFile  Filename
   | CopyFile    Filename  Filename
   | MoveFile    Filename  Filename
   | ChMod       Perms     Filename
  deriving (Show)
type FileAST = [FileOp]
```

2.3.2 Execution

The *execFileOp* function executes a single *FileOp*.

```
execFileOp :: FileOp → IO ()
execFileOp (CreateFile filename) =
    void $ spawnProcess "touch" [filename]
execFileOp (DeleteFile filename) =
    void $ spawnProcess "rm" ["-f", filename]
execFileOp (CopyFile from to) =
    void $ spawnProcess "cp" [from, to]
execFileOp (MoveFile from to) =
    void $ spawnProcess "mv" [from, to]
execFileOp (ChangeMode (MkPerms o g w) filename) =
    let permString = (toOctal o) : (toOctal g) : (toOctal w) : "" in
    void $ spawnProcess "mv" [permString, filename]
```

2.3.3 DSL Syntax

We have data structures representing our DSL, but their names do not match the familiar Unix operations, and they do not comprise complete ASTs by themselves, so we provide some wrappers with the familiar names.

Note that although *FileAST* is actually just an alias for a list, we choose not to admit that fact here and interact with it only through its monad interface (*return*). This keeps the semantics of our AST separate from Haskell's list semantics—they are identical, but only coincidentally. It is an implementation detail.

```
touch, rm       :: Filename → FileAST
mv              :: Filename → Filename → FileAST
chmod           :: Perms    → Filename → FileAST

touch           = return ∘ CreateFile
rm              = return ∘ DeleteFile
mv    x y       = return $ MoveFile x y
cp    x y       = return $ CopyFile x y
chmod p f       = return $ ChMod p f
perms o g w     = MkPerms o g w
```

We have chosen to obscure the fact that the AST is a list by never requiring the user to admit it in practice. All of the primitive operations return ASTs already, so constructing lists manually is never necessary. The other fun-

damental operation we must support is the joining of two ASTs to indicate operation sequencing; to this end, the &&& operator is a synonym for list concatenation that has domain-specific semantics (i.e., *do these file operations in sequence*) rather than general Haskell semantics (i.e., *concatenate these two lists*).

We use *mplus* rather than its alias $+\!\!+$ (in this context) simply because it is more general.

```
(&&&) = mplus
```

The *fileDSL* function is the entry-point to our DSL. It executes a *FileAST*.

```
fileDSL :: FileAST → IO ()
fileDSL = mapM_ execFileOp
```

2.3.4 Shallow Embedding For Free

By making a "shallow" version of each primitive—that is, one which simply evaluates to the underlying Haskell semantics of the domain primitive rather than to a structure representing that primitive—we can derive a shallowly embedded variation of our DSL.

The fact that it is deeply embedded behind the scenes is merely an implementation detail; if these were the only functions exported from this module then this DSL would be, for all intents and purposes, shallowly embedded.

```
touch'       = fileDSL ∘ touch
rm'          = fileDSL ∘ rm
mv'    x y   = fileDSL $ mv    x y
cp'    x y   = fileDSL $ cp    x y
chmod' p f   = fileDSL $ chmod p f
```

2.3.5 Example

Note that *exampleDeep* executes the DSL after each of the DSL primitives has already been evaluated (and turned into an AST) by passing it to *fileDSL*, while *exampleShallow* is a sequential **do** block that executes each primitive as it is evaluated. Both functions produce the same effect.

```
exampleDeep :: IO ()
exampleDeep = fileDSL $
  mv    "data"             "data.bak" &&&
  cp    "output/data"      "data"     &&&
  chmod (perms All Read Read) "data"  &&&
  rm    "output/data"

exampleShallow :: IO ()
exampleShallow = do
  mv'    "data"             "data.bak"
  cp'    "output/data"      "data"
  chmod' (perms All Read Read) "data"
  rm'    "output/data"
```

2.4 Type Safety

FILEDSL makes heavy use of data types due to its deeply embedded nature—all of our syntax has a structural counterpart in its corresponding *FileOp* constructor. Since Haskell is a statically-typed language, it detects type errors at compile-time; that means if we can encode semantic constraints within the data types used by our DSL, then we can make the host compiler enforce those constraints automatically. For an example of this, consider the `chmod` primitive in FILEDSL.

`chmod` is a UNIX command that changes the permissions of a given file. Its basic syntax (where `$` represents a Bash prompt) is as follows:

```
$ chmod 777 filename
```

The literal 777 is an octal number representing the level of access to the file granted to users of the system. Each of the three 3-bit octal digits holds a bitfield indicating the access levels of the file's owner, the file's group, and other users respectively. Within each digit, the three bits (starting from the most significant bit) represent read access, write access, and execute access respectively. One very commonly used file mode is 744, which means "file owner has full access, file group and other users have read-only access". 7 corresponds to full access, since all three access bits are 1, while 4 corresponds to read-only access since only the most significant bit, corresponding to read access, is 1.

This notation is a convenient representation for the computer; it is a concise way to fit a lot of information into a small number of bits. For humans, however, working with it requires specialized technical skills (like mentally visualizing and manipulating octal bitfields) as well as remembering arbtirary conventions about the syntax (like which bit refers to which access type). To the uninitiated reader, the syntax offers no clues as to its meaning. It provides no indication as to what range of data may be valid as an argument, or as to the meaning of each symbol.

In capturing the semantics of `chmod`, we did not simply map the *syntax* of the UNIX command onto our DSL. If we had done that, our `chmod` primitive would have had to accept an octal literal. Octal literals in Haskell are of the form `0o777` and are of type `Int`, so our *FileOp* datatype would have looked something like this:

```
data FileOp
   = CreateFile  Filename
   | DeleteFile  Filename
   | CopyFile    Filename  Filename
   | MoveFile    Filename  Filename
   | ChMod       Int       Filename
```

This would simplify things a bit—there would be no more need for the *toOctal* function. In fact, the *Permission* and *Perms* types could go away entirely. However, before we go ahead and make this change, we should first make sure we are not inviting any catastrophes.

Consider a user who is familiar with `chmod`, but who is not as familiar with Haskell. This user may not know about the seldom-used `0o777` octal syntax and attempt to call the `chmod` primitive with a decimal literal, as in `chmod 744 "myfile"`. (Alternatively, this user may be extremely familiar with Haskell and less so with `chmod`, neglecting to realize that the argument is supposed to be octal at all.) In this case, the argument would be interpreted incorrectly as the octal value 1350_8—an invalid argument under our semantics. In order to more closely match the UNIX syntax, we could change our `chmod` to accept decimal literals that *look like* the intended octal value (meaning $744 \equiv 744_8$) and convert them internally, but that only further obfuscates the true meaning of the argument. Aside from that, this solution only changes the site of the problem; when a seasoned veteran of both UNIX and Haskell reasonably assumes that the semantically octal argument can

be supplied by an octal literal and passes `0o744`, it will be interpreted as the decimal number 484_{10}, which, interpreted literally in octal, is a bogus number.

Internally, the decimal and octal representations are indistinguishable. If the user uses the wrong one, we cannot even give them a good error message; the best we can do is crash at runtime and tell them that they supplied an invalid file mode, and to make sure they are using the right encoding. In fact, if the misinterpreted argument is itself a valid file mode, then all is lost; we cannot even detect that anything wrong has occurred, and the user's files will become incorrectly permissioned, perhaps without anyone noticing. The implications of this are both numerous and obvious.

The problem is that only a very small subset of syntactically possible inputs to `chmod` are valid, but the user is nevertheless forced to select from the entire range. Indeed, regardless of whether we consider the argument to be an Int—decimal or octal—or a String, only $8^3 = 512$ values are semantically relevant. The rest are inherently malformed and must be rejected. It is up to the user to ensure that they do not supply one of these nearly limitless invalid values. This raises the question: *Why include those values in the argument domain at all?*

Instead of using the primitive Int type for our representation of permissions, we have constructed a type, Perms, with exactly the same cardinality as the set of valid file modes. We injectively map Perms onto the semantically-relevant subset of String using *toOctal*, and the result is a representation of permissions that is both provably[1] equivalent to the traditional literal-based notation and definitionally *incapable* of providing an invalid file mode. We did not have to do anything to achieve this level of safety, other than define our datatypes and ensure that they cannot contain semantically invalid data. Haskell's type checker enforces the semantic contract for us.

This highlights one of the major strengths of embedding a DSL in a strongly typed language like Haskell; were the same DSL to be implemented in Ruby, which has no compile-time type checking (or indeed, any "compile-time" at all), the `chmod` primitive would *have* to be prepared to accept any value at all. It would be up to us to either meticulously inspect the argu-

[1] By exhaustion, if the reader is bored.

ment to ensure it meets all of our preconditions (allowing us to fail gracelessly at runtime zero or more times in the future), or to place the onus of type-correctness on the user and let incorrect inputs propagate internally (allowing us to fail with even less grace or predictability). The strong, static nature of Haskell's type system takes this unbounded sequence of graceless run-time failures and collapses them all into a single, descriptive compile-time message.

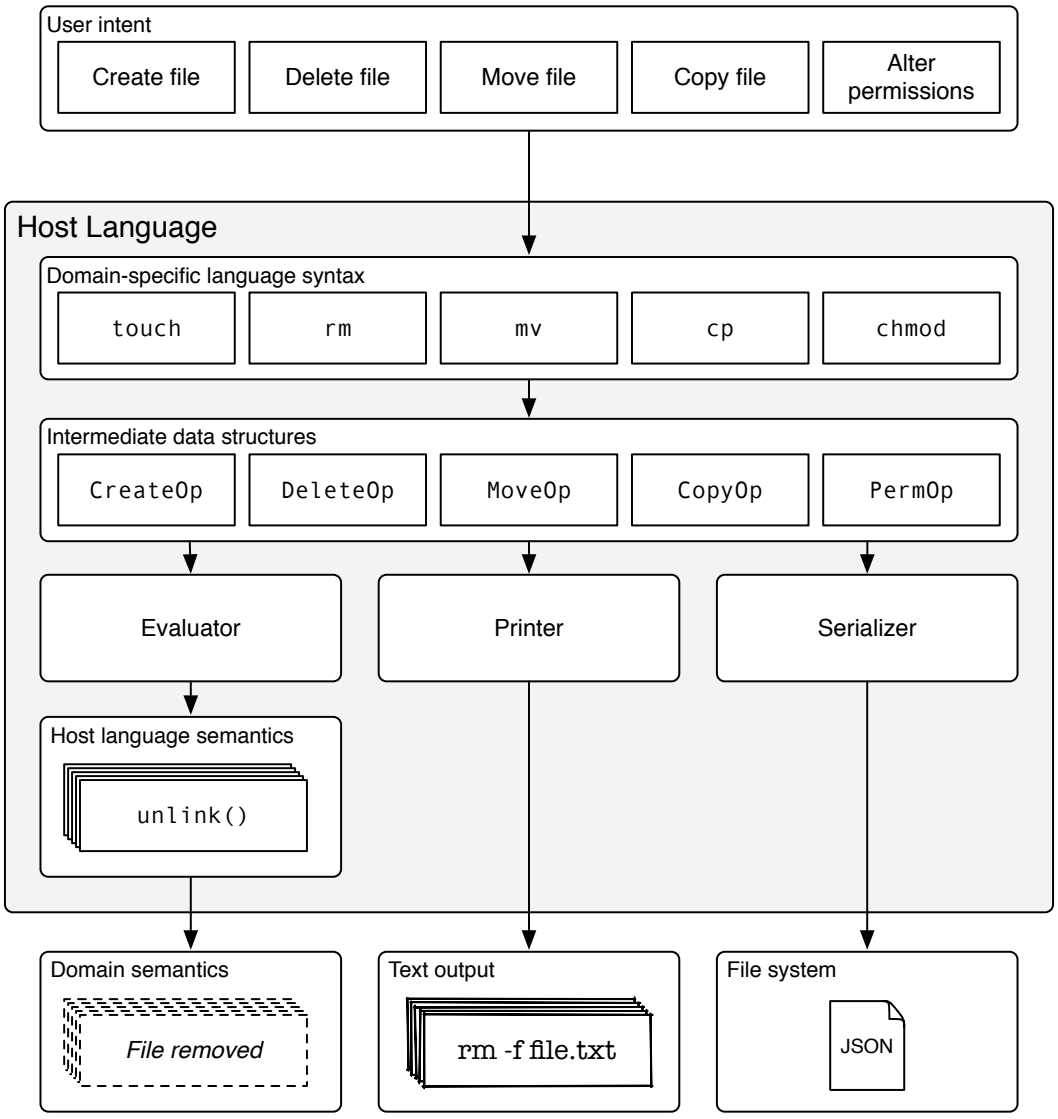


Figure 2.2: A simple overview of a deeply embedded DSL.

Chapter 3

Code Graphs and Coconut

3.1 Background

Coconut (*COde COnstructing User Tool*) is a continuing project at McMaster University dedicated to researching provably correct code generation, especially as it relates to the field of medical imaging [KAC06, ACK+04, AK07b, AK07a].

The Coconut project encompasses many related projects, but the one that is of most relevance to this thesis is its family of purely functional DSLs that encode the *instruction set architectures* (ISAs) of several IBM processors. Each DSL targets a specific architecture and allows developers to write highly optimized functions at the lowest possible level without having to worry about tedious details such as register allocation or optimal instruction ordering.

3.2 Code Graphs

The data structures that Coconut uses to store and manipulate code are called *code graphs*. There is a very mathematical treatment of code graphs in [ACK+04], but as our goals in this thesis are much more general than those of that report, we will omit any rigorous mathematical discussion about the structure of code graphs and focus on their practical applications to domain-specific languages.

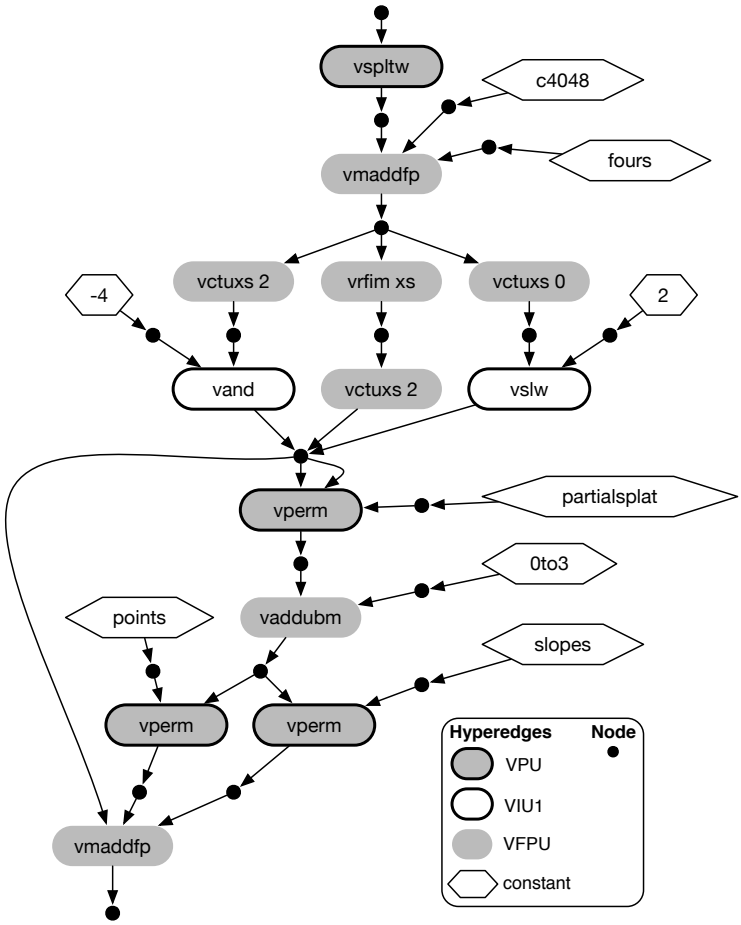


Figure 3.1: A Coconut code graph that encodes a function using the AltiVec instruction set. Reproduced more-or-less directly from [ACK+04].

A code graph is an acyclic graph structure that encodes a function's data flow. It can be represented by a hypergraph, where a node represents a value and is labelled with its type, and a hyperedge represents an operation and is labelled with its name. The incoming and outgoing tentacles of a hyperedge represent the inputs and outputs of the corresponding operation. Similarly, nodes of indegree 0 are given an ordering and considered input nodes of the whole graph while nodes of outdegree 0 are given an ordering

and considered output nodes. All inputs and outputs must preserve their respective orders.

It is worth noting that we may also represent a hypergraph as a bipartite graph, with one set of nodes representing values, and the other set representing operations. This is the representation that we use for visualization of code graphs, as bipartite graphs are much easier to draw than hypergraphs.

Code graphs homogenize the representation of code-as-data by separating the structure of the code from the semantics of the domain—where a traditional AST encodes both domain semantics and graph structure in its definition, code graphs encode only a graph structure, leaving domain semantics out of it. This lets us establish a set of common patterns and algorithms to talk about and manipulate code as data without worrying (except where necessary) about the underlying domain semantics.

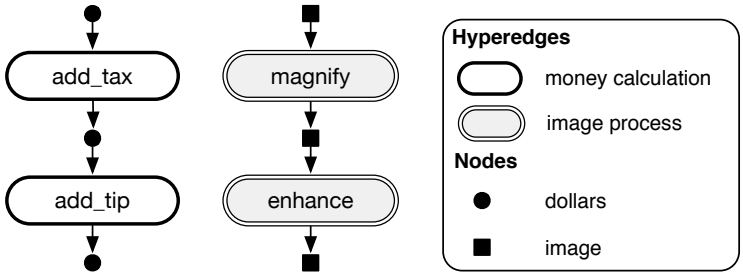


Figure 3.2: Two completely different code graphs from unrelated domains exhibiting identical structure.

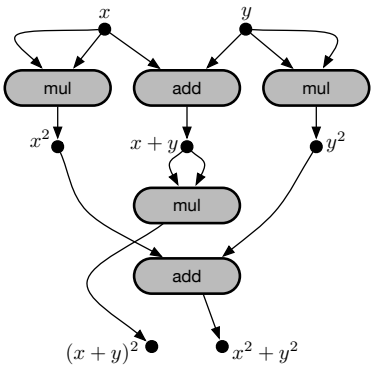
The above figure demonstrates how omitting domain semantics from the structure of our representation allows semantically unrelated code graphs to be represented using identical structures. This homogeneity is a key part of what makes code graphs so useful for domain-specific languages. Were we to represent these two code graphs using domain-specific AST types, we would have to encode the graph (tree) structure for each one separately, with it arising implicitly through the use of recursive data types. It may look something like this:

```
data MoneyExpr  =  AddTax MoneyExpr
                |  AddTip MoneyExpr
                |  – ....
data ImageExpr  =  Magnify ImageExpr
                |  Enhance ImageExpr
                |  – ...
```

These types are not compatible with any other pre-existing type, so they cannot take advantage of pre-existing algorithms. In fact, they share no common structure (as far as Haskell is concerned) whatsoever, so it is difficult (though not impossible) to make even simple algorithms work on both structures without duplicating them wholesale.

3.3 Code Generation

As mentioned at the beginning of this section, Coconut is concerned with, among other things, generating efficient code for supercomputer architectures. Exactly what "efficient code" means and how to generate it depends greatly on the characteristics of the target processor. It is not hard to generate executable code from a code graph. C code, especially, can be generated easily by simply serializing the code graph using a topological sorting algorithm and emitting one variable assignment per edge. Figure 3.3 shows an example of this.




```
void fn0(float in0, float in1,
    float *out0, float *out1) {
  float aux0 = in0 + in1;
  float aux1 = in0 * in0;
  float aux2 = in1 * in1;
  *out0 = aux0 * aux0;
  *out1 = aux1 + aux2;
}
```

Figure 3.3: A simple code graph with two inputs and two outputs, and a possible serialization as C code.

For many problems, simply emitting C code would be sufficient; whatever the architecture, a native C compiler is likely to be able to take care

of architecture-specific concerns and make sure the code is compiled relatively efficiently. However, not all compilers are perfect—especially for novel architectures—and compiler bugs can lead to impossible-to-debug (or even diagnose) defects. Aside from that, sometimes we do require absolute control over the execution of our function. In these cases, emitting C code is not going to be enough; we need to emit assembly code directly. Emitting assembly code the same way we would emit C code will give us a valid program, but not one that makes very efficient use of the CPU's resources.

In high-performance computing scenarios, it is not particularly important how many cycles it takes to run a function one time; rather, it is important how many cycles it takes to run that function thousands of times in a row, on a huge array of data. We can make use of some loop scheduling techniques to greatly increase the throughput of our function when applied to large arrays of data. This is discussed briefly in Section 5.7.1.

3.4 A Dual-Purpose DSL

The code graphs presented here so far have all been fully-formed—precomputed and pre-rendered. The question remains: how do we achieve a code graph from scratch?

Constructing a code graph can be a tricky thing. In Section 2.4 we saw how easily a user can inadvertently do the incorrect thing while believing it to be correct, and how it can be impossible to even *detect* this scenario, let alone provide adequate feedback—and that was in the presence of only a few simple primitives, most of which map directly to existing and familiar functionality. Every aspect of the structure of a code graph is essential to its overall meaning. If even a single edge or node is misplaced within the code graph, it becomes effectively meaningless; if it still constitutes a valid code graph at all, then it will be one whose semantics are undefined. If the code graph is ultimately used for code generation, it may produce valid code that simply exhibits unexpected behaviour. This kind of problem can be very hard to debug, and such egregious and untraceable bugs would quickly undermine the credibility of our software. The Coconut solution to this is to create a type-safe embedded DSL in Haskell.

Many DSLs written in Haskell make use of the monad design pattern in order to give the impression that there is imperative state behind the DSL. This is a natural fit for DSLs that construct a data structure because, obviously, the data structure needs to be kept somewhere as it is mutated—designing a monadic DSL would let us keep the code graph state behind the scenes, exposing primitives which mutate that state and ultimately extract the finished code graphs. However, we are not actually attempting to model the building of a code graph with our DSL. In fact, if anything, we would like to hide the fact that a code graph is being built at all, since that is more-or-less an implementation detail. So perhaps a monadic interface is not necessarily the best option for this kind of DSL.

Monadic DSLs do not capture the purely functional spirit of Haskell. They force explicit sequentialization of operations and specialized syntax. One of Coconut's primary goals is to provide a natural and expressive syntax embedded within Haskell for writing pure domain-specific functions. Haskell monads do provide an expressive syntax, but it is a syntax for describing imperative computations, not for pure functions. The functions we tend to describe with Coconut are not imperative (at least, we don't want to represent them that way) and so we avoid embedding the DSL inside Haskell's imperative **do** blocks.

Instead, Coconut's DSL is based purely on Haskell's declarative syntax. Below is an implementation of the function $\log_2(x)$ written using Coconut's DSL for targeting the SPU instruction set of IBM's Cell/BE platform.

```
log2SPU :: (PowerType n, HasJoin String (VR n))  ⇒ VR n → VR n
log2SPU v = evalPoly
  where
   c0PexpPart      =  fa expAsFloat c0
   frac            =  onePlusMant 23 v
   killSign        =  shli v 1
   expByte         =  shufb killSign killSign
                        $ unbytes [ shufb0x00, shufb0x00, shufb0x00, 0
                                  , shufb0x00, shufb0x00, shufb0x00, 4
                                  , shufb0x00, shufb0x00, shufb0x00, 8
                                  , shufb0x00, shufb0x00, shufb0x00, 12
                                  ]
   expAsFloat      =  csflt (ai expByte (−127)) 0
   (offset : c0 : coeffs)  =  lookup8Word (22, 20) log2OffsetsCoeffs v
   evalPoly        =  hornerV (c0PexpPart : coeffs) fracMoffset
   fracMoffset     =  fs frac offset
```

The primitives of the DSL (i.e., CPU instructions) have been underlined to emphasize their enhanced semantic role over other functions.

PowerType is the type class that represents our instruction set (so named because it encompasses the PowerPC instruction set as well as the SPU instruction set). Every instruction from the target architecture that will be usable by the user must be included as a function in the interface of PowerType, accompanied by an appropriate implementation elsewhere. This raises the important point that there may be multiple implementations of PowerType and thus multiple separate sets of semantics for a single shared DSL syntax. In this case, there are two implementations of PowerType, denoted by the names INTERP and GRAPH. These are both empty types, useful only as type-level discriminators (e.g., VR GRAPH represents a value in the GRAPH interpretation).

INTERP's implementation of PowerType reduces our DSL to a shallow embedding by mapping our primitives directly onto the corresponding instruction's semantics, as we did for FILEDSL in Section 2.3.4. It deals in numbers and simulates the flow of data through our functions as though it were executing on the target architecture. This helps domain experts develop and verify functions without needing to access a real architecture or even compile their functions into assembly code at all.

The GRAPH implementation of PowerType, on the other hand, turns our primitives into code graph constructors and transformers. The domain values in this interpretation are not floating-point numbers, but entire code graphs. Each code graph represents the entire flow of data from the inputs of the function all the way down to its corresponding domain value. As values are combined by instructions, the corresponding code graphs are merged together to produce a common ancestor for the the output code-graphs. The result of a function under this interpretation is a code graph encoding all of the data flow from the inputs of the function to the output, with no need for monads. See Figure 3.3 for an example of how code graphs are built declaratively within Coconut.

It's worth noting that this declarative interface automatically discards unused data paths within the graph. If, for example, we ultimately discarded the $x^2 + y^2$ output in Figure 3.3, then the resulting code graph would be the one associated with the $(x + y)^2$ node—one with only two instructions intead of five.

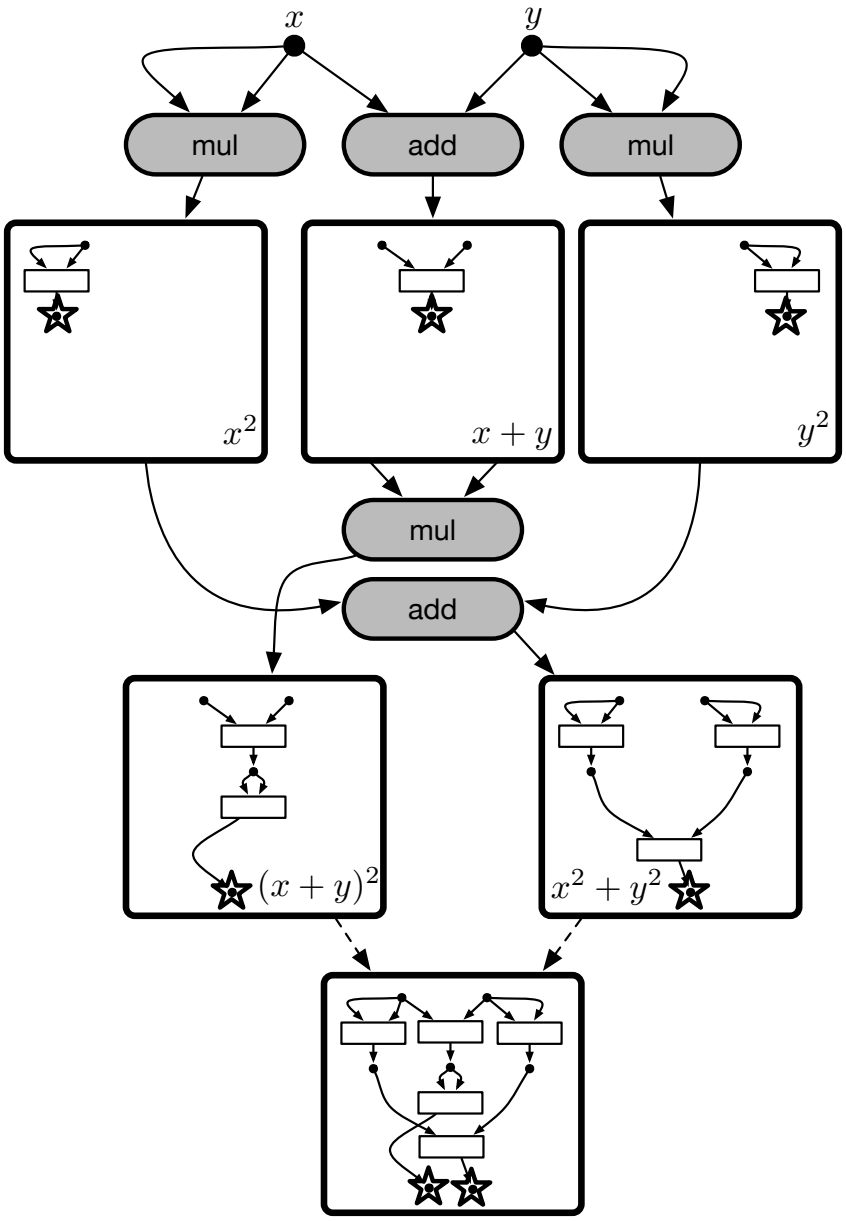


Figure 3.4: The code graph from Figure 3.3 annotated with partial code graphs.

Chapter 4

A Scala-Based Approach to Code Graphs

4.1 Introduction

In Chapter 3, we saw how code graphs can be used to form the basis of a domain-specific language, as well as Coconut's code-graph-based declarative assembly language. In this chapter we will look at how some of Scala's novel features let us tackle the same problem with a very different, more object-oriented approach, while still maintaining (and in some cases, even improving upon) the level of type safety that was achieved in the Haskell version.

Source code in this chapter is presented in a quasi-literate style—it does not constitute an entire compilable program, but it is a subset of one, and prose is interspersed amongst the code.

4.2 ExprDSL: A Simple DSL in Scala

Before describing how our code graph library works, it is good to see an example of it in action. This will help immediately highlight the strengths of Scala as a host language, as well as some of the specific techniques used.

In order to demonstrate the straightforwardness of building a DSL using Scala code graphs, we will construct a very simple expression library us-

ing our code graph library, with a DSL that encodes regular arithmetic and boolean expressions, which we will call EXPRDSL.

4.2.1 Node and Edge Labels

First we need to define the types for both values and primitives inside our DSL. In this toy language, we will use only integers and booleans as values.

```
object NodeLabels {
  sealed abstract class ExprNodeLabel
  case class INT()  extends ExprNodeLabel
  case class BOOL() extends ExprNodeLabel
}
```

We will support all the usual arithmetic and logical primitives, as well as numerical comparisons. The Sig object contains the traits from which our primitives will derive.

```
object EdgeLabels {
  import NodeLabels._
  sealed abstract class ExprEdgeLabel
  object Sig {
    trait Nullary[T]       extends ExprEdgeLabel with EdgeLabel.Nullary[T]
    trait Unary[T, U]      extends ExprEdgeLabel with EdgeLabel.Unary[T, U]
    trait Binary[T, U, V]  extends ExprEdgeLabel with EdgeLabel.Binary[T, U, V]
  }
  case class  constInt(value: Int)          extends Sig.Nullary[INT]
  case class  constBool(value: Boolean)     extends Sig.Nullary[BOOL]
  case object add  extends Sig.Binary[INT, INT, INT]
  case object sub  extends Sig.Binary[INT, INT, INT]
  case object mul  extends Sig.Binary[INT, INT, INT]
  case object div  extends Sig.Binary[INT, INT, INT]
  case object lt   extends Sig.Binary[INT, INT, BOOL]
  case object leq  extends Sig.Binary[INT, INT, BOOL]
  case object gt   extends Sig.Binary[INT, INT, BOOL]
  case object geq  extends Sig.Binary[INT, INT, BOOL]
  case object equ  extends Sig.Binary[INT, INT, BOOL]
  case object and  extends Sig.Binary[BOOL, BOOL, BOOL]
  case object or   extends Sig.Binary[BOOL, BOOL, BOOL]
}
```

Our node and edge labels are made up of case classes and case objects—these are Scala's window to *algebraic data types*, which in Haskell are defined

with the **data** keyword. Scala's approach results in a type hierarchy, rather than a single data type with multiple constructors. As a result, we are able to construct complex type hierarchies on which we can pattern match, as opposed to Haskell's sum types which cannot support a hierarchical structure without requiring deeply nested constuctors.

4.2.2 An Example: Fahrenheit-to-Celsius

Already we have defined enough to create a code graph using the relatively-syntactically-sparse DSL that comes built into our code graph library. Here is an example of a code graph that converts temperatures from degrees Fahrenheit to degrees Celsius using the well-known formula

$$C = \frac{5}{9} \cdot (F - 32)$$

```
object fahrenheitToCelsius
    extends CodeGraph[ExprNodeLabel, ExprEdgeLabel]
    with    CodeGraphBuilder[ExprNodeLabel, ExprEdgeLabel] {
  type Input  = Unary[INT]
  type Output = Unary[INT]

  val fahrenheit = input
  val minusThirtyTwo = sub(fahrenheit, constInt(32)())
  val timesFive = mul(minusThirtyTwo, constInt(5)())
  val celsius = div(timesFive, constInt(9)())
  output(celsius)
}
```

The trait CodeGraphBuilder provides a DSL-like set of primitives for constructing a code graph using a syntax reminiscent of the one we saw used by Coconut in Section 3.4. Unlike the Coconut DSL, however, this DSL is made up of state mutation primitives rather than pure functions. These mutation primitives are available strictly within the implementations of CodeGraphBuilder, and the code graphs themselves do not appear externally stateful. (cf. Haskell's ST monad for performing referentially transparent computations with local mutable state).

fahrenheitToCelsius is a complete code graph object that contains the structure of our conversion function and can be used anywhere a code graph is expected. Furthermore, by specifying the Input and Output, we made the

code graph strongly typed. This is one of the places that Scala's type system offers us flexibility where Haskell's doesn't. Each primitive, including the **`input`** and **`output`** meta-primitives, is strongly typed; the **`input`** operation's return type depends on the type assigned to Input, and the *fahrenheit* binding's type is inferred accordingly (in this case, it is a single INT-typed value node). Unary[T] is merely a semantically relevant alias for Tuple1[T]—analogous aliases exist for Binary[T, U] and Ternary[T, U, V].

Note that we can simply use the edge labels as though they were functions, making them DSL primitives in and of themselves; this function application maps internally to an insertion of the appropriate edge along with materialization of result nodes. There is no code required on the domain expert's part to do this. This is made possible by Scala's powerful implicit conversion mechanism, which lets us implicitly provide a function application definition for anything deriving from the code graph's edge label type. Once we've defined our edge labels, we can immediately build a graph with them using function application within CodeGraphBuilder. Most importantly, the function application is well typed according to the type and arity we assigned to each primitive. For example, since sub extends Binary[INT, INT, INT], the extension method sub(...) accepts two INT-typed arguments, and returns a single INT-typed value node. This is enforced at compile-time like any other function call.

The syntax, while not overly verbose, still requires us to reference the edge labels by name, which is not exactly natural for this domain. However, one of the major strengths of this approach to code graphs is that we can easily subclass CodeGraphBuilder to provide more domain-specific syntax and otherwise augment the expressivity of the code graph library to an arbitrary extent.

4.2.3 A More Domain-Specific DSL

Let us first define a new CodeGraph subtype that will be the base type of all of our expressions. This is not strictly necessary but it simplifies some type signatures and makes the type organization more semantically precise. CodeGraphInterface is a trait that contains abstract Input and Output types, enabling strongly-typed interfaces on CodeGraphs.

```
trait Expression
  extends CodeGraph[ExprNodeLabel, ExprEdgeLabel] with CodeGraphInterface
```

A parameterized subtrait defined in the companion object will act as an easy way to specify the types of an expression's inputs and outputs.

```
object Expression {
  trait Typed[Input0 <: Product, Output0 <: Product]
    extends Expression
  {
    type Input = Input0
    type Output = Output0
  }
}
```

We can refine our DSL's syntax by extending the CodeGraphBuilder trait. We also mix in our typed Expression trait so we don't have to mix it in ourselves every time we use ExpressionBuilder.

```
trait ExpressionBuilder[Input <: Product, Output <: Product]
    extends CodeGraphBuilder[ExprNodeLabel, ExprEdgeLabel]
    with Expression.Typed[Input, Output]
```

Anything defined in this trait will be visible inside the code graph definitions that implement it. This lets us easily add extension methods and type conversions that are automatically available when defining code graphs with our DSL.

The most important thing we need to do is provide some domain-specific operators for graph nodes (which are the things that represent values in our DSL). We do this by providing an implicit conversion (denoted by the **implicit** keyword) from a NodeName to a wrapper type containing our desired operations. First, we define the operators on integer nodes. Note that the return types are correct based on the return types we assigned the primitives.

```
  implicit class IntegerOperations(n: NodeName[INT]) {
    def  +   (m: NodeName[INT]): NodeName[INT]   = add(n, m)
    def  -   (m: NodeName[INT]): NodeName[INT]   = sub(n, m)
    def  *   (m: NodeName[INT]): NodeName[INT]   = mul(n, m)
    def  /   (m: NodeName[INT]): NodeName[INT]   = div(n, m)
    def  <   (m: NodeName[INT]): NodeName[BOOL]  = lt(n, m)
    def  <=  (m: NodeName[INT]): NodeName[BOOL]  = leq(n, m)
    def  >   (m: NodeName[INT]): NodeName[BOOL]  = gt(n, m)
    def  >=  (m: NodeName[INT]): NodeName[BOOL]  = geq(n, m)
    def  === (m: NodeName[INT]): NodeName[BOOL]  = equ(n, m)
  }
```

We do the same for boolean operations.

```
  implicit class BooleanOperations(n: NodeName[BOOL]) {
    def  &&  (m: NodeName[BOOL]): NodeName[BOOL]  = and(n, m)
    def  ||  (m: NodeName[BOOL]): NodeName[BOOL]  = or(n, m)
  }
```

As one last bit of syntactic sugar, we automatically promote literal values into graph nodes by the appropriate constant-materializing primitive.

```
  implicit def intConst(value: Int): NodeName[INT] =
    constInt(value)()
  implicit def boolConst(value: Boolean): NodeName[BOOL] =
    constBool(value)()
}
```

4.2.4 Fahreinheit-to-Celsius Revisited

These small additions to CodeGraphBuilder allow us to rewrite our original temperature conversion function as follows:

```
object fahrenheitToCelsius extends ExpressionBuilder[Unary[INT], Unary[INT]] {
  val fahrenheit = input
  val minusThirtyTwo = fahrenheit - 32
  val timesFive = minusThirtyTwo * 5
  val celsius = timesFive / 9
  output(celsius)
}
```

We can write this as a one-liner as well.

```
object fahrenheitToCelsius extends ExpressionBuilder[Unary[INT], Unary[INT]] {
  output(((input − 32) ∗ 5) / 9)
}
```

Separating values into separate variables is generally recommended, however, as the symbolic names of the variables can be associated with the corresponding graph nodes via Scala's type introspection mechanisms. This can be very helpful for debugging and graph rendering. In any case, the resulting code graph is shown in Figure 4.1.

4.2.5 Graph Splicing

Sometimes it is useful to compose code graphs together to reuse functionality. For this reason, CodeGraphBuilder provides function application on CodeGraphs, analogous to the function application on edge labels for edge insertion, that splices a code graph into another code graph. Like edge construction, graph splicing is type-safe and matches the interface of the spliced code graph. The arguments must match the types specified in the CodeGraphInterface trait, and the result nodes will automatically be typed accordingly as well. Here is an example of a code graph that checks to see if a temperature in degrees Fahrenheit is at or below the freezing point:

```
object isFreezing extends ExpressionBuilder[Unary[INT], Unary[BOOL]] {
  output(fahrenheitToCelsius(input) <= 0)
}
```

It is implemented by first converting the temperature to degrees Celsius and then comparing it with zero. Note that we "call" the *fahrenheitToCelsius* code graph directly, as though it were a function. Splicing *fahrenheitToCelsius* results in the whole code graph becoming part of *isFreezing*. Figure 4.2 demonstrates that the code graph structure of *isFreezing* contains the structure of *fahrenheitToCelsius* in its entirety.

4.2.6 Evaluation

With only a small amount of code, we have created a strongly typed, deeply embedded expression DSL and implemented two simple expressions with it. Now that we have graphs representing functions, we can do whatever we wish with them. One obvious option is to evaluate them under the expected semantics. The *evaluate* function takes an expression graph and a sequence

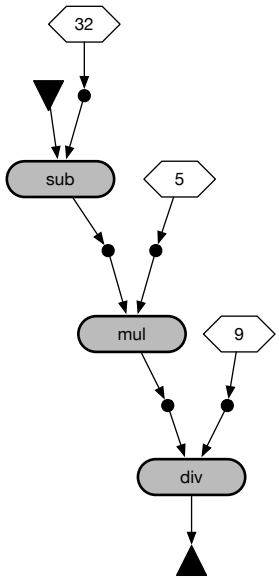


Figure 4.1: The *fahrenheitToCelsius* code graph.

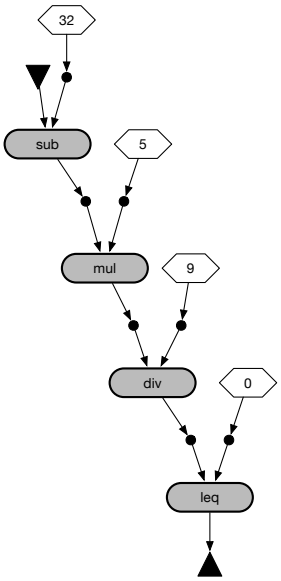


Figure 4.2: The *isFreezing* code graph.

of values representing the inputs to the expression graph, and evaluates the expression.

```
def evaluate[Input <: Product, Output <: Product]
  (cg: Expression[Input, Output], inputs: Seq[Any]): Seq[Any] = {
   import CodeGraphOps._
```

To facilitate evalutation, we simply need to store a mapping from graph node names to their corresponding values. Initially, the only known values are the inputs. Using the built-in *topSort* extension method (imported from *CodeGraphOps* above) to sort the edges in dependency order, we can simply evaluate each edge knowing that the values needed for the arguments have already been computed and are available in *env*.

```
    var env: Map[CodeGraph.NodeKey, Any] = cg.inputs.zip(inputs).toMap
    for(edgeKey ← cg.topSort) {
      val edge = cg.edge(edgeKey).head
      def args = edge.args.map(x ⇒
          env.getOrElse(x,
            throw new RuntimeException(s"Node not evaluated: $x")))
        def arg0[T] = args(0).asInstanceOf[T]
        def arg1[T] = args(1).asInstanceOf[T]
      def results = edge.label match {
        case constInt(x) ⇒ Seq(x)
        case constBool(x) ⇒ Seq(x)
        case add ⇒ Seq(arg0[Integer] + arg1[Integer])
        case sub ⇒ Seq(arg0[Integer] - arg1[Integer])
        case mul ⇒ Seq(arg0[Integer] * arg1[Integer])
        case div ⇒ Seq(arg0[Integer] / arg1[Integer])
        case lt ⇒ Seq(arg0[Integer] < arg1[Integer])
        case gt ⇒ Seq(arg0[Integer] > arg1[Integer])
        case equ ⇒ Seq(arg0[Integer] == arg1[Integer])
        case and ⇒ Seq(arg0[Boolean] && arg1[Boolean])
        case or ⇒ Seq(arg0[Boolean] || arg1[Boolean])
      }
      env = env ++ edge.results.zip(results).toMap
    }
    cg.outputs.map(x ⇒ env.getOrElse(x,
      throw new RuntimeException(s"Node not evaluated: $x")))
}
```

4.3 Error Messages

A strongly typed DSL makes writing correct code much easier due to the automatic rejection of many incorrect programs. Any code graph that would be malformed due to a semantically invalid operation will be rejected outright by the compiler. Achieving a code graph with invalid semantics is much more difficult (if not impossible) using a strongly typed construction interface.

So far we have only seen code graphs whose well-formedness is witnessed by the types of the nodes and edges—the fact that the Scala compiler accepts the code that constructs a code graph is effectively proof of its well-formedness. But what happens when a code graph is *not* well formed? We have already established that the compiler will fail. However, it is important that users be able to quickly identify and rectify problems, which means

that one of our secondary goals is to provide users not only with a compilation failure, but also with meaningful feedback in the form of an error message.

Our library does not make heavy use of macros, nor does it force users to use any non-standard syntax. Because of this, type errors are reported exactly as they would be in any other context. Consider the following function in ExprDSL:

```
object isPositive extends ExpressionBuilder[Unary[INT], Unary[BOOL]] {
  val x = input
  val pos = x > 0
  output(x)
}
```

The **output** primitive expects an argument that matches the result type of the code graph. We have mistakenly attempted to use the input, an INT-typed value, as the output, which must be a BOOL-typed value. Upon compilation, we will see something like this:

```
Error:(216, 12) type mismatch;
 found   : NodeName[INT]
 required: NodeName[BOOL]
    output(x)
           ^
```

This level of type-safety exists in every code graph operation that our DSL supports.

4.4 Type-Safe Code Graph Construction

The most fundamental feature of our code graph library is its type-safe domain-specific language framework. Our goals when producing a domain-specific language are, broadly:

- capture the maximum amount of information about a code graph;
- minimize the amount of syntax that is necessary;
- offer total type safety; and

- allow easy domain-specific refinement of the syntax.

In order to achieve these goals, we need to first define an interface through which a code graph can be built. This trait is called CodeGraphBuilder and is described in Section 4.7. It is a mutable code graph implementation with internally-available mutator primitives. Up until this chapter, we have dealt exclusively in immutable structures. In Chapter 3, we explicitly avoided any imperative-style code, preferring a pure functional style exclusively. Here, however, we make use of mutable state to build our code graph imperatively.

We embrace mutable state here in Scala for the same reason we avoid it in Haskell: to fit in with the natural idioms of the host language. In Haskell, using state is cumbersome and gets in the way of its naturally declarative style. In Scala, which is not declarative and supports mutable state inherently, the practical difference between a pure functional interface and an imperative one is far less significant. So, we elect to make use of the option which grants us the most flexibility, which is to make some use of internal mutable state for inherently mutator-like operations (like constructing a code graph), but keep the external interface pure. Scala's rich object system allows us to encapsulate all state within a small scope, exposing only an immutable impression of it externally, much like Haskell's ST monad.

4.5 A Trait-Based Object Model

Scala's *traits* are class-like packages of behavior and data which can be inherited like any class, but which are not rooted in the type system, and can be multiply inherited from. They are similar to interfaces in Java or C#, except that they can contain a subset of a full implementation, rather than just method declarations. They cannot be instantiated directly, but they can be freely composed by classes, unlike abstract base classes. It is idiomatic in Scala for data structures to be most generally represented by traits, rather than concrete or even abstract base classes. This completely separates the notion of a data structure from its concrete implementation; in fact, using this technique, many ad-hoc implementations of the same data structure may be used depending on the context.

We use this same idiom in our code graph library. Our code graph is characterized essentially the same way it is in Coconut. NodeKey and EdgeKey are both long integers.

```
trait CodeGraph[N, +E] {
  def inputs:  Seq[NodeKey]
  def outputs: Seq[NodeKey]
  def nodes:   Map[NodeKey, N]
  def edges:   Map[EdgeKey, Edge[E]]
}
```

There is no "standard" implementation of a code graph. This trait is truly the most canonical representation there is. Even the basic immutable code graph constructor is implemented in an ad-hoc fashion using an anonymous trait implementation.

```
object CodeGraph {
  /* ... */
  def apply[N, E]
    ( nodes0: Map[NodeKey, N], edges0: Map[EdgeKey, Edge[E]],
      inputs0: Seq[NodeKey], outputs0: Seq[NodeKey] ): CodeGraph[N, E] =
     new CodeGraph[N, E] {
       val nodes = nodes0
       val edges = edges0
       val inputs = inputs0
       val outputs = outputs0
     }
}
```

Each immutable code graph constructed in this way has its own anonymous singleton implementation of the CodeGraph trait pre-wired to the given data.

4.6 Strongly-Typed Code Graph Interfaces

One of the biggest changes we are making over the Haskell implementation of code graphs is to be able to encode the input and output types and arities of a code graph and its operations. So, we need a mixin trait that encodes the input and output types of a code graph. Product is a base trait for product types (e.g., a tuple).

```
trait CodeGraphInterface {
  type Input  <: Product
  type Output <: Product
}
```

We define our typed interface trait without type parameters, instead using abstract types to specify the input and output type signature of the code graph. This allows code graphs of different interfaces to be regarded as having a shared base type.

```
object CodeGraphInterface {
  type Aux[Input0, Output0] = CodeGraphInterface {
    type Input  = Input0
    type Output = Output0
  }
  /* ... */
}
```

The type aliased by Aux is a *refinement type*, which means it is a subtype of CodeGraphInterface with additional structural constraints—in this case, that the input and output types are equal to the ones specified in the parameters. Note that this is not a class, but rather the name of a type. Any CodeGraphInterface whose Input and Output types are known at compile-time to match Input0 and Output0 will be considered a subtype of the corresponding Aux type.

In a sense, the Input and Output types are existentially quantified, since the types are known to exist but the type system does not know about them at compile time. In order to represent non-existentially-quantified types, we can use the nested Aux alias to supply type parameters. Thus, if we wish to talk about code graphs with typed interfaces in general, we use the trait CodeGraphInterface. If we wish to talk about code graphs with a *particular* interface, then we use the refinement CodeGraphInterface.Aux[Input, Output].

We also define a similar interface for edge labels, allowing us to capture the type and arity of individual operations.

```
trait EdgeLabel[Args0 <: Product, Results0 <: Product] {
  type Args    = Args0
  type Results = Results0
}
```

4.7 The CodeGraphBuilder Trait

All of the domain-specific language associated with our code graph library is contained in a trait called CodeGraphBuilder. Within the scope of this trait is a mutable code graph and a set of primitives to mutate it in a domain-generic way

The main benefit of using a trait as the base representation of CodeGraph, rather than an abstract class, is that any class can make itself into a code graph by simply implementing the CodeGraph trait, regardless of what other base classes it may have. The implementations of the CodeGraph's abstract methods can even come from other traits that don't necessarily implement CodeGraph themseleves, and that is the technique we use for our CodeGraphBuilder trait.

As we saw in Section 4.2, when we are constructing a code graph with our DSL, we actually create an entirely new class (or singleton object) that implements both CodeGraph and CodeGraphBuilder. Although CodeGraphBuilder does not implement CodeGraph, it does provide concrete implementations for all of the abstract members of CodeGraph, so implementing both traits yields a full implementation of CodeGraph without linearizing the type hierarchy (i.e., making CodeGraphBuilder a subtrait of CodeGraph). That way we can specialize both CodeGraph and CodeGraphBuilder to our own purposes separately, and then mix them together. Users may mix them together into a single subtrait if they wish (as we did in our ExpressionBuilder trait on page 33), but we purposely do not enforce that restriction.

The idea behind CodeGraphBuilder is to provide a very limited context in which code graph mutators are visible, and to appear immutable everywhere else. Here is the basic structure of CodeGraphBuilder:

```
trait CodeGraphBuilder[N, E] extends CodeGraphInterface { self ⇒
  implicit protected val cg: MutableCodeGraph.Aux[N, E, Input, Output] =
    new MutableCodeGraph[N, E] {
      type Input  = self.Input
      type Output = self.Output
    }
  def nodes   = cg.nodes.toMap
  def edges   = cg.edges.toMap
  def inputs  = cg.inputs.toSeq
  def outputs = cg.outputs.toSeq

  /* ... */
}
```

For now we have omitted the mutation primitives, but we can see that, at its core, CodeGraphBuilder is nothing more than a mutable code graph with a typed interface. (The implementation of MutableCodeGraph is extremely simple: one appropriate mutable container for each of the four components of CodeGraph.)

4.8 Dependent Types

In most functional languages such as Haskell, types and data operate on two different levels. Type expressions contain types, and value expressions contain values. There is no way to, for example, define a function which takes an integer and returns a tuple of that length, because the length of the tuple is part of its type. In other words, there's no general way to map a value onto a type. Some languages, such as Coq and Agda, deal with values and types at the same syntactic level and are able to talk about types that depend on values. These types are called *dependent types*. One very powerful application of dependent types is the *dependent function*, which is a function of the form

$$f : (x : A) \to B(x).$$

This is a function whose result type depends on the value of the first argument. That is, B is a type parameterized by a *value* of type A, rather than by another type—it is a function from a value to a type.

Scala's powerful type system incorporates dependent types, which sets it apart from Haskell. Unlike in traditionally dependently-typed languages

such as Coq and Agda, Scala allows a more restricted form of dependent types called *path-dependent types*. Types and values are still distinct and cannot be freely mixed in type expressions. However, a type that is nested inside an outer type actually belongs to the *instances* of the outer type rather than to the outer type's class, as is the case in most OOP languages such as C# and Java. In order to access these types, type expressions may contain values, but only within a dot-separated path terminating in a type name (e.g., `foo.bar.ReturnType`). The concrete type referred to by the type name depends on the values in the path. It lets us represent dependent functions:

$$\textbf{def}\, f(x\colon \mathsf{A})\colon \mathsf{x.B} = \{\ /*\ ...\ */\ \}$$

It should be noted that Haskell (via GHC extensions) does have some support for using type-level *representations* of data. For example, it supports type-level natural numbers, giving rise to types such as Vec 32 Int, which might be a statically sized 32-element Int vector. Representing data at the type level is not, however, the same as dependent typing. The type-level natural number 32 is not the same thing as (nor does it contain) the Int value 32, nor can a non-literal Int value be used to determine a type-level natural. In particular, type-level representations of data do not facilitate dependent functions.

4.9 Type-Level Transformations

In Section 4.6, we defined traits that capture the input and output types of both code graphs and operations within that code graph. Using these types is not trivial, however, as the types specified by CodeGraphInterface and EdgeLabel are merely tuples of an arbitrarily chosen node type. Whereas the input type of a code graph may be (Int, Bool, Float), we need the **`input`** meta-primitive to produce a tuple of the form (NodeName[Int], NodeName[Bool], NodeName[Float]). Tuples all implement Product which allows us to access them generically at run-time, but this does not help us at compile time, where tuples of different sizes and types are effectively unrelated. We need some compile-type mechanism for polymorphically mapping types to other types.

To achieve this, we make extensive use of the `shapeless`[1] library by Miles Sabin. `shapeless` uses dependent types to provides a wide variety of generic type-level algorithms for working with heterogeneous lists, type classes, and other type-level patterns. It also leverages Scala's type-safe macro system to provide generic algorithms on tuple types of arbitrary size, which is crucial to our type-safe interface. This level of flexibility at the type level is one of the major ways that Scala excels at hosting type-safe domain specific languages.

To solve the aforementioned problem of mapping a tuple of node types onto a tuple of corresponding NodeNames, we make use of `shapeless`'s HList (heterogeneous list) algorithms to map the NodeName type class over the types in a tuple.

We define a trait that encodes the result of wrapping each type in a tuple inside of a NodeName.

```
trait NodeNameTupleMapped[L] extends Serializable { type Out <: Product }

object NodeNameTupleMapped {
  type Aux[L, Out0] = NodeNameTupleMapped[L] { type Out = Out0 }
```

We use an Aux type (a standard idiom in `shapeless`) to specify the otherwise existential output type.

We turn NodeNameTupleMapped into a type-level partial function by defining an implicit instance of it for each element in the domain. When a NodeNameTupleMapped is resolved implicitly at compile-time, the relevant instance (and thus the output type of our dependent function) will be chosen depending on the type of the input.

The first instance applies to the type PUnit, which is a stand-in for the usual Unit type that implements the required base trait Product. It functions as the base case and simply maps onto itself.

```
  implicit def nodeNamePUnitMapper:
      NodeNameTupleMapped.Aux[PUnit, PUnit] =
    new NodeNameTupleMapped[PUnit] {
      type Out = PUnit
    }
```

[1] Available at `https://github.com/milessabin/shapeless` as of October 21, 2015.

The other case is slightly more complicated and follows the same general pattern as the rest of the type-level machinery in our code graph library. The list of implicit arguments forms a chain of type-level transformations. Notice that the final parameter of each argument's type (i.e., the Out type of the Aux refinement type) matches the first parameter of the following argument's type.

```
    implicit def nodeNameTupleMapper[Args <: Product, AGen <: HList,
            AMap <: HList, ATup <: Product]
      (implicit
        agen: Generic.Aux[Args, AGen],
        amap: hlist.Mapped.Aux[AGen, NodeName, AMap],
        atup: hlist.Tupler.Aux[AMap, ATup]):
          NodeNameTupleMapped.Aux[Args, ATup] =
            new NodeNameTupleMapped[Args] {
              type Out = ATup
            }
}
```

The conversion proceeds as follows:

1. *agen* is looked up. Generic is `shapeless`'s mechanism for converting an arbitrary Product type to a generic type-level list (HList). An appropriate implicit instance is found (defined inside the `shapeless` library), and the type of AGen is able to be inferred from it accordingly. AGen is the resulting HList type.

2. *amap* is looked up. Mapped is a way of mapping a higher-order type over an HList. In this case, we map NodeName over the HList type witnessed by *agen*. AMap is the resulting HList such that every element is a NodeName wrapping the respective argument type.

3. *atup* is looked up. Tupler is a way of converting an HList to a tuple of the appropriate length. ATup is the resulting tuple, and is used as the Out type (i.e., the return type) of our type-level function.

This is the reason we enforce all of our input and output types to derive from Product, as it allows those types to be generically transformed in crucial ways.

This is only one example of dependent-type based machinery used in our code graph library. It is used throughout our library to generate strong type signatures on-the-fly based on only a few simple type parameters. We make use of it extensively in our CodeGraphBuilder trait.

4.10 Type-Safe Graph Mutators Using Dependent Types

In Section 4.7, we saw the foundation of the CodeGraphBuilder trait with all of its mutators omitted, and in Section 4.8 we saw how dependent types (along with `shapeless`'s generic programming functionality) can help us construct type-safe interfaces on-the-fly. Now we are prepared to take a look at how CodeGraphBuilder's type-safe mutators are implemented. Let us consider the **`input`** meta-primitive.

```
def input[T <: Product]
  (implicit
    nni: NodeNameInstantiator.Aux[Input, T],
    tuw: TupleUnwrapper[T]): tuw.Out = {
  if(_inputs != null) {
    return _inputs.asInstanceOf[tuw.Out]
  }
  val inputs = nni.apply()
  val nodes = inputs.productIterator.collect { case x: NodeName[N] ⇒ x }.toSeq
  for(node ← nodes) addNode(node)
  cg.inputs = MutableSeq(nodes.map(_.key): _*)
  val unwrapped = tuw(inputs)
  _inputs = unwrapped
  unwrapped
}
```

This is mostly straightforward, except for the two implicit parameters which actually do the entire duty of ensuring type safety. The two implicit parameters perform the following operations:

nni This is a dependent function that materializes nodes with fresh randomly generated node names for the given input types and returns a well-typed tuple of nodes.

tuw This is used to get rid of the type Tuple1[T] in case it arises, replacing it with *T*. This is necessary, otherwise users of CodeGraphBuilder have to unwrap the result from every operation.

Note that the return type is *tuw*.Out, meaning that **`input`** is a dependently typed function. Haskell, by comparison, is not capable of this level of type-level flexibility. We cache the result internally so that multiple uses of **`input`** will not materilize multiple sets of input nodes. The cached value *_inputs* is of type Any, requiring a bit of manual type coercion. It is safe in this case, however, since the real type is already definitively known when it is needed.

Other graph mutators, such as splicing, the **`output`** function, and edge insertion, use similar techniques to have dependent type signatures. This is what allows code graph construction to be automatically both extremely powerful and type-safe with minimal boilerplate.

4.11 Macros

Some of CodeGraphBuilder's primitives have somewhat awkward syntax when used directly. For example, splicing a code graph is done with a helper class called Splicer. Here is our *isFreezing* example from Section 4.2 with the splicing operation represented directly.

```
object isFreezing extends ExpressionBuilder[Unary[INT], Unary[BOOL]] {
  output(Splicer(fahrenheitToCelsius).apply(Tuple1(input)) <= 0)
}
```

This is unsightly and unwieldy—for one thing, the name Splicer is arbitrary and reflects an implementation detail within CodeGraphBuilder. For another, the *apply* method requires a tuple, even for a single argument. Scala lacks a nice 1-tuple syntax, so we are stuck saying Tuple1 (or a semantically relevant alias) all the time. Luckily Scala is able to provide strongly typed source code transformations using compile-time macros. Unlike preprocessor macros, such as those featured in C, Scala's macros operate at the AST level. Rather than textual replacement of tokens, Scala macro functions accept entire syntax trees as arguments and return a replacement tree as a

result. The resulting tree is spliced into the caller's AST at the point the macro was invoked with type safety intact.

Here is the macro version which accepts a list of arguments rather than a tuple. Note that it is still type-safe despite the use of Any, because the resulting substituted AST will still be dealing with strongly-typed methods. Type errors will not be allowed to propagate.

```
def apply(args: Any*): Any = macro CodeGraphMacros.spliceSyntaxImpl
```

The corresponding macro implementation is straightforward. It simply accepts any number of arguments and, depending on how many there are, emits the appropriate product type as an argument to the strongly-typed version. The q prefix on the string indicates that it is a quasiquote, meaning that the code inside the string is parsed into a Scala AST at compile-time.

```
def spliceSyntaxImpl(c: whitebox.Context)(args: c.Tree*): c.Tree = {
  import c.universe._
  val prefix = c.prefix
  args match {
    case Nil ⇒ q"$prefix.apply(<>)"
    case x :: Nil ⇒ q"$prefix.apply(Tuple1($x))"
    case xs ⇒ q"$prefix.apply((..$xs))"
  }
}
```

A similar macro system is available for Haskell called Template Haskell, although its use requires special syntax at the call site, adding substantial syntactic overhead to a DSL built for it.

4.12 Conclusion

In this chapter we have seen how easy it is to build a domain-specific code graph with our Scala code graph library, and how easy it is to build a domain-specific language on top of it. Many of the features described in this chapter are not achievable in Haskell, such as any of the instances of dependent types (including anything involving the `shapeless` library).

We feel that this library and its associated DSL achieves its goals as stated in Section 4.4. Scala has proven its worth as a rich and powerful host language

for domain-specific languages; it provides (and augments) the type safety and functional style that Haskell programmers are accustomed to, while its imperative underpinning and Java ecosystem make it an accessible and practical language for general use.

In the next chapter we will look at an example of implementing a subset of Coconut in order to demonstrate how quickly our library can help achieve its goal of modeling, simulating, and scheduling architecture-specific functions.

Chapter 5

Coconut Revisited

5.1 Overview

In Chapter 3 we briefly introduced the Coconut project and talked about how it makes use of code graphs to achieve its goals of creating highly-optimized architecture-specific functions. The Haskell library is very expressive and has certainly achieved its goals, given the performance of its results, such as those as discussed in [AK09]. Nevertheless, each Coconut Haskell DSL is implemented in a relatively ad-hoc fashion—each DSL is separate from the next, sharing only the underlying code graph data structures and algorithms. The type classes used for primitives, and thus the primitives themselves, are unrelated and so most of the critical machinery such as type-safe code graph construction must be handled anew with each new application of Coconut. This is both an inefficient use of a domain expert's time and a frustrating source of problems as the code base ages; by capturing each new instruction set in an ad-hoc DSL, we end up with a preponderance of essentially disjoint and increasingly obsolete code bases.

In fact, the motivation for this Scala code graph library lies in the inability of recent versions of the Glasgow Haskell Compiler (GHC) to compile the oldest parts of Coconut, and the level of difficulty experienced attempting to bring it up to date. In this chapter we will revisit Coconut and demonstrate how we can use our Scala code graph library to achieve the same goals with far less code, and a far greater opportunity for code reuse.

5.2 Encoding an Instruction Set

In Section 3.4, we saw a example of Coconut's Haskell DSL. As mentioned before, each architecture-specific DSL is implemented as a separate type class. We have seen PowerType, the type class implementing a DSL for IBM's PowerPC and SPU instruction sets, but there are other, very similar DSLs that target other architectures. Each one needs to implement its own mapping from primitives to code graph construction behaviour. These implementations generally rely on using raw graph transformations that must be hand-verified. Mistakes in this part can result in difficult-to-diagnose bugs when the seemingly safe, strongly typed interface yields bizarre behaviour caused by a bug in an underlying primitive function.

While the same is certainly true of our library, we have factored the core functionality out into a single Scala trait where all the graph transformation behaviour can be easily identified, audited, and verified. Implementors of new DSLs need only use the existing functionality without concerning themselves with mechanical details of code graph transformation and potentially introducing bugs. Furthermore, by providing a reusable Scala library, we allow legacy code bases to remain current by simply switching to a newer version of our library.

Encoding an instruction set in a Coconut Haskell DSL is tantamount to creating an entirely new ad-hoc code graph construction DSL. In this case, the primitives are described by the PowerType type class.

```
class PowerType n where
  data VR n
```

The associated VR type encodes a value that lives in a vector register. There are other types of registers, but for demonstration purposes we only need VR. The type class parameter n is never actually used directly, but funtions only as a label at the type level to discriminate between implemenations of the DSL.

The primitives (i.e., architecture instructions) are listed here, with their signatures specified in terms of the associated value types. Only a tiny excerpt is reproduced here—a real DSL may contain hundreds of primitives depending on the scope of the original instruction set.

```
fa   :: VR n → VR n → VR n          – add
fs   :: VR n → VR n → VR n          – subtract
fm   :: VR n → VR n → VR n          – multiply
fma  :: VR n → VR n → VR n → VR n   – fused multiply-add
– ...
```

This is merely a type class definition and these primitives lack definitions. In order to turn these primitives into a code graph, a whole implementation of this type class (including possibly hundreds of instruction primitives) must be created just to map each primitive onto the underlying code graph transformation semantics. Code reuse is essentially impossible, as a minimum of one implementation must be created for the type class, even if it functions only as a pass-through to existing behaviour. A new architecture DSL requires a significant amount of boilerplate to be written before any code graphs can be produced.

By contrast, our Scala code graph library is able to produce code graphs with only the values and operations defined. No boilerplate is necessary, and domain experts may begin using their primitives to construct code graphs immediately.

For our subset of Coconut, we need only one type of register, corresponding to a vector register, hence the name VEC.

```
object Registers {
  sealed abstract class Register
  case class VEC() extends Register
}
```

For the purposes of this thesis, we will encode exactly the instructions we need in order to implement the exponential function (see Section 5.4) and no more. It turns out we need nine different instructions (plus the const primitive) so we encode those here.

```
object Instructions {
  import Registers._
  abstract class Instruction
  object Sig {
    trait Nullary extends Instruction with EdgeLabel.Nullary[VEC]
    trait Unary extends Instruction with EdgeLabel.Unary[VEC, VEC]
    trait Binary extends Instruction with EdgeLabel.Binary[VEC, VEC, VEC]
    trait Ternary extends Instruction
      with EdgeLabel.Ternary[VEC, VEC, VEC, VEC]
  }
  case class  const(value: Vec4)  extends Sig.Nullary
  case object a                   extends Sig.Binary
  case object and                 extends Sig.Binary
  case class  cflts(scale: Int)   extends Sig.Unary
  case object fcgt                extends Sig.Binary
  case object fm                  extends Sig.Binary
  case object fma                 extends Sig.Ternary
  case class  roti(value: Int)    extends Sig.Unary
  case object selb                extends Sig.Ternary
  case object shufb               extends Sig.Ternary
}
```

As in our example DSL in Section 4.2, we are now prepared to create full code graphs using a type-safe DSL consisting of the primitives we have defined here. In fact, we do not even need to define a new syntax, since the function application style that our code graph library provides by default is appropriate.

For convenience, we create a trait to represent our domain-specific code-graphs.

```
trait FunctionGraph extends
    CodeGraph[Register, Instruction] with
    CodeGraphInterface
```

We will also create an existentially-typed function builder trait and an explicitly typed subtrait.

```
trait FunctionBuilder
  extends FunctionGraph
  with CodeGraphBuilder[Register, Instruction]

object FunctionBuilder {
  trait Typed[Input0 <: Product, Output0 <: Product] extends FunctionBuilder {
    type Input = Input0
    type Output = Output0
  }
}
```

5.3 Auxiliary Functions and Modules

Now that we have our primitives defined, we can take advantage our library's extensible nature to provide auxiliary functionality to users of our domain-specific language.

On the Haskell side, Coconut uses a pure functional code graph interface, as we saw in Section 3.4. This makes it relatively easy to construct auxilliary funtions for use within the DSL—they are just Haskell functions that operate on the value types of our DSL. The semantics of the primitives automatically establish it as a code graph construction function.

In Scala, we have to work a little bit harder. Since our code graphs are not built purely, but rather using state mutators, we cannot simply create auxiliary functions the same way we would in Haskell—each auxiliary function requires access to a CodeGraphBuilder so that it has access to its mutators (let alone something to mutate).

There are two ways to support auxiliary functions:

1. Create the auxiliary functions as entire code graphs—not a big deal considering the low syntactic overhead of code graph construction—and use graph splicing syntax to "call" the graphs as though they were functions.

2. Place the functions inside a class that depends on CodeGraphBuilder in its constructor, and then instantiate that class where the functions are needed, using the contextual builder as its argument. Then we can use them as actual functions.

Both approaches have their advantages. Under the first option, each auxiliary function will be a standalone code graph object that can be easily inspected, analyzed, printed, and debugged. The downside is mainly syntactic—graph splicing is done with function application, meaning an unsightly set of extra parentheses will arise occasionally.

The alternative is to keep our auxiliary functions as regular Scala functions, but put them inside a class that requires a CodeGraphBuilder in order to be constructed. Since this gives us the least syntactic overhead and the most flexibility, this is the technique we choose to employ. Note that a wrapper code graph can be trivially constructed that merely calls into a module function, giving us the benefits of the other approach with a small amount of overhead.

As an example, consider mathematical functions. Since Coconut is primarily concerned with numerical computation, it makes sense to have a library of useful mathematical functions at the DSL level. In order to facilitate this, we create a class called MathUtils that depends on FunctionBuilder in its constructor, effectively making MathUtils a domain-specific module.

```
class MathUtils(builder: FunctionBuilder) {
  import builder._
```

The second line is important, as it brings all of the builder's mutators and conversions into scope, essentially activating our DSL within the context of this class.

Here is one of the functions from the math library, which evaluates a polynomial.

```
  def evalPolynomial(coeffs: Seq[NodeName[VEC]])
   (v: NodeName[VEC]): NodeName[VEC] = {
    var i = 0
    var r = coeffs.last
    for(c ← coeffs.init.reverse) {
     r = fma(r, v, c) named s"poly$i"
     i += 1
    }
    r
   }
}
```

Note that it uses the code graph DSL seamlessly—such is the power of importing a CodeGraphBuilder into the current scope.

Also of note is the **named** meta-primitive being invoked on the result of the fma primitive. CodeGraphBuilder is capable of associating textual names with each graph node in order to aid in debugging, and in fact in cases where a graph node is assigned to a named field, the textual names will be deduced automatically using Scala's type reflection capabilities. In some cases, we would like to assign a name manually, so we do that here in order to sequentially number the steps of a polynomial evaluation.

In order to use the module from within a function, we can create a local object that extends our module class. This has the benefit of creating a namespace for our functions.

```
object EvalPoly extends FunctionBuilder.Typed[Unary[VEC], Unary[VEC]] {
    object math extends MathUtils(this)
    val coeffs = /* compute coefficients ... */
    output(math.evalPolynomial(coeffs)(input))
}
```

5.4 Implementing the Exponential Function

To test out our code graph library, we will implement the exponential function from Coconut's SPU math library. This function computes $\exp(x)$ for a 32-bit floating point number x. See [AS09] and [AS10] for a detailed description of how exp is implemented.

We present the original, slightly simplified Haskell code here without explanation.

```
expSPU    :: PowerType n ⇒ VR n → VR n
expSPU v  = final
  where
    final              = selb result maxFloat restrictDomainmax
    result             = fm exp evalPoly
    vBylog2            = fm v (unfloats4 (1 / (log 2) ∗ (1 + 1.5 ∗ 2 ∗∗ (−24))))
    restrictdomainmin  = fcgt (unfloats4 (−127)) vBylog2
    domainmin          = selb vBylog2 (unfloats4 (−127)) restrictdomainmin
    restrictDomainmax  = fcgt vBylog2 (unfloats4 (129 − 128 ∗ 2 ∗∗ (−23)))
    vBylog2AsInt       = cflts domainmin (23)
    exponent           = and vBylog2AsInt (unwrds4 0xff800000)
    exp                = a exponent (unwrds4 0x3f800000)
    frac               = onePlusMant 20 vBylog2AsInt
    coeffs             = lookup8Word (22, 20) coeffLists vBylog2AsInt
    evalPoly           = hornerV coeffs frac

coeffLists = [[0.48891047, ...], ...]
```

It is mostly straightforward; however, make note that the functions *onePlusMant*, *lookup8Word*, and *hornerV* are auxiliary functions within Coconut that represent many primitive instructions. *coeffLists* is a list-of-lists holding the values to be looked up by the *lookup8Word* instruction, which performs in-register table lookup.

One important thing about *expSPU* is that it is not, in fact, a code graph. Turning it into a usable code graph requires extra machinery to materialize input nodes, evaluate the function, and extract the finished code graph from the resulting output node.

Implementing this function in our Scala DSL is quite simple. The functions *onePlusMant* and *hornerV* (renamed to *evalPolynomial*) are encapsulated in our MathUtils module shown in the previous section. Rather than making constant materialization functions such as *unfloats4* (used to duplicate a constant across all four 32-bit slots of a 128-bit vector register) primitives in the DSL, as in the Haskell implementation, we elect to move them into a utility module called VecUtils and give them slightly more semantically relevant names.

The *lookup8Word* function performs an 8-way register-level lookup (via two 4-way vector registers). It has been moved into its own module which encapsulates the underlying table-lookup functionality called LookupTable8.

The data to look up, formerly held in a variable called *coeffLists*, is now part of the lookup table. The *floats* function translates a row of floating-point values into their underlying integral representations within the table. The underlying code has been ported directly from Coconut.

Here is the Scala code graph implementation of the exponential function as ported from Coconut.

```
object exp extends FunctionBuilder.Typed[Unary[VEC], Unary[VEC]] {
  object vec       extends VecUtils(this)
  object math      extends MathUtils(this)
  object coeffTable extends LookupTable8(this) {
    override val values = Seq(floats(0.48891047, ...), floats(...), ...)
  }

  val v                 = input
  val vByLog2           = fm(v, vec.splatFloat4({
                            1 / scala.math.log(2) * (1 + 1.5 * exp2f(-24))
                          }.toFloat))
  val restrictDomainMin = fcgt(vec.splatFloat4(-127), vByLog2)
  val domainMin         = selb(vByLog2, vec.splatFloat4(-127),
                            restrictDomainMin)
  val restrictDomainMax = fcgt(vByLog2, vec.splatFloat4(129 - 128 * exp2f(-23)))
  val vByLog2Int        = cflts(23)(domainMin)
  val exponent          = and(vByLog2Int, vec.splatInt4(0xff800000))
  val exp               = a(exponent, vec.splatInt4(0x3f800000))
  val frac              = math.onePlusMant(20, vByLog2Int)
  val coeffs            = coeffTable.lookup(20)(vByLog2Int).zipWithIndex.map {
                            case(n, i) => n named s"coeff$i"
                          }
  val polyResult        = math.evalPolynomial(coeffs)(frac)
  val raw               = fm(exp, polyResult)
  val result            = selb(raw, vec.splatFloat4(Float.MaxValue),
                            restrictDomainMax)
  output(result)
}
```

Figure 5.1 on page 60 shows an automatically generated representation of this code graph.

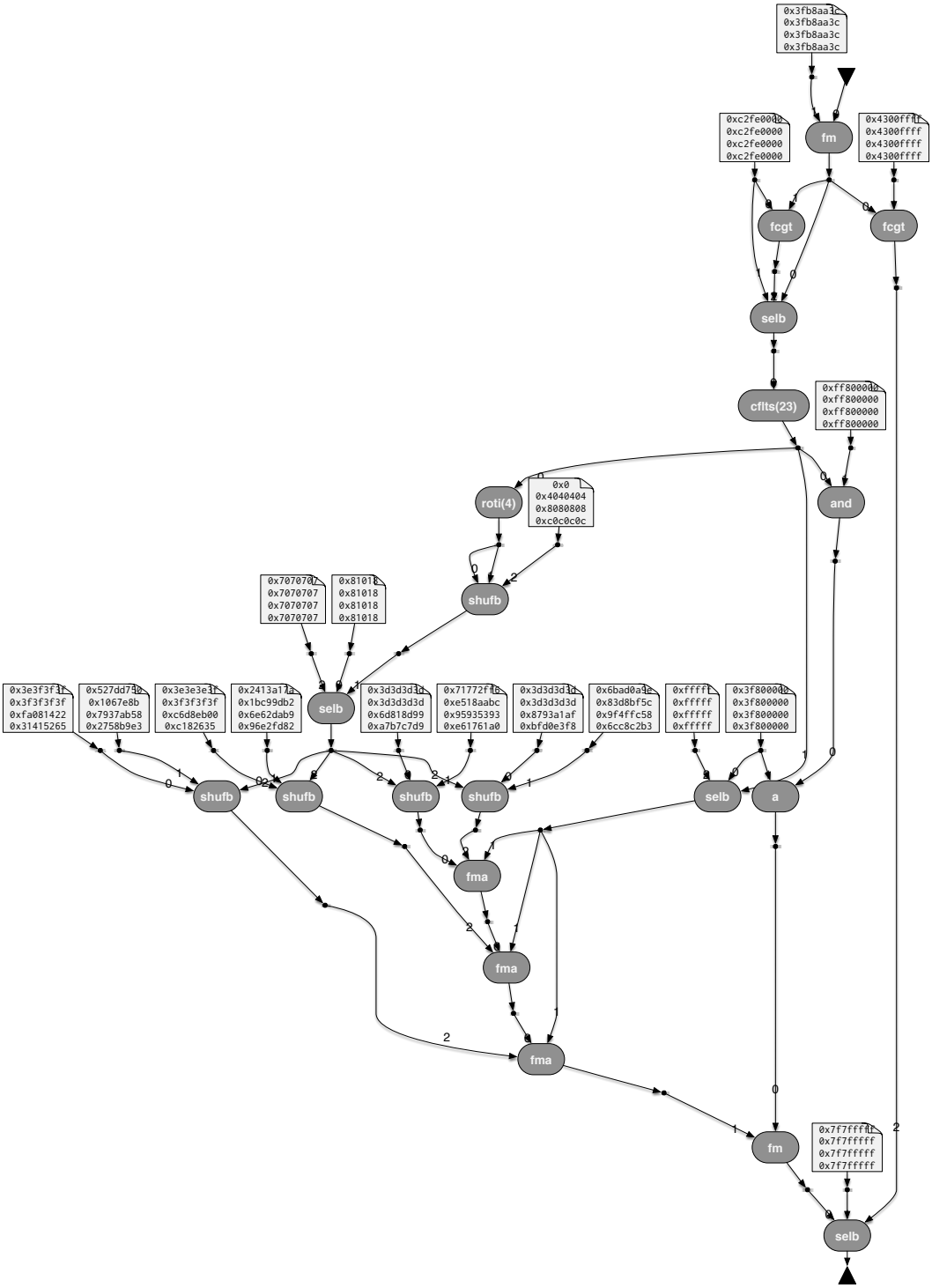


Figure 5.1: An automatically generated rendering of the exp code graph.

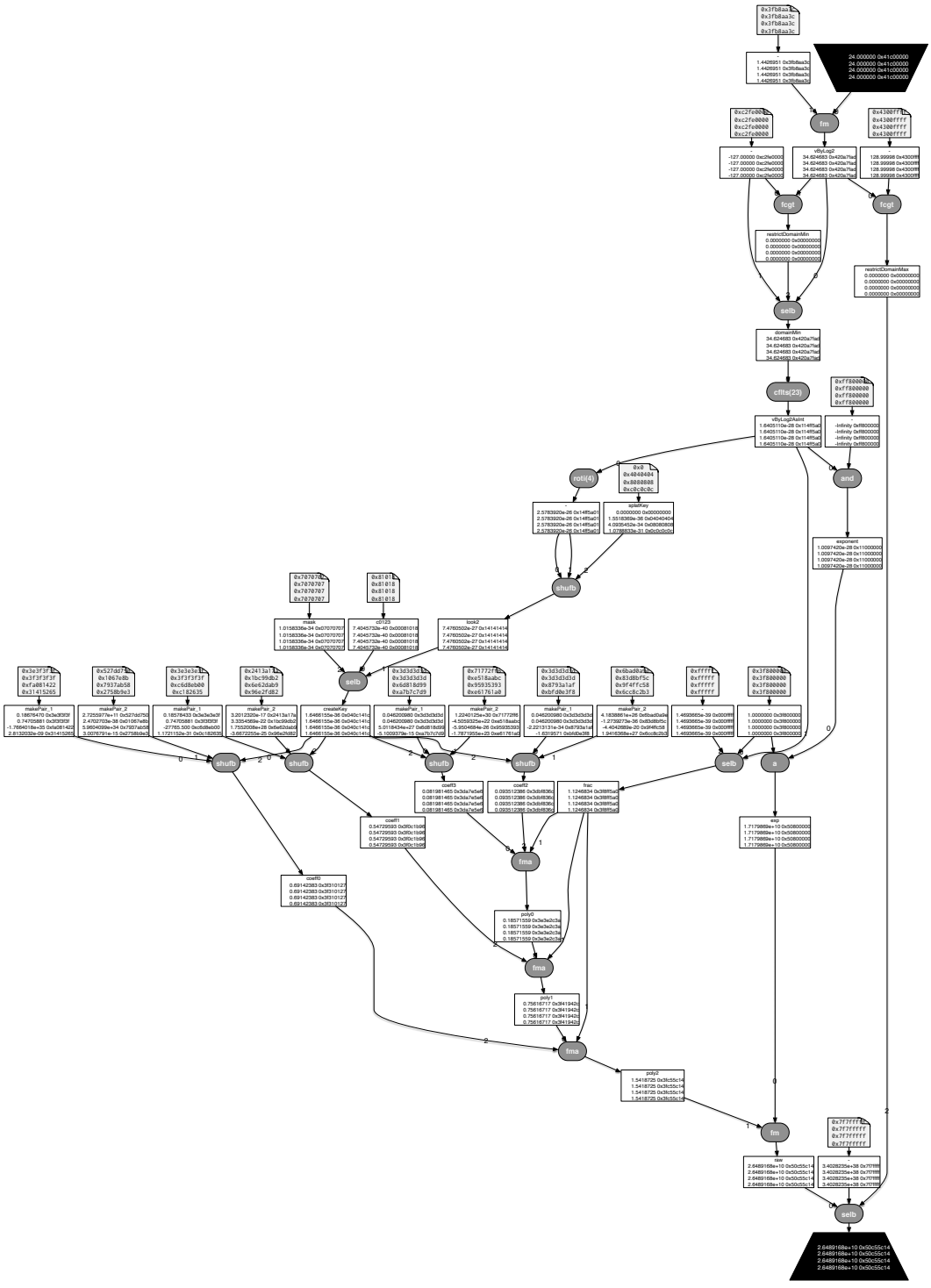


Figure 5.2: An automatically generated rendering of the exp code graph being evaluated, with intermediate nodes labelled by their name and value.

5.5 A Code Graph Interpreter

In Section 3.4, we saw that Coconut's Haskell DSLs have two sets of semantics—one for simulating functions using real values, and one for constructing code graphs. Because of the construction of our DSL, it is not possible to do the same in Scala—our primitives are state mutators, not pure functions, and anyway we aren't using a type-class-based primitive system with which to provide multiple implementations of the primitives. However, as we saw with FILEDSL in Section 4.2, we can use the built in *topSort* code graph operation to trivially produce a set of graph edges in data dependency order and evaluate them one-by-one until all nodes are computed. We do the same here, using the interpreter semantics from Coconut as a reference. The implementation is omitted, but it is largely identical to the expression evaluator in Section 4.2.

5.6 Debugging Techniques

Since our execution is deferred beyond the point of code graph construction, we cannot use normal debugging techniques such as breakpoints or print statements to trace the execution of our algorithms. In order to debug functions written for Coconut, they generally return a pair of values—the computed result node, as well as a table of intermediate values labelled by name. When run with interpreter semantics, this gives us a lens into the inner workings of our function.

See Table 5.1 for an example of one of these tables as output by Coconut. It gives us a relatively clear picture of what is going on inside the algorithm, but it places a burden on the domain expert—they must manually collate the intermediate results into a table, duplicating the symbolic value names inside string literals, and ensuring that all relevant intermediate values are represented. In Haskell, the symbolic names of value bindings are not available at runtime through any normal means, so we are stuck doing a lot of repetative mechanical work by hand.

Haskell has some runtime type introspection support in the Data.Data and Data.Typeable modules, but it still cannot help us extract the names of temporary bindings inside **where** clauses, which is where our bindings live. In Scala, however, our intermediate values actually become public fields of

Name	Slot1 (float)	Slot1 (hex)	Slot2 (float)	···
v	24.0	41c00000	17.0	···
final	2.6489167872e10	50c55c14	2.415499e7	···
result	2.6489167872e10	50c55c14	2.415499e7	···
vByLog2	34.62468338012695	420a7fad	24.52581787109375	···
restrictdomainmin	0.0	0	0.0	···
domainmin	34.62468338012695	420a7fad	24.52581787109375	···
restrictDomainmax	0.0	0	0.0	···
vByLog2AsInt	1.6405109784889363e-28	114ff5a0	1.504572583741766e-31	···
exponent	1.0097419586828951e-28	11000000	9.860761315262648e-32	···
exp	1.7179869184e10	50800000	1.6777216e7	···
frac	1.1246833801269531	3f8ff5a0	1.02581787109375	···
evalPoly	1.541872501373291	3fc55c14	1.4397495985031128	···
coeffs0	0.6914238333702087	3f310127	0.6914238333702087	···
coeffs1	0.5472959280014038	3f0c1b96	0.5472959280014038	···
coeffs2	9.35123860836029e-2	3dbf836c	9.35123860836029e-2	···
coeffs3	8.198146522045135e-2	3da7e5e6	8.198146522045135e-2	···

Table 5.1: Excerpt from debug table returned by Coconut's SPU exp implementation, which operates on four 32-bit vector slots.

our CodeGraphBuilder, allowing us to trivially access every named intermediate value using Scala's built-in runtime reflection mechanisms. In fact, CodeGraphBuilder automatically gives appropriate symbolic names to any graph nodes whose name can be deduced through reflection. The **named** meta-primitive can also be used to manually assign a name to a graph node.

What's more, since we are evaluating the code graph ourselves, rather than allowing the host language to do it for us, we have access to the value of *every* intermediate value in the computation—even the ones that are part of a subexpression or auxiliary function. In contrast, any values that are materialized as part of a subexpression or auxiliary function in Coconut are inaccessible to the interpreter and must be refactored into a separate, named value binding in order to access its value directly.

Rather than producing tables of values as in the Haskell implementation, we can actually produce a visual representation of the code graph with nodes labelled by their symbolic names and values. This lets us see the values changing as they flow through the algorithm and pinpoint the source of erroneous values much more easily than in the Haskell implementation.

See Figure 5.2 for an example of our exp code graph being evaluted with intermediate values visible and labelled.

5.7 Instruction Scheduling and Beyond

As discussed in Section 3.3, Coconut's primary motivation is to generate efficient architecture-specific code. In general, the only way to do this properly is to generate assembly code directly, since C compilers are often too general to effectively capture and optimize the quite-specialized patterns our functions are built around. Using a priori knowledge of the domain, Coconut has outperformed general compilers in the domain of vectorized mathematical functions. See [AK09] for some of those results.

5.7.1 Explicitly Staged Software Pipelining

The instruction scheduling algorithm used by Coconut is known as *Explicitly Staged Software Pipelining* (ExSSP). A full treatment of this algorithm and its relationship with code graphs can be found in [Tha06].

In a modern CPU architecture, the execution of individual instructions is broken up over the course of several clock cycles (a *pipeline*) in order to facilitate greater throughput; there is a several cycle delay between the dispatch and the completion of an instruction when its results are available, but many instructions may be executing in parallel as long as their operands do not depend on each other. As a result, generating an assembly program with the instructions in toplogically sorted order (recall Figure 3.3 for an example of this), while correct, will result in a woefully inefficient program. Each successive instruction may have to wait the full length of the instruction pipeline before it can read the result of its immediate predecessors and be dispatched. See Figure 5.3 for a visualization of an instruction pipeline.

There is not much we can do about this in the case of a single execution of the function, but since numerical computation generally involves large arrays of data, we can use some function vectorization techniques to produce high throughput for mapping our function over an entire array.

The ExSSP algorithm makes use of *loop unrolling* and *software pipelining* in order to produce an efficient loop that performs our computation in pieces,

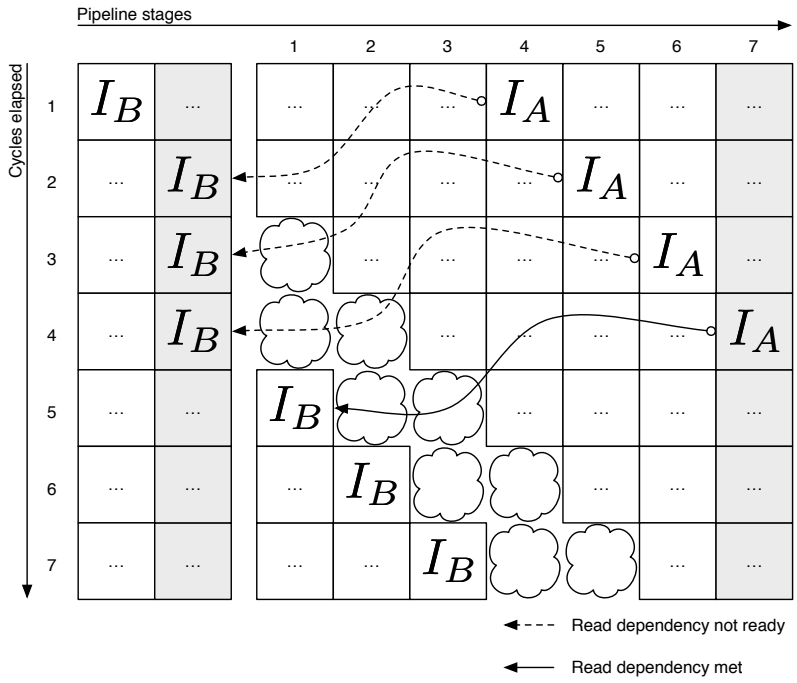


Figure 5.3: A visual depiction of a 7-cycle execution pipeline. The grey column on the left represents the instruction to be dispatched next, and the grey column on the right represents the completion stage of the instruction. I_B requires the result of I_A and so pipeline stalls (the bubbles) are issued to compensate.

allowing the inputs from several loop iterations to be processed by different parts of our algorithm at once. This gives us a large number of *independent* instructions that can be scheduled freely amongst each other, since different loop iterations never depend on each other. Thus we can maximize throughput by ensuring that no instruction is scheduled before its operands are ready, preventing the stall scenario pictured in Figure 5.3. In-depth discussion of scheduling algorithms is beyond the scope of this thesis (see [Tha06] for full details), but it is important to note that the algorithm is based heavily on code graphs, and is agnostic to the underlying implementation, meaning it (and any other known code graph algorithm) can

be straightforwardly implemented in Scala and be used to put code graphs into production in a relatively short amount of time.

5.7.2 Nested Code Graphs

Past treatments of code graphs that capture control flow [AK07b, Tha06] make use of a loop specification construct called MultiLoop that uses nested hypergraphs to establish control-flow relationships between the strictly data-flow type of code graphs we have seen so far.

Our code graph library supports strongly-typed nested code graphs almost trivially. For example, we could add the following DSL primitive to our Scala Coconut DSL:

```
case class nest[Input <: Product, Output <: Product, N <: Register, E <: Instruction]
  (cg: CodeGraph[N, E] with CodeGraphInterface.Aux[Input, Output])
  extends Instruction with EdgeLabel[Input, Output]
```

This is a primitive that references a strongly-typed code graph and whose argument and result types are deduced from that code graph. Note that the nested code graph may have more specific types than the outer graph, and we could in fact fix N and E to be specific subtypes of Register and Instruction.

For instance, we could define a new abstract subtype of Instruction called DataInstruction, and make all of our existing primitives derive from that. Other primitives, such as control flow instructions, can derive from another subtype, such as ControlInstruction. If we then restrict our data-flow code graphs to be of type CodeGraph[Register, DataInstruction], then we have established a data-flow-specific specialization of a more general code graph—such a code graph cannot contain control-flow instructions. However, since DataInstruction derives from Instruction, we can still nest it in a more general control-flow-aware code graph using the nest primitive. We can even alter nest to *only* accept CodeGraph[Register, DataInstruction], thus ensuring the semantic validity of the nesting.

There is still much research to be done in this area. The CodeGraphBuilder trait, being invariant in its node and edge label types, makes it difficult for the modules from Section 5.3 to be made generic in their underlying edge labels. This means that, for example, it may be difficult to make DataInstruction primitives usable in code graph builders that are gen-

eral enough to contain arbitrary Instruction primitives. While representing nested code graphs in a type-safe way is already possible, safely and generically *constructing* code graphs using this kind of nesting is an avenue of future research.

Chapter 6

Conclusion and Future Research

We have constructed a lightweight library in Scala that functions both as a type-safe platform for continuing the past research of the Coconut project, and as a base for the rapid development and usage of domain-specific languages in general. We have shown that it is more than capable of achieving the same results as past Coconut DSLs with far less boilerplate and syntactic overhead. We have also seen that it can easily achieve results in other domains, with EXPRDSL in Section 4.2.

We have used code graphs in their most general formulation in this thesis. Code graphs have been used in more specialized formulations, such as in [AK07b], where control flow and data flow are modeled as nested code graphs. This was discussed briefly in Section 5.7.2. While we have left the door open to a natural well-typed formulation of nested code graphs, we have not explored it as a way to model control-flow. In particular, the current invariance of CodeGraphBuilder's type parameters raises significant issues in using the module pattern in Section 5.3 when multiple types of primitives are introduced. Mitigating these issues and further exploring type-safe modeling of control-flow via nested code graph construction is an avenue for future research.

The library presented in this thesis is fairly lightweight and stands alone. Other Scala DSL libraries such as Yin-Yang [JSS+14] and Delite [BSL+11, RSL+11, SRB+13] feature much more complex treatments of DSLs. We chose to build our own library from scratch so that we could directly incorporate the underlying code graph data structure into it, rather than

making it a byproduct of some other DSL's intermediate representation—specifically, we wanted to, for the purposes of this thesis, explore the utility of code graphs as DSL intermediate representations in and of themselves. Nevertheless, it is worth exploring alternative DSL libraries as a front-end to code graph generation.

Scala has proven itself a worthy host to domain-specific languages. Its support for dependently typed method signatures has opened the door for a level of automatic type safety that was not possible with Haskell. We have only scratched the surface of its powerful type-safe macro system—there is much future work to be done investigating the syntactic possibilities that macros can bring to our code graph library and its DSLs.

Bibliography

[ACK+04] Christopher Kumar Anand, Jacques Carette, Wolfram Kahl, Cale Gibbard, and Ryan Lortie. Declarative assembler. SQRL Report 20, Software Quality Research Laboratory, McMaster University, October 2004. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[AK07a] Christopher Kumar Anand and Wolfram Kahl. A domain-specific language for the generation of optimized SIMD-parallel assembly code. SQRL Report 43, McMaster University, May 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[AK07b] Christopher Kumar Anand and Wolfram Kahl. Multiloop: Efficient software pipelining for modern hardware. In *CASCON '07: Proc. 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 260–263, New York, 2007. ACM.

[AK09] C.K. Anand and W. Kahl. An optimized Cell BE special function library generated by Coconut. *Computers, IEEE Transactions on*, 58(8):1126–1138, Aug 2009.

[AS09] Christopher K. Anand and Anuroop Sharma. Unified tables for exponential and logarithm families. AdvOL Report 2009/2, McMaster University, 2009.

[AS10] Christopher K. Anand and Anuroop Sharma. Unified tables for exponential and logarithm families. *ACM Transactions on Mathematical Software*, 37(3), 2010.

[BSL+11] Kevin J Brown, Arvind K Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun.

A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100. IEEE, 2011.

[JSS+14] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: Concealing the deep embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 73–82. ACM, 2014.

[KAC06] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy MacCaull, Michael Winter, and Ivo Düntsch, editors, *RelMiCS 2005*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[Kah11] Wolfram Kahl. Dependently-typed formalisation of typed term graphs. *arXiv preprint arXiv:1102.2653*, 2011.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[RSL+11] Tiark Rompf, Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented DSLs. *arXiv preprint arXiv:1109.0778*, 2011.

[SRB+13] Arvind K Sujeeth, Tiark Rompf, Kevin J Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, et al. Composition and reuse with compiled domain-specific languages. In *ECOOP 2013–Object-Oriented Programming*, pages 52–78. Springer, 2013.

[Tha06] Wolfgang Thaller. Explicitly staged software pipelining. Master's thesis, McMaster University, Department of Computing and Software, 2006. `http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf`.