CPCG: A Cross-Paradigm Code Generator

Bianca Curutan
Department of Computing and Software
McMaster University

June 25, 2013

Abstract

The goal of this project is to demonstrate the existence of a language of design that bridges the gap between abstract algorithms and explicit code. The design decisions made between the specifications of abstract algorithms and their concrete implementations may be encoded in a code generator using this tangible language of design. The Cross-Paradigm Code Generator (CPCG) is designed as a tool to generate code, allowing programmers to solve problems at the level of abstraction of the problem domain. Two domain specific languages were developed to incorporate the core ideas of this project; that is, the concepts and semantic characteristics of all the paradigms and the choices behind the design of algorithms. The generated code demonstrates the existence of abstract algorithms and a tangible language of design.

Acknowledgments

I would like to thank my supervisor, Dr. Jacques Carette, without whom this research project would not have been possible. I greatly valued his guidance and expertise throughout the course of my studies at McMaster University.

Special thanks to the members of my examination committee, Dr. Jacques Carette and Dr. Wolfram Kahl, for taking time out of their busy schedules to review and improve the quality of this work.

I would also like to thank my family, friends, and fellow students for their continuous encouragement, advice, and support.

Contents

List of Figures

Executive Summary

The goal of this project is to demonstrate the existence of a language of design that bridges the gap between abstract algorithms and explicit code. When used to describe distinct concepts or thought patterns for algorithms, the four programming paradigms — functional, logic, object-oriented (OO), and imperative — appear unrelated initially. They are, however, able to express certain common concepts, albeit in different ways. The design decisions made between the specifications of abstract algorithms and their concrete implementations may be encoded in a code generator using this tangible language of design.

The Cross-Paradigm Code Generator (CPCG) is designed as a tool to generate code, allowing programmers to solve problems at the level of abstraction of the problem domain. Two domain specific languages (DSLs) were developed to incorporate the core ideas of this project; that is, the concepts and semantic characteristics of all the paradigms (referred to as the internal language in this report) and the choices behind the design of algorithms (referred to as the design language). The CPCG uses the internal DSL along with supplemental Haskell modules and data structures representing a series of languages (Java, Lua, Haskell, λProlog, C#, Lisp, Lua, Prolog, and Scala) to implement ASTs.

To generate code, the secondary design language is used in conjunction with the DSL to accommodate external choices as provided by the user. The CPCG converts programs written in the internal language into intermediate representations of the implementation languages based on the design decisions, and then applies the final syntactical details before outputting source

code. The generated code demonstrates the existence of abstract algorithms and a tangible language of design.

This report describes characteristic concepts of programming paradigms as well as the technical nature of DSLs. Subsequent sections provide an in-depth examination of the CPCG project, including its modules (categorized by type), DSL design, development and implementation details, and testing information.

Chapter 1

Introduction

Despite having the capability to essentially model the same programs, some languages, such as C/C++ and Java, are typically favoured by industry while others, such as Mozart-Oz and Haskell, are more often found in academic environments. As such, developers are typically more familiar with some types of programming languages than others. This trend is mainly due to the separate natures or paradigms of the programming languages.

Stating that the functional, logic, object-oriented (OO), and imperative paradigms are totally distinct fundamental styles of computer programming is a reasonable notion. They vary not only in obvious stylistic and syntactical ways, but also in the abstractions used to represent the elements of programs and the steps used to compose computations. Moreover, they are foundationally based on different models of computation: the λ-calculus for functional programming, first-order logic for logic programming, and the Turing machine for OO and imperative programming. (Chapter 3 contains more information about the four programming paradigms.) However, deeper analysis observes that the paradigms share some fundamental concepts and patterns that, when abstracted, form the foundation of any cross-paradigm implementation language.

The aim of the CPCG is to demonstrate that although the four programming paradigms seem unrelated, the design decisions made between the specifications of abstract algorithms and their concrete implementations may

be tangibly encoded in a code generator. It is out of scope of this academic project to abstract all possible shared concepts within these paradigms; only concepts required to implement some simple, standard programs are examined. A more detailed description of this goal and its limitations are found in Chapter 2.

To achieve this goal, one DSL was developed to represent the abstract core ideas shared among the paradigms, independent of language implementation, and another to provide and limit design choices for the algorithms. This report describes the technical nature of DSLs in Chapter 4; information about these specific DSLs may be found in Chapters 7 and 8. Projects developed by other students and programmers, such as Story of an Acyclic Graph Assemble (SAGA) program [BC11] and a Generic Object-Oriented Language (GOOL) [CC12], were used as guides for developing the initial AST types and design DSL before the modules were adapted to better accommodate the CPCG's goals.

Chapter 5 of this report explains the project requirements — what users should expect with regards to use and functionality of the code generator. Information about the methodology on the design of the project is included in Chapter 6. Chapters 9 and 10 describe the development, implementation, and testing phases and contain code samples.

Chapter 2

Project Goals and Process

The goal of this project is to demonstrate that, although the four programming paradigms — functional, logic, OO, and imperative — for algorithms appear unrelated initially, the design decisions made between the specifications of the abstract algorithms and their concrete implementation may be encoded in a code generator. In particular, that there exists a fundamental set of concepts and design decisions used to represent the elements of programs and the steps used to compose computations at a base level, and that a design language based on these external choices is a concrete artifact, not just theoretical.

"Ideal" solutions for simple, standard programs were developed or collected, then similarities were theorized based on observation and analysis with conceptual input from a variety of programming languages textbooks [Cla00,FWH01,Mit96,Mit02,Seb01]. This method is, of course, likely liable to human error and prone to bias based on the researcher's knowledge base and experience.

Ultimately, the following were developed:

- A DSL to internally demonstrate some needed concepts. Semantic pseudocode was initially used to capture high-level ideas before writing the abstractions and data structures in the host language, Haskell.

- A design language (Chapter 8) to provide — yet limit, per the scope of the project — the design choices for users to input to the software.
- Language ASTs to supplement the DSL with additional abstractions that are evident in one or more paradigms but not in the others. Eventually, these language ASTs were identified more accurately as paradigm ASTs since programming languages from within the same paradigm translate similarly.
- Renderers to map DSL code to intermediate representations of code in the respective paradigm, forming "almost" source code — again, there is one per paradigm.
- Pretty printers to apply the proper syntax to the intermediate representations of code and generate the final compilable and implementable source code. Since syntax naturally differs among languages, regardless of the paradigm(s) they are in, there is one printer per language.

As previously mentioned, only enough abstract characteristics are identified to produce a sample of programs using the CPCG. It is out of scope of this academic project to identify all common features between the programming paradigms.

Chapter 3

Programming Paradigms

The internal language represents some of the main concepts shared between implementation languages within each of the programming paradigms. But what do these so-called "main concepts" refer to? This section of the report provides perspective on both the similarities and differences between the paradigms as well as explores ideal quicksort solutions coded in each of the primary project languages.

3.1 Concepts of OO Languages

There is much mainstream support for OO programming [Seb01], even within non-OO languages as explored in Section 3.5. In OO design, programs are thought of as a collection of interacting objects that invoke methods or pass messages [Pie02]. These objects are each viewed as independent entities with their own state which is accessed and modified by methods in the program. The OO paradigm promotes the reusability, sharing, and extensibility of code, popular characteristics in industry. Objects are encapsulated into classes to logically group variables, types, methods, and events and/or inherit them from the classes' associated superclasses, enabling the creation of new types [weba].

Classes are constructs which are used to create instances, known as objects, of themselves based on blueprint code; classes represent concepts

whereas objects represent phenomena. Classes consist of instance variables, which are local to the class itself, and methods to associate state and behaviour with the objects of that class [CC12]. Generally, classes are instantiated as objects before their data are accessible or modifiable, each instance maintaining its own set of data separate from other instances; classes represent concepts whereas objects represent phenomena.

As previously mentioned, methods are one component of classes. Based on their signatures, methods accept certain types of arguments as input; these types may include base types, such as int or boolean, or compound types, such as lists or arrays. The data are then processed or modified and certain types of results are returned if return statements are defined, or only control flow is returned if the return type is void. The signature also includes information about the scope of methods. If the method is private, it is a helper used in other methods within the class and, thus, is only accessible within the class. If the method is public, the method is accessible outside of the class or object. In C++, there are also protected methods which are accessible from members of the same class or derived classes [weba].

Classes generally contain one or more special methods without return types (void or otherwise) called constructors — they are the methods used to initialize new objects. Execution of the constructor performs memory allocation for the object and initializes any instance variables [CC12]. More than one constructor may exist in classes due to overloading. It is also possible to restrict instantiation to only one object, known as the singleton design pattern.

Some of the main features of languages within the OO paradigm are [Nor]

- Inheritance: Classes inherit and sometimes override characteristics from base classes, such as structural and behavioural features, creating a hierarchy of classes. However, the derived classes are also developed to include new methods and instance variables that modify state and store additional data respectively, extending the scope of the base classes.

- Subtyping or subtype polymorphism: Subtypes are data types that are related to other data types by some notion of substitutability. Using an example, if S is a subtype of T, then any term of type S is safely used in a context where a term of type T is expected. Generally, subtyping should be reflexive and transitive [Pie02].

- Data abstraction: The system hides certain details of how data is stored, created, and maintained. Data abstraction enforces separation between the abstract properties of data types and the concrete details of their implementations, a key factor in the CPCG project. The abstract properties of the data type are publicly accessed and modified, but the concrete implementation is private.

- Dynamic dispatch: Sending the same message to different objects may yield different results or responses. The methods executed within objects depend on how the objects were implemented or changed; objects do not maintain only static attributes [Mit02, FWH01].

Some examples of object-oriented languages are Ruby, Groovy, and C++; the object-oriented languages generated in the CPCG are Java, C#, and Scala.

3.1.1 Quicksort in Java

The ideal quicksort solution in Java is shown in Figure 3.1 and the generated solution is in Appendix B, Figure B.3.

In this example, Java has a distinct advantage — the mutable array is sorted in place without using any extra memory, making it relatively fast and more efficient than implementations which use immutable structures.

3.2 Concepts of Imperative Languages

Imperative programming is a paradigm that emphasizes changes in program state via describing computation in terms of statements as functions of time [Nor]. The basic idea of imperative programming is guided sequences

```
public class Quicksort {
  public void quicksort(int[] items) {
    if (items.length > 1) {
        quicksortRec(items, 0, items.length-1);
    }
  }

  private void quicksortRec(int[] items, int leftIndex,
  int rightIndex) {
    if (leftIndex < rightIndex) {
      int pivotIndex = partition(items, leftIndex, rightIndex);
      quicksortRec(items, leftIndex, pivotIndex-1);
      quicksortRec(items, pivotIndex+1, rightIndex);
    }
  }

  private int partition(int[] items, int leftIndex,
  int rightIndex) {
    int pivotValue = items[rightIndex];
    int i = leftIndex-1;
    int j = leftIndex;
    while (j < rightIndex) {
      if(items[j] < pivotValue) {
        i++;
        swap(items, i, j);
      }
      j++;
    }
    swap(items, i+1, rightIndex);
    return i+1;
  }

  private void swap(int[] items, int i, int j) {
    int k = items[i];
    items[i] = items[j];
    items[j] = k;
  }
}
```

Figure 3.1: Ideal Quicksort Solution in Java

of abstractions of one or more actions to commands, each of which may have observable effects (or side effects) on the program state. The imperative paradigm most closely resembles machine language since it grew naturally from assembly language, generally making it more popular among and familiar to programmers.

Imperative programs essentially consist of sequences of assignments to variables, describing a set of possible sequences. For execution, the statements are performed in a stepwise, sequential manner as described by the program. The flow of execution is modified by conditional and looping statements, abstracting the branch instructions found in the underlying machine instruction set.

The main features of imperative languages are procedures and functions, expressions and variable assignments (i.e., assign or reassign the value stored in the storage location which sometimes occurs frequently within programs), data and control structures, I/O commands, and error and exception handling mechanisms. Again, the importance of the order of procedures is emphasized as different orders of operations (or different inputs) may or may not yield different outputs.

Examples of imperative languages are Pascal, Go, and Ada. The imperative languages used in the CPCG are Lua, the ideal quicksort solution of which is very conceptually similar to Java's OO solution in Figure 3.1, and C.

3.2.1 Quicksort in Lua

The ideal quicksort solution in Lua is shown in Figure 3.2 and the generated solution is in Appendix B, Figure B.5.

3.3 Concepts of Functional Languages

Functional programming relates to the theory of functions rather than to the theory of sets, concentrating on the high-level of "what" is being computed instead of the low-level "how" [Hud89]. It has its roots in the λ-calculus, a

```
function quicksort(items)
  if (#items > 1) then
    return qsRec(items,1,#items)
  end
end

function qsRec(items,leftIndex,rightIndex)
  if (leftIndex < rightIndex) then
    local pivotIndex = partition(items,leftIndex,rightIndex)
    qsRec(items,leftIndex,pivotIndex-1)
    qsRec(items,pivotIndex+1,rightIndex)
    return items
  end
end

function partition(items,leftIndex,rightIndex)
  local pivotValue = items[rightIndex]
  local i = leftIndex-1
  local j = leftIndex
  while (j < rightIndex) do
    if (items[j] < pivotValue) then
      i = i+1
      swap(items,i,j)
    end
    j = j+1
  end
  swap(items,i+1,rightIndex)
  return i+1
end

function swap(items,i,j)
  local k = items[i]
  items[i] = items[j]
  items[j] = k
end
```

Figure 3.2: Ideal Quicksort in Lua

formal system in mathematical logic for expressing computations via variable binding and substitution. All computations are done by applying or calling functions and the natural abstraction, abstracting a single expression to a function which are then evaluated as an expression, is the function. Program execution involves functions as first-class objects [Zeh] calling each other and returning results; state and mutable data are generally avoided.

In the functional paradigm, all functions are first-class values without state changes and complex functions are built by composing simpler functions. Functions are treated like other values and passed as arguments to other functions or returned as results of functions, known as higher-order programming. Furthermore, it is possible to define and manipulate functions from within other functions [webe].

Functional programming languages follow a declarative approach since they describe actions. Developers declare what programs do by defining functions that map inputs to outputs, taking in arguments and returning a single solution to move forward. Computation is treated as the evaluation of mathematical functions. The use of functional programming languages is generally emphasized in academia rather than in commercial systems, although the Scala language tends to be an exception to this rule due to its multi-paradigm affinity to both functional and OO programming.

Some of the main characteristics of functional languages, according to Normark, [Nor] are

- Capability to accommodate higher-order functions (HOFs): HOFs are functions that take other functions as their arguments, such as the map function, and/or return functions [Dav]. They are used to refactor code or reduce the amount of repeating or redundant code.

- Avoidance of side effects: Functions have side effects if, in addition to returning a value, some state is modified or interactions with calling functions of the outside world are observable. Avoiding side effects makes functional languages purer and less prone to errors.

- Performance on immutable data: The state of immutable data is not changed after it is created. Thus, rather than altering existing values,

copies are created to preserve the original data. A good example of this is the original ideal solution determined for Haskell's quicksort algorithm 3.3.

- Referential transparency property: Expressions are replaced with their values without changing the behavior of the program. The same effects and output occur if given the same input [OSG08].

- Application of lazy evaluation: In Haskell, function arguments are not evaluated unless these values are required to evaluate the function call, permitting the use of infinite lists in programs. This laziness contrasts strict evaluation where the function's arguments are evaluated before invoking the function, regardless of whether or not these values are necessary.

- Use of recursion: Iteration in functional languages is not technically possible since it inherently requires side effects (see section 3.2 for more details on side effects), focusing on the "what" rather than the "how" of functional programming [Lip11]. Thus, iteration is simulated via recursion — functions invoke themselves to be called over and over again.

Examples of functional languages are OCaml, LISP, Clojure, and F#. The functional language generated as well as used as the host language in the CPCG is Haskell; LISP source code is also generated in this project.

3.3.1 Quicksort in Haskell

The ideal quicksort solution in Haskell is shown in Figure 3.3 and the generated solution is in Appendix B, Figure B.1.

The ideas behind this quicksort solution are discussed in more detail in Chapter 9. For now, however, note the use of "(x:xs)", indicating that a list data structure is being used. Also note the line "import Data.List" to use Haskell's built-in partition function from the Data.List library rather than a user-developed function to accomplish the same task.

```
module Quicksort where
import Data.List

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort less ++ (x:quicksort rest)
  where (less, rest) = partition (< x) xs
```

Figure 3.3: Ideal Quicksort in Haskell

3.4 Concepts of Logic Languages

Logic programming is a paradigm based on first-order logic — more precisely, Horn clauses. Programming is achieved by declaring predicates (facts) and rules of inference (templates for building valid arguments) rather than using assignment and control-flow structures [Seb01]. These predicates and rules of inference define the scope of the problem. The absence of functions infers that logic programming does not return values.

Just as imperative and OO are similar paradigms with regards to design decisions, semantics, etc., so too are the functional and logic paradigms similar — the logic paradigm also follows a declarative approach using mathematical logic. However, unlike functional programming, logic programming includes unification, solving equations between terms with variables, and backtracking search.

Programs implemented in logic programming languages are executed by querying whether particular facts are deduced from the predicates and rules of inference in the program [Dav]. Since the program actually solves the problem, the steps leading to the solution are minimal and proving its validity is relatively simple compared to other cross-paradigm languages.

According to Apt, core ideas behind logic programming are as follows [Mit02]:

- Computing takes place over the domain of all terms defined over a "universal" alphabet.

```
quicksort([], []).
quicksort([Head | Tail], Sorted) :-
  partition(>(Head), Tail, Less, Rest),
  quicksort(Less, SortedLess),
  quicksort(Rest, SortedRest),
  append(SortedLess, [Head | SortedRest], Sorted).
```

Figure 3.4: Ideal Quicksort in λProlog

- Values are assigned to variables by automatically generating substitutions called Most General Unifiers (MGUs); these values contain variables called logical variables [Pie02].
- Control is provided by a single mechanism: automatic backtracking.

Examples of logic programming languages are Mercury, Ciao, and Mozart-Oz. The logic language used in the CPCG is λProlog, a version of the well-known Prolog language intended for higher order programming.

3.4.1 Quicksort in λProlog

The ideal quicksort solution in λProlog is shown in Figure 3.4 and the generated solution is in Appendix B, Figure B.2.

Unlike the Haskell solution seen previously, this λProlog code does not need to import any libraries to use the built-in partition function. However, the design decision to use an immutable list data structure and copy data is also used here.

The rather minimalist quicksort solutions for Haskell and λProlog are not considered "true" quicksort implementations since the elements are not sorted in-place, although the problem is still solved via the divide-and-conquer algorithm.

3.5 Similarity Analysis of Paradigms

The discussion of programming paradigms in the previous sections may have only served to highlight their differences rather than their similarities, so this section reviews some introductory examples of how the paradigms relate.

The most apparent comparison thus far is perhaps imperative versus OO programming. Ignoring the details of the syntax in the two quicksort solutions for Lua and Java respectively, it is clear these two languages share core features as explored in their in-place quicksort pseudocode (Figure 3.5). More examples of imperative and OO programming are found in a case study of Imperative Objects [Pie02].

The similarities between the paradigms do not, of course, end there. In fact, using the F# functional programming language, it is possible to define classes (such as the Vector2D class in Figure 3.6) and interfaces containing only abstract members (such as the IShape interface in Figure 3.7) as used and inherited in Java or other OO languages. These examples as well as more complete explanations of these phenomena may be found in [Dav].

Functional programming may also be compared to imperative programming. F#, for example, includes full support for imperative interaction via mutable variables, arrays, loops, I/O, and exceptions (Figure 3.8) [Dav].

So now imperative, functional, and OO programming have all been demonstrated (at least at a high-level) to have common features and the capability to support similar design decisions in their code. But what about the logic paradigm? Using F# again to represent functional languages, a library is built by combining HOFs and effects to support logic programming, as shown in Figure 3.9 [Dav] .

Alternatively, logic languages like Prolog are used to create "objects" as in OO, such as the Locomotive class in Figure 3.10 [HM90, McC88].

The given examples may not be intuitive at first — it is generally easier to see similarities in languages when code is implemented with same the design decisions and semantics. This is especially true for recursive functions since recursion is supported in all four paradigms.

```
quicksort(array) {
  if length(array) < 1 return
  quicksortRecursive(array, startIndex, endIndex)
}

// Recursive step of quicksort
quicksortRecursive(array, leftIndex, rightIndex) {
  if leftIndex < rightIndex {
    pivot := partition(array, leftIndex, rightIndex)
    quicksortRecursive(array, left, pivot-1)
    quicksortRecursive(array, pivot+1, right)
  }
}

// Order array so array[leftIndex] < pivot <= array[rightIndex]
partition(array, leftIndex, rightIndex) {
  pivot := array[rightIndex]
  iterLeft := leftIndex-1
  loop iterRight from leftIndex to rightIndex-1 {
    if array(iterRight < pivot) {
      iterLeft++
      swap(array, iterLeft, iterRight)
    }
  }
  swap(array, iterLeft+1, right)
  return iterLeft+1
}

swap(array, index1, index2) {
  temp = array[index1]
  array[index1] = array[index2]
  array[index2] = temp
}
```

Figure 3.5: In-place Quicksort Pseudocode

```
type Vector2D(dx:float, dy:float) =
  let len = sqrt(dx*dx + dy*dy)
  member this.DX = dx
  member this.DY = dy
  member this.Length = len
  member this.Scale(k) = Vector2D(k*dx, k*dy)
  member this.ShiftXY(x,y) = Vector2D(dx = dx+x, dy = dy+y)
  static member Zero = Vector2D(dx=0.0, dy=0.0)
```

Figure 3.6: Vector2D Class in F# [Dav]

```
open System.Drawing
type IShape =
  abstract Contains : Point -> bool
  abstract BoundingBox : Rectangle
let circle(center:Point,radius:int) =
  { new IShape with
    member x.Contains(p:Point) =
      let dx = float32 (p.X - center.X)
      let dy = float32 (p.Y - center.Y)
      sqrt(dx*dx+dy*dy) <= float32 radius
    member x.BoundingBox =
      Rectangle( center.X-radius, center.Y-radius, 2*radius+1,
      2*radius+1 )
}
```

Figure 3.7: IShape Interface in F# [Dav]

```
// Mutable Reference Cells
> let cell1 = ref1;; // create cell
val cell1 : int ref
> cell1;;
val it : int ref = { contents = 1 }
> !cell1;; // get contents
val it : int = 1
> cell1 := 3;;    // set contents
val it : unit = ()
> cell1;;
val it : int ref = { contents = 3 }
> !cell1;;
val it : int = 3

// Arrays
> let arr = [| 1.0; 2.0; 4.0 |];;
var arr : float[]
> arr.[1];;
val it : float = 1.0
> arr.[1] <- 3.0;;
val it : unit = ()
> arr;;
val it : float[] = [| 1.0; 3.0; 4.0 |]
```

Figure 3.8: Imperative Interaction in F# [Dav]

```
let unify2 (X,Y) (XX,YY) = X^=XX << Y^=YY
let unify3 (X,Y,Z) (XX,YY,ZZ) = X^=XX << Y^=YY << Z^=ZZ

let rec append2 args k () = (newvar4(), unify3 args) |>
fun((X, XS, YS, ZS), uniArgs) ->
  uniArgs(nil, YS, YS)         <|k<|()
  uniArgs((X^|XS),YS,(X^|ZS))  << append2(XS, YS, ZS) <|k<|()

let kEnd vars s = printfn "\ntrue:"
  for (v, tm) in vars do printfn "%s = %s" v (tm.ToString())

let lpRun vnames lpGen = let vars = List.map newvar vnames
  lpGen vars (kEnd (List.zip vnames vars)) ()

lpRun ["ZS"] (fun [ZS] -> append2((%%1 ^| %%2 ^| nil),
(%%3 ^| %%4 ^| nil), ZS))
// Output:
// true
// ZS = 1 | 2 | 3 | 4 | []
```

Figure 3.9: Logic Programming in F# [Dav]

```
MODULE locomotive.
LOCAL train.

make_train(train(S,Cl,Co),S,Cl,Co). color(train(S,Cl,Co),Cl).
speed(train(S,Cl,Co),S).
country(train(S,Cl,Co),Co).
journey_time(Train,Distance,Time) :-
  speed(Train,S), Time is Distance div S.
```

Figure 3.10: Locomotive Class in Prolog [HM90]

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (+) (fibonacci ((-) n 1)) (fibonacci ((-) n 2))
```

Figure 3.11: Fibonacci Solution in Haskell

```
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, SN) :-
    N1 is N - 1, N2 is N - 2,
    fibonacci(N1, SN1), fibonacci(N2, SN2),
    SN is SN1 + SN2.
```

Figure 3.12: Fibonacci Solution in λProlog

The notorious Fibonacci sequence is a well-known example of a recursive algorithm. The solutions displayed in Figures 3.11, 3.12, 3.13, and 3.14 (written to make them more intuitively similar and with the pre-condition to input non-negative arguments) may not be ideal for each language, although they do represent the same design decision to implement via cascading recursion. Generally, the recursive definition of the Fibonacci numbers is inefficient since the previous two numbers must be re-calculated, even though they had already been calculated before [webb]. In order to style similar solutions, each language's benefits are not taken full advantage of, such as Haskell's capability to define infinite lists or Java and Lua's capability for memoization to cache previous computations.

```
public int fibonacci(int n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Figure 3.13: Fibonacci Solution in Java

```
function fibonacci(n)
    if (n < 2) then
        return n
    end
    return fibonacci(n-1) + fibonacci(n-2)
end
```

Figure 3.14: Fibonacci Solution in Lua

Chapter 4

DSLs

After analyzing code for a sample of standard programs, a DSL was reverse-engineered to incorporate common concepts and patterns and used in the CPCG tool. Before delving further into the details of this software project and how DSLs are involved, however, it is important to understand what DSLs are; not to mention the reader benefits from learning more about internal and external DSLs.

This section of this report was influenced by the technical reports for SAGA [BC11] and GOOL [CC12] as these projects are similar to the CPCG tool in this sense.

4.1 What are DSLs?

DSLs belong to a class of special-purpose languages. Unlike general-purpose programming languages, such as Java and Lua, or general-purpose modeling languages, like UML, DSLs are generally small programming or specification languages dedicated to concisely expressing solutions to problems in a specific domain. During the design and development phases of any project, programmers should thoroughly understand the problems to be addressed, then determine if it is worthwhile to create DSLs or the problem is better solved by using existing general-purpose programming languages.

The goal of DSLs is to simplify programming by focusing on certain types of problems or domains and, thus, efficiently express narrower sets of solutions within the context of that scope [webd]. However, the limited scope of DSLs also limits their capabilities. (For example, SQL is intended for relational database queries, but CSS is used to describe the presentation semantics of documents written in markup languages.) In other words, DSLs are not intended to solve problems outside of their defined domain. They are typically not Turing complete, so programming tasks in one DSL is not always achievable using other DSLs.

DSLs are implemented by either interpretation or code generation [Fow]. Interpretation refers to reading in the DSL script and then executing it at runtime; of the two, it is the "easier" option to implement. Despite that fact, however, code generation is sometimes essential and the generated code is usually written in a high-level language, such as Java or Lua.

For non-programmers to accomplish specific tasks, DSLs are especially suitable as they are intended to be intuitive — hence, fluent and easier to read and write. Ideally, DSLs should improve the efficiency and productivity of users.

4.2 Internal versus External DSLs

DSLs are defined as internal (embedded) or external (freestanding) depending on how they are designed and implemented. Internal DSLs are akin to new languages within the host language as programming is done in the host language, but with the aid of some integrated constructs. They may not exist without the host language, especially if flow control is dependent on primitives within the host language. This type of DSL is particularly useful if the programmer wants the DSL to feel like or resemble a specific language.

By contrast, external DSLs are parsed independently of the host language [Fow]; only the functionality of these DSLs is specified during development. Any language and tools may be used to implement this type of DSL.

Both types of DSLs have advantages and disadvantages, so a variety of factors need to be considered when deciding between the two types for im-

plementation purposes. A clear benefit of internal DSLs is that they provide the grammar, tools, and parsers via the host language, so the effort to create them is relatively low. Furthermore, the developer may take advantage of the inherent capabilities within the host language. Unfortunately, it also means the developer and, thus, the expressiveness of the DSL, is restricted by the host language's own flexibility, limitations, and idiosyncrasies, not to mention limitations caused by the developer's knowledge of the host language. It is most advantageous to select host languages that are familiar, are highly flexible, and have as few restrictions and idiosyncrasies as possible, making it easier to exploit the metaprogramming capabilities of the host language. The code for internal DSLs is usually processed dynamically, implying that more extensive error checking and validation is required.

External DSLs, on the other hand, have the advantage of flexibility since they employ their own custom grammar and syntax. The developer selects the language's symbols, operators, constructs, and structures as desired to fit the intended domain. Generally, this customization allows for a more natural interface with the DSL, making it easier for domain experts to express ideas [BC11]. However, full compilers need to be developed, including intermediate representations of code, to parse and process external DSLs in order to map syntax to semantics; this extra step is a drawback with regards to programming complexity and the development workload. In general, external DSLs provide better control than internal DSLs when validating DSL syntax since the syntax is validated as the grammar is defined.

4.3 Relevance to Project

Although it is, of course, possible to implement an external DSL in the CPCG, an internal DSL was created for simplicity, using Haskell as the host language. The intended users of this DSL are programmers or individuals familiar with development. The domain of this project's DSL involves code generation for the four primary languages (λProlog, Java, Lua, and Haskell) and other secondary programming languages (C, C#, Lisp, Scala, and Prolog).

Chapter 5

Requirements

The goal of this project is to demonstrate that there exists a language of design from which one may generate code from abstract algorithms. In order to determine what the requirements are, the first step is to understand why they are needed. The following subsections explore the user, language, and software requirements of the CPCG project in more detail.

5.1 User Requirements

The user requirements for the CPCG project are brief, but important:

- Users are programmers or other individuals with technical backgrounds in at least one of the four programming paradigms.
- Users are responsible for ensuring that the selected design decisions are compatible (e.g., should not enter both list and array data structures). Furthermore, design choices that are unnecessary for the selected algorithm are ignored (e.g., selecting a pivot for the Fibonacci sequence).

5.2 Language Requirements

Adapted from the GOOL project [CC12], the following is a list of requirements and limitations that are incorporated in the design and implementation of the CPCG's DSLs:

- The internal language is purposefully limited such that it abstracts only fundamental cross-paradigm concepts and patterns; specialized features are instead detailed in the paradigm AST modules. For example, for- or while-loops are used in OO languages, but not in functional languages which use recursion to perform repetition.
- The internal language is expressive enough to produce a choice selection of simple, standard programs.
- The design language is limited to the choices selected by the researcher for the purpose of proving the existence of a tangible language of design. Concepts that are not included in the design language are subsequently unable to generate the desired code.

5.3 Software Requirements

This section describes the operation of the system rather than its specific behaviours, as follows:

- The cross-paradigm abstract concepts are demonstrated in the internal language. The CPCG uses these fundamental features to generate code in multiple languages, despite its singular design.
- The software uses generic pseudocode to resemble any of the four paradigms to translate code written in the DSL into intermediate representations of code in any paradigm and then into compilable source code.
- The process of generating code is unambiguous, complete, and consistent. Users are able to compile and run the generated code without any issues, regardless of the language it is written in.

- The CPCG addresses a number of software quality metrics, such as reliability, maintainability, testability, and understandability.

These software requirements are similarly adapted from [CC12].

Chapter 6

Design Methodology

The first step in the design phase consisted of understanding what was to be proven; that is to say, the existence of a language of design and of shared core concepts between different programming languages given an internal implementation and an external set of design choices. Prior to actually developing the language of design, however, intermediary steps were taken to explore and work with some anticipated common features. The design methodology of this project was influenced by the GOOL project [CC12], which generates code in OO languages. The CPCG essentially started out as an expansion of the GOOL project to generate code in cross-paradigm languages, rather than OO alone, before the influence of design choices was explored.

6.1 DSL Design

A key design decision of this project was the use of internal DSLs. The DSL for the internal language abstractly links together the assessed programming languages, regardless of their associated paradigm(s), and simplifies the intermediate representations of code. Prior to constructing the DSL for the CPCG, a series of solutions for the standard quicksort function were developed or collected (from sources such as [Zhu13], etc.) and examined to create a set of ideal solutions. (These solutions are included for reference in the

```
quicksort(items)
  if items.length < 1 do nothing
  else
    choose index
    pivot := items[index]
    partition pivot less rest
    return quicksort(less) ++ pivot ++ quicksort(rest)
```

Figure 6.1: Quicksort Pseudocode

appropriate paradigm sections in Chapter 3.) In order to identify enough shared concepts from these solutions to start developing the DSL, quicksort was translated into the pseudocode in Figure 6.1 to provide a high-level description of the algorithm.

The next steps involved abstracting the common concepts by using an AST. The AST for the DSL is composed of data types and structures in Haskell. Using individual analysis as well as referencing structures on [CC12], some features that were initially identified as patterns are

- Program
- Function
- Nodes: Variable, Integer, Selector (such as length)
- Expressions: Node, Let, If, Concatenation, Partition, Choose, Operation, Self, Unit
- Operations: Less, Minus, Plus, Times, Divide, etc.

These concepts were used to develop the initial internal AST type as displayed in Figure 6.2.

The DSL was updated continuously based on the needs of the code generator as additional code samples and research papers were used to help identify concepts of and promote a richer language. A sample of the final internal AST type is included in Section 7.1, Figure 7.1.

Information about the design DSL may be found in Chapter 8.

```
data Program = Program Name [Function]
type Name = String

data Function = Funct [Expr]

data Expr = NodeExpr Node
          | LetExpr (Binding,Expr) [Expr]
          | IfExpr Conditional
          | ConcatExpr Concatenation
          | PartExpr Node Node
          | ChooseExpr Node Node
          | OperExpr Operations
          | SelfExpr [Expr]

data Concatenation = Concat2 Node Node
                   | Concat3 Node Node Node

data Conditional = IfExpr Expr Expr Expr

data Binding = BVar Variable
             | BPair (Variable,Variable)
type Variable = String

data Node = Var Variable
          | Int Integer
          | Sel Variable Property

data Property = Length

data Operations = Less Node Node
                | Minus Node Node
                | Plus Node Node
                | Times Node Node
                | Divide Node Node
```

Figure 6.2: Initial Internal AST Type

6.2 Intermediate Representations of Code

Due to general syntactic and semantic differences of code across the four paradigms, rendering code from the internal DSL immediately to the intended language is not easily accomplished. Therefore, representations of code following the format of the intended paradigm (but written in the project's DSL) are used as intermediary steps, one per paradigm.

To generate "almost" source code, the CPCG starts by using the ASTs specific to the paradigms along with the algorithm implementations written in the DSL, such as ImplQuicksort. Other modules, referred to in this report as intermediary or "To" modules, are then used to convert programs from the internal AST type to the paradigm AST types.

6.3 Pretty Printing

Once programs are rendered into their intermediate representations by the process mentioned above, applying the syntactical details to the code is a fairly straightforward process. Each implementation language has its own pretty printer to build compilable, working source code using the PrettyPrint package [webc]; although the reader should note that some necessary functions or structures are assumed to exist, such as the init function in λProlog to return all but the last element in lists or the List structure (and its associated functions) in C. The printers and their accompanying dictionaries should only add syntactical details to the structures, any abstraction details having already been applied in the other steps of the code generation process.

Chapter 7

CPCG Program

This chapter gives an overview of the CPCG software tool, including the internal language, paradigm-specific modules, and language-specific modules. There are also sections describing compromises made in this project as well as some suggestions for future enhancements.

7.1 Internal Language

Recall that the internal language represents some of the core concepts shared between implementation languages within the programming paradigms. To generate code, the internal language is used with intermediate representations of paradigm code and pretty printers for the individual programming languages; these intermediate representations directly produce the connections between the DSL and the compilable code. A sample of the internal AST type is included in Figure 7.1.

Some structures were removed from the initial DSL (and perhaps moved to the paradigm modules) as they were later discovered not to be common across all paradigms, while other structures and types were added as befit other algorithms to be generated. Examination of quicksort solutions in Haskell (Figure 3.3) and Java (Figure 3.1), for example, observes that two different structures are used to store the data to be sorted — list and array.

```
data Program = Program [D.Choices] Name [Declaration] [Function]
type Name = String

data Function = Funct Access Declaration [Expr]

data Access = Public | Private

data Declaration = FDecl (Type,[D.Choices]) Name [Declaration]
                 | PDecl (Type,[D.Choices]) Name

data Expr  = NodeExpr Node
           | LetExpr (Binding,Expr) [Expr]
           | IfExpr Expr [Expr] [Expr]
           | ConcatExpr [D.Choices] Concatenation
           | OperExpr Operations
           | SelfExpr Type [Expr]
           | CallExpr Type Name [Expr]

data Concatenation = List Node Expr | Concat Name Expr Expr

data Binding = BVar (Type,[D.Choices]) Variable
             | BPair (Binding,Binding)
type Variable = String

data Node = Var Variable
          | Int Integer
          | Sel Node Property

type Index = Node
data Property = Length [D.Choices]
              | Get [D.Choices] Index Index
              | Set [D.Choices] Index Node

data Operations = Less Node Node
                | Greater Node Node
                | Minus Node Node
                | Plus Node Node
```

Figure 7.1: Sample from Final Internal AST Type

The DSL has to represent both these ideas without compromising abstraction in one language or the other.

7.2 Paradigm Modules

This section outlines the modules used to generate code in the various paradigms.

7.2.1 Paradigm ASTs

The ASTFunctional, ASTImperative, ASTLogic, and ASTOO files store the ASTs for the programming paradigms, one for each of the four. Intuitively, there are many data structures that these modules share with the internal AST type, although there are also some that differ based on the needs of the language and the ways in which algorithms are generated. Consider binding structures (Figure 7.2), for instance. Languages such as λProlog (logic) and Haskell (functional) allow pair bindings, while Java (OO) and Lua (imperative) generally only make use of single bindings, unless additional structures or classes are created or call-by-reference is used. Note, however, that the AST for OO includes a pair binding structure since one of the OO languages explored — Scala — does, in fact, allow pair bindings and must be accommodated.

Another item of note is the abstraction of types. Although some languages (such as Java) require specific types to be declared in functions and their parameters, other languages (such as λProlog) do not require this level of detail. This has less to do with the type of paradigm these languages are in and more to do with type inference. Some languages require types to be explicitly given because inference is undecidable in certain situations.

7.2.2 Intermediary Modules

The intermediary modules restructure code written in the internal language to the appropriate paradigm style, producing the aforementioned intermediate representations of code. There are generally few, but important, changes

```
-- Logic AST
data Binding = BVar Variable
             | BPair (Variable,Variable)
  deriving Show

-- Functional and OO ASTs
data Binding = BVar (Type,Struct) Variable
             | BPair (Variable,Variable)
  deriving Show

-- Imperative AST
data Binding = BVar (Type,Struct) Variable
  deriving Show
```

Figure 7.2: Binding Structures

that take place here that alter the format of the code. Code for languages within the same paradigm is generated from the same intermediary modules as their code is semantically similar, so syntax has little relevance.

Sample code from the intermediary module for functional programming is shown in Figure 7.3; the modules for the other paradigms are designed similarly.

7.3 Language Printers

The pretty printers for Java, Lua, Haskell, and λProlog are the primary printer files as these languages were originally selected for this software project. Using the existing paradigm modules, however, the researcher developed printers for the following languages in addition to the first four:

- Functional: Lisp and Scala
- OO: Scala and C#
- Imperative: C

As portrayed above, Scala has the curious trait that it is used for both functional and OO programming. In fact, numerous programs are successfully

```
toFunct :: I.Program -> F.Module
toFunct (I.Program ch nm decllist fnlist) =
    if any (`elem` ch) [D.Para Log,D.Para Imp,D.Para OO]
        then error "Invalid paradigm choice"
        else if any (`elem` ch) [D.Lang C, D.Lang Java,
        D.Lang LP, D.Lang Prolog, D.Lang Lua]
            then error "Invalid language choice"
            else if (elem (D.Loop Iter) ch)
                then error "No loop iteration available"
                else if (elem (D.Lang Haskell) ch && elem
                (D.Lib Part) ch)
                    then F.Module nm ["Data.List (partition)"]
                    fns
                    else if (elem (D.Lang Haskell) ch && elem
                    (D.Str Array) ch
                        then F.Module nm ["Control.Monad.ST",
                        "Data.Array.ST", "Data.Foldable",
                        "Control.Monad"] fns
                        else F.Module nm [] fns
  where
    fns = map build fnlist
```

Figure 7.3: Sample from Intermediary Module for Functional Programming

generated using functional paradigm modules and design choices and OO paradigm modules and design choices, once again giving evidence to the shared semantic core between cross-paradigm languages.

Sample code from the Haskell pretty printer is included in Figure 7.4. Notice how "self" is used as a parameter in almost every line of all the pretty printers, so that functions have the capability to call themselves recursively as needed. Also note how the code generated for Haskell and Java clearly differ at this point. For instance, Java uses infix notation while Haskell (in this software) uses prefix notation.

7.4 Compromises

This project is primarily focused on proving that common features exist between the four paradigms and that a language of design is tangible. The generated code was mainly verified for compilability and workability rather than idealness; in some cases, even the high-level code contains compromises. Due to scope and time constraints, abstracting cross-paradigm concepts required a fair bit of flexibility and compromise during the development of the CPCG, at least during these early stages of research.

Consider only the last line of the high-level quicksort pseudocode shown in Figure 7.5 as a compromise example. The keyword "return" is used, although this concept is not always needed since it is implicit in some languages, such as Haskell. The line also shows the concatenation of "less" (containing elements less than the pivot value), "pivot", and "rest" (containing all other elements); while this is valid for lists, concatenation is not required for arrays. In these and other comparable situations, concepts are nevertheless included in the internal language to represent the ideas behind them, explicit or not, which may be seen as a compromise to the DSL's integrity to some users.

A similar idea was applied at the code generation level. Compare the ideal quicksort solution in Haskell (Figure 3.3) to the generated solution (Figure B.1); the first thing one may notice is the use of len (i.e., length) and indices in both the array and list versions of the algorithm. It is apparent

```
printHaskell :: Module -> Doc
printHaskell (Module nm imps fns) = text "module" <+> text nm
    <+> text "where" $$ impts imps $$ vcat (map build fns)

build :: Function -> Doc
build (Funct (FDecl _ nm paramlist) exs) = text nm
    <+> params paramlist <+> equals $+$ nest 4 (expr nm exs)

params :: [Declaration] -> Doc
params [] = empty
params (x:xs) = foldl1 (<+>) (map (\x -> pdecls x) (x:xs))
  where
    pdecls (PDecl (t,s) v) = text v

impts :: [String] -> Doc
impts [] = empty
impts (x:xs) = text "import" <+> text x $$ impts xs

expr :: String -> Expr -> Doc
expr self (NodeExpr nd) = node self nd
expr self (IfExpr ex1 ex2 ex3) = text "if" <+> expr self ex1
    $+$ nest 4 (text "then do" $+$ nest 4 (expr self ex2)
    $$ text "else do" $+$ nest 4 (expr self ex3))
expr self (OperExpr o) = oper self o
expr self (SelfExpr t p) = callexpr self t self p
expr self (LetExpr b ex) = letexpr self b
    $+$ nest 4 (expr self ex)
expr self (CallExpr t nm p) = callexpr self t nm p
expr self (AsstExpr nd ex) = node self nd <+> text "<-"
    <+> expr self ex
expr self (ConcatExpr c) = concatexpr self c
expr self (Exprs ex1 ex2) = expr self ex1 $$ expr self ex2
expr self (PairExpr (ex1,ex2)) = parens $ expr self ex1
    <^> expr self ex2
expr self (PartExpr p i) = text "partition"
    <+> parens (less <+> node self p) <+> node self i
expr self (ArrExpr (StartIndex,EndIndex nd)) = integer 0
    <+> parens (node self nd <> minus <> integer 1)
```

Figure 7.4: Sample from Haskell Pretty Printer

```
return quicksort(less) ++ pivot ++ quicksort(rest)
```

Figure 7.5: Compromise Example

that these parameters are not necessary in any of the explored languages for the list version of quicksort. However, due to their necessity in the array version, it is included in the core program file for simplicity and, hence, is generated in both versions of quicksort.

Another compromise is the assumption that certain needed structures or methods exist elsewhere, such as the list structure and methods (e.g., concat and length) for C's list version of quicksort (Figure 7.6, adapted from [Aro]).

7.5 Future Enhancements

Some recommendations for potential future enhancements of the CPCG are

- Expand the DSL to increase the project's scope and abstract more concepts, accommodating a wider array of programs to generate.
- Alter the format of the DSL as needed to make it more user-friendly and human-readable; the DSL appears to become less comprehensive as more advanced concepts are added.
- Add or remove concepts from the DSL as more programming languages are added to the code generator since shared concepts in the existing languages may not necessarily share these same concepts with the new languages.
- Add new choices to the design language to allow for more varied code generation.
- Modify CPCG source code to generate ideal solutions rather than just compilable, working code.

```
struct List {
    long val;
    struct List *next;
};

long length(struct List *tlist) {
    struct List *ptr = tlist;
    long count = 0;
    while(ptr != NULL) {
        count = count + 1;
        ptr = ptr->next;
    }
    return count;
}

struct List* concat(struct List* l1, struct List* l2) {
    long x = length(l1);
    long y = length(l2);
    if (x >0 && y > 0) {
        end_of_list(l1)->next = l2;
    }
    if(x == 0) {
        return l2;
    }
    return l1;
}
```

Figure 7.6: List Structure and Methods in C

Chapter 8

Design Language

The design language (displayed in full in Figure 8.1) is used to supplement the other modules by defining possible choices that users may provide to the CPCG tool; these choices affect the subsequent generated code. For example, the partition function used in the quicksort algorithm has different requirements based on the design decisions of the user, such as using array versus list data structures or using in-place sorting versus copying.

The types of choices available to users are divided into six categories:

1. Pivot: Select the head (start) or last (end) element of the chosen data structure. The middle element may only be used in the array data structure.

2. Structure: Select the data structure to be examined. None is used if no data structure is present in the algorithm, such as in the "Hello World" implementation; the other available choices are list and array.

3. Loop: Select recursion or iteration as the form of repetition in the algorithm.

4. Libraries: If available within the language libraries, select built-in partition or concatenation functions rather than using the CPCG to generate its own.

5. Language: There are currently nine languages to select from in the CPCG — Haskell, Java, Lua,λProlog, Lisp, C, Scala, Prolog, and C#.

6. Paradigm: Scala is the only language that needs a specific paradigm chosen. For the other languages, the choice of paradigm is implicit in the CPCG.

When exploring any algorithm, the user is responsible for selecting enough applicable design choices to generate the desired output; unnecessary or invalid choices are ignored or errors are produced. Again using quicksort as an example — this algorithm should be given design input for pivot, structure, loop, and language at the base level. Library choices are optional and, generally, so is the choice of paradigm (except for Scala which must be specified as OO or functional to ensure the correct intermediary module is used). On the other hand, for examples such as Fibonacci and factorial, pivot, structure, and libraries, are not relevant design choices.

The design language is developed in such a way that it is relatively easy for programmers to add new design choices (as mentioned in Section 7.5), either to the existing categories or by creating new categories entirely.

```
type Library = String

data Choices = Piv Pivot
             | Str Structure
             | Loop Loop
             | Lib Libraries
             | Lang Language
             | Para Paradigm
  deriving Eq

data Pivot = Head | Mid | Last
  deriving Eq

data Structure = None | List | Array
  deriving Eq

data Loop = Rec | Iter
  deriving Eq

data Libraries = Part | Concat
  deriving Eq

data Language = Haskell
              | Java
              | C
              | LP
              | Prolog
              | CSharp
              | Scala
              | Lua
              | Lisp
  deriving Eq

data Paradigm = Func | Log | OO | Imp
  deriving Eq
```

Figure 8.1: Design Language

Chapter 9

Development and Implementation Details

9.1 Problems and Changes

Creating the initial DSL was by no means an easy feat, nor was translating the algorithms for standard programs from the internal DSL to intermediate representations, then to source code in multiple languages. Many of these languages were not familiar to the developer before this project, including the DSL's host language, Haskell. More often than not, the differences between the languages overshadowed the similarities, making it difficult to identify and abstract the needed concepts.

After creating skeletal versions of the ASTs, it was easier and more efficient to start development with simpler algorithms and examples than quicksort, particularly ones that are intuitively similar in each of the four languages (semantically and syntactically) to get the code up and running, then backtrack and adapt the older code to the changes. In this situation, the design choices were initially, but temporarily, ignored to be able to grasp the more basic concepts of the project and what the ultimate goal is. As the project developed, more algorithms and languages were added, strengthening the theory that the syntax and, in some ways, semantics of each language are not very relevant.

Eventually, the concept of design choices was added back into the project. This addition, of course, presented its own set of difficulties. What is the best way to represent the design choices? At what level (i.e., program, function, expression, etc.) do the design choices need to be considered? How do the design choices affect the final generated source code? Ultimately, the design choices were added to the project via the design AST, the structures and types of which were later integrated into the existing paradigm modules.

Adding design choices to the project did not influence the pretty printers as the design is reasonably assumed to modify the code at the design/semantic level (i.e., in the "To" modules) rather than the syntactic level of code generation. One observation was made regarding the design choice for looping via iteration or recursion: any variables that are modified within the iterative loop need to be passed as parameters for the recursive version of the function.

The reader should note that it is not necessarily possible to integrate the design choices into every language due to the time constraints, scope of the project, and/or nature of the algorithm (e.g., there are no pivot choices in the exponent algorithm). However, there are a fair amount of design choices and example algorithms to demonstrate if and how design choices affect code generation.

Over time, both the project and the experience of the developer progressed. As the DSLs became more expressive and complete in the sense that they involved more or less features as required, it became less difficult to abstract the concepts or accommodate the design choices and obtain the intended output.

Chapter 10

Testing

Testing the CPCG was an ongoing and sometimes tedious task. With every new language, algorithm, and design change added, new modules needed to be designed, developed, and tested to ensure compatibility with the existing modules; not to mention the modification of previous files. If modules from earlier stages of the code generation process were updated, especially the internal AST type, all subsequent modules then had to be modified and tested as well to accommodate the changes. For example, the IfExpr and Conditional data structures were modified in the internal AST type to a single IfExpr structure (Figure 10.1). Similarly, ConcatExpr was modified to represent the difference between concatenating an element to a list and concatenating two lists (Figure 10.2), both of which are needed for the immutable list version of quicksort.

```
-- Before making changes
data Expr = IfExpr Conditional
data Conditional = If Expr Expr Expr

-- After Making Changes
data Expr = IfExpr Expr [Expr] [Expr]
```

Figure 10.1: IfExpr Modifications in Internal AST Type

```
data Expr = ConcatExpr Concatenation

-- Before Making Changes
data Concatenation = Concat2 Node Node
                   | Concat3 Node Node Node

-- After Making Changes
data Concatenation = List Node Expr --element with list
                   | Concat Expr Expr --two lists
```

Figure 10.2: Concatenation Modifications in Internal AST Type

These two updates took place in the internal AST type, the root module for all code generation in the CPCG. As such, more changes were required to be made to the algorithm implementations and paradigm modules, then all the modules were tested using the methods discussed in the next sections.

10.1 The Box Approach

10.1.1 Black-Box Testing

Black-box test cases were designed based around the requirements and specifications of the software project and applied to virtually every level of testing (as described in more detail in Section 10.2). This type of testing was used frequently during and after the development of the CPCG, especially in situations where changes were made to existing modules. For example, code is generated based on a language choice, design choices, and algorithm choice. If the printed implementation matched the expected printed output code, the test passed; otherwise, the test failed and notes were made to prepare for white-box testing (Section 10.1.2) to determine what the problem was.

As the project progressed, the design changed to better suit the software requirements and simplify the code by removing redundancies. Entire blocks of code were constantly being modified and moved within or across the Haskell modules, making black-box testing a simple method to use to ensure the input parameters are accepted as expected (such as values and

types since Haskell is a strongly-typed programming language) and the desired results were still produced.

10.1.2 White-Box Testing

White-box testing was used regularly in conjunction with black-box testing during the development and maintenance stages for the project. Although white-box testing is applicable to every level of testing, it was more commonly used at the unit level for the CPCG while black-box testing (Section 10.1.1) was used for higher-level testing.

White-box testing was not only used to confirm that the project code works as expected for every path, control flow, data flow, edge case, error case, etc.; it was also used to remove data structures and functions that were no longer being used. As the project developed, new functionality was needed while older functionality became redundant or unnecessary, such as the previously mentioned if-else and concatenation expression examples from the internal AST type (Figures 10.1 and 10.2).

10.2 Level Testing

According to the Software Engineering Body of Knowledge (SWEBOK), there are three big test stages that conceptually distinguish level testing [Soc05]:

1. Unit Testing: Verifies the functioning of components in isolation of software pieces which are separately testable; test cases were typically designed at the class level (for OO design) or module level (logic and functional design) by the developer to ensure the specific function performed as expected.

2. Integration Testing: Verifies the interaction between software components iteratively or all at once. This iterative approach to integration testing was usually preferred in the CPCG as it is easier and more efficient to identify which units were causing the problems and fix them.

3. System Testing: Verifies the behaviour of the whole system to ensure the requirements and specifications are met and deals with edge and error cases.

The developer's approach to level testing adopted the bottom-up perspective whereby the lowest level units were tested first (unit testing) before integration and system testing took place, rather than the top-down point of view to examine the system first before testing the software's integration and individual units [Soc05].

Chapter 11

Conclusions

The results of this project demonstrate common concepts and patterns of the cross-paradigm languages examined and, by extension, other languages within the four main paradigms as well. Furthermore, the outputted languages are generated equally well from the DSL.

New languages may be added to the software project with little difficulty based on the existing modules as languages within the same paradigm were discovered to be very semantically similar, meaning the syntax has little importance.

Appealing and ambitious changes to the CPCG, such as adding more concepts to the DSL to bring it closer to an ultimate goal of making it Turing complete, make it more usable and efficient.

This tool is sufficient to prove the existence of common concepts and patterns across programming paradigms. However, it is not likely to be used in non-academic environments as it is hardly feasible to expand it so extensively for general use in industry.

Knowledge and use of the CPCG familiarizes individuals with the similarities in cross-paradigm languages, making it easier to apply this knowledge and learn new programming languages more quickly and efficiently.

Finally, with sufficient evidence, this project proves the existence of abstract algorithms and a language of design that is actually tangible and not just theoretical.

Bibliography

[Aro] Himanshu Arora. C Linked List Example. `http://www.thegeekstuff.com/2012/08/c-linked-list-example/` [Last accessed: June 2013].

[BC11] Lucas Beyak and Jacques Carette. SAGA: A Story Scripting Tool for Video Game Development. Technical report, McMaster University - Department of Computing and Software, April 2011.

[CC12] Jason Costabile and Jacques Carette. GOOL: A Generic OO Language. Technical report, McMaster University - Department of Computing and Software, April 2012.

[Cla00] Robert G. Clark. *Comparative Programming Languages*. Addison Wesley, 3rd edition, November 2000.

[Dav] Dr. Rowan Davies. Computer Science and Software Engineering Programming Paradigms. `http://undergraduate.csse.uwa.edu.au/units/CITS3242/` [Last accessed: June 2013].

[Fow] Martin Fowler. `http://martinfowler.com/` [Last accessed: June 2013].

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, January 2001.

[HM90] Joshua S. Hodas and Dale Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. Technical re-

port, University of Pennsylvania - Department of Computer and Information Science, May 1990.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. Technical report, Yale University - Department of Computer Science, September 1989.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, April 2011.

[McC88] Francis Gregory McCabe. Logic and Objects: Language, application, and implementation. Technical report, Imperial College of Science and Technology, November 1988.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, September 1996.

[Mit02] John C. Mitchell. *Concepts in Programming Languages.* Cambridge University Press, 1st edition, October 2002.

[Nor] Kurt Normark. Overview of the four main programming paradigms. `http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html` [Last accessed: June 2013].

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell.* O'Reilly Media, 1st edition, December 2008.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, January 2002.

[Seb01] Robert W. Sebesta. *Concepts of Programming Languages.* Addison Wesley, 5th edition, July 2001.

[Soc05] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge.* IEEE Computer Society Press, March 2005.

[weba] Classes (C# Programming Guide). `http://msdn.microsoft.com/en-us/library/x9afc042.aspx` [Last accessed: June 2013].

[webb] Fibonacci numbers. http://en.literateprograms.org/Fibonacci_numbers [Last accessed: June 2013].

[webc] HackageDB. http://hackage.haskell.org [Last accessed: June 2013].

[webd] Java World. http://www.javaworld.com/ [Last accessed: June 2013].

[webe] The Haskell Programming Language. http://www.haskell.org/haskellwiki/Haskell [Last accessed: June 2013].

[Zeh] Norbert Zeh. CSCI 3136: Principles of Programming Languages. http://web.cs.dal.ca/~nzeh/Teaching/3136/ [Last accessed: June 2013].

[Zhu13] Lijun Zhu. personal communication, June 2013.

Appendix A

High-level Module Descriptions

A.1 ASTInternal

AST to define the internal language. Used in the implementation modules to express algorithms.

A.2 ASTDesign

AST to define the design choices that users may input for generating code.

A.3 Main

Provides the user interface to generate code with the parameters: file path, algorithm name (must match expected/existing name), and design choices from the design module.

A.4 Paradigm Modules

A.4.1 ASTFunctional

AST to define the functional paradigm for writing code to be generated, including modified versions of the internal language's data types and structures as well as additional ones to abstract functional-only concepts.

A.4.2 ASTOO

AST to define the OO paradigm for writing code to be generated, including modified versions of the internal language's data types and structures as well as additional ones to abstract OO-only concepts.

A.4.3 ASTC

AST to define the imperative paradigm for writing code to be generated, including modified versions of the internal language's data types and structures as well as additional ones to abstract imperative-only concepts.

A.4.4 ASTLogic

AST to define the logic paradigm for writing code to be generated, including modified versions of the internal language's data types and structures as well as additional ones to abstract logic-only concepts.

A.5 Intermediary "To" Modules

A.5.1 ToFunctional

Module to map DSL concepts from ASTInternal to functional concepts from ASTFunctional and start resembling functional code design.

A.5.2 ToOO

Module to map DSL concepts from ASTInternal to OO concepts from ASTOO and start resembling OO code design.

A.5.3 ToImperative

Module to map DSL concepts from ASTInternal to imperative concepts from ASTImperative and start resembling imperative code design.

A.5.4 ToLogic

Module to map DSL concepts from ASTInternal to logic concepts from AST-Logic and start resembling logic code design.

A.6 Pretty Printer Modules

A.6.1 Helpers

Provides helper functions for frequently used printer syntax, such as pipe "|" and dblslash "//".

A.6.2 Primary Languages

PrintHaskell

Maps "almost" functional code from ToFunctional to the proper syntax and generates compilable, working source code in Haskell.

PrintJava

Maps "almost" OO code from ToOO to the proper syntax and generates compilable, working source code in Java.

PrintLua

Maps "almost" imperative code from ToImperative to the proper syntax and generates compilable, working source code in Lua.

PrintLambdaProlog

Maps "almost" logic code from ToLogic to the proper syntax and generates compilable, working source code in λProlog.

A.6.3 Secondary Languages

PrintCSharp

Maps "almost" OO code from ToOO to the proper syntax and generates compilable, working source code in C#.

PrintLisp

Maps "almost" functional code from ToFunctional to the proper syntax and generates compilable, working source code in Lisp. PrintLisp is not currently able to generate source code for the quicksort algorithm.

PrintC

Maps "almost" imperative code from ToImperative to the proper syntax and generates compilable, working source code in C.

PrintScala

Maps "almost" OO code from ToOO to the proper syntax and generates compilable, working source code in Scala; also has the capability to map "almost" functional" code from ToFunctional to the proper syntax and generate compilable, working source code in Scala.

A.7 Implementation Modules

A.7.1 Quicksort Algorithms

Quicksort

Implementation of the Quicksort algorithm written in the DSL from ASTInternal.

QuicksortHelpers

Implementation of helper functions (used in the Quicksort algorithm) written in the DSL from ASTInternal.

A.7.2 Iterative Algorithms

Fibonacci

Implementation of the Fibonacci algorithm written in the DSL from ASTInternal.

Factorial

Implementation of the Factorial algorithm written in the DSL from ASTInternal.

Exponent

Implementation of the Exponent algorithm written in the DSL from ASTInternal.

GCD LCM

Implementations of the greatest common divisor (GCD) and least common multiple (LCM) algorithms written in the DSL from ASTInternal.

A.7.3 Other Algorithms

Palindrome

Implementation of the Palindrome algorithm written in the DSL from ASTInternal.

Maximum

Implementation of the Maximum algorithm written in the DSL from ASTInternal to find the maximum of two numbers.

Hello

Implementation of the Hello algorithm written in the DSL from ASTInternal to print “Hello World!”.

Appendix B

Generated Quicksort Solutions

B.1 Haskell (Functional)

The generated quicksort solution in Haskell is shown in Figure B.1 and the ideal solution in Figure 3.3.

B.2 λProlog (Logic)

The generated quicksort solution in λProlog is shown in Figure B.2 and the ideal solution in Figure 3.4.

B.3 Java (OO)

The generated quicksort solution in Java is shown in Figures B.3 and B.4 and the ideal solution in Figure 3.1.

B.4 Lua (Imperative)

The generated quicksort solution in Lua is shown in Figures B.5 and B.6 and the ideal solution in Figure 3.2.

```
module QuicksortProg where
quicksort items len =
    if ((<) len 1
        then do
            items
        else do
            quicksortRec items 0 ((-) len 1)
quicksortRec items leftIndex rightIndex =
    if (>=) leftIndex rightIndex
        then do
            items
        else do
            let pivot = (head items) in do
                let (less,rest) = partition (< pivot)
                (tail items) in do
                    quicksortRec less 0 ((-1) (length less) 1)
                    ++ (pivot:quicksortRec rest 0
                    ((-) (length rest) 1))
```

Figure B.1: Generated Quicksort Solution in Haskell

```
quicksort(Items,Len,Squicksort) :-
    Len < 1-> Squicksort = Items;
    quicksortRec(Items,0,Len-1,Squicksort).
quicksortRec(Items,LeftIndex,RightIndex,SquicksortRec) :-
    LeftIndex >= RightIndex->SquicksortRec = Items;
    [Head|Tail] = Items,Pivot is Head,
    partition(>(Pivot),Tail,Less,Rest),
    length(Less,LenLess),
    quicksortRec(Less,0,LenLess - 1,SLess),
    length(Rest,LenRest),
    quicksortRec(Rest,0,LenRest - 1,SRest),
    append(SLess,[Pivot|SRest],SquicksortRec).
```

Figure B.2: Generated Quicksort Solution in λProlog

```
public class QuicksortProg {
    public Integer[] quicksort(Integer[] items,int len) {
        if (len < 1) {
            return items;
        }
        else {
            return quicksortRec(items,0,len-1);
        }
    }
    private Integer[] quicksortRec(Integer[] items,
    int leftIndex,int rightIndex) {
        if (leftIndex >= rightIndex) {
            return items;
        }
        else {
            int[] pivot = rightIndex;
            pivot = partition(pivot,items,leftIndex,rightIndex);
            quicksortRec(items,leftIndex,pivot-1);
            quicksortRec(items,pivot+1,rightIndex);
            return items;
        }
    }
    private void partition(int pivotIndex,Integer[] items,
    Integer leftIndex,Integer rightIndex) {
        int pivotValue = items[pivotIndex];
        swap(items,pivotIndex,rightIndex);
        int swapIndex = partitionIter(leftIndex,leftIndex,
        rightIndex,items,pivotValue);
        swap(items,swapIndex,rightIndex);
        return swapIndex;
    }
```

Figure B.3: Generated Quicksort Solution in Java (Part I)

```
private int partitionIter(int swapIndex,int i,
int rightIndex,Integer[] items,int pivotValue) {
        while (i < rightIndex) {
            int iVal = items[i];
            if (iVal < pivotValue) {
                swap(items,i,swapIndex);
                swapIndex = swapIndex+1;
                i = i+1;
            }
            else {
                i = i+1;
            }
        }
        return swapIndex;
    }
    private void swap(Integer[] items,int i,int j) {
        int iVal = items[i];
        int jVal = items[j];
        items[i] = jVal;
        items[j] = iVal;
    }
}
```

Figure B.4: Generated Quicksort Solution in Java (Part II)

```
function quicksort(items,len)
    if (len < 1) then
        return items
    else
        return quicksortRec(items,1,len)
    end
end
function quicksortRec(items,leftIndex,rightIndex)
    if (leftIndex >= rightIndex) then
        return items
    else
        local pivot = rightIndex
        pivot = partition(pivot,items,leftIndex,rightIndex")
        quicksortRec(items,leftIndex,pivot - 1)
        quicksortRec(items,pivot + 1,rightIndex)
        return items
    end
end function partition(pivotIndex,items,leftIndex,rightIndex)
    local pivotValue = items[pivotIndex]
    swap(items,pivotIndex,rightIndex)
    local swapIndex = partitionIter(leftIndex,leftIndex,
    rightIndex,items,pivotValue)
    swap(items,swapIndex,rightIndex)
    return swapIndex
end
function partitionIter(swapIndex,i,rightIndex,items,pivotValue)
    while (i < rightIndex) do
        local iVal = items[i]
        if (iVal < pivotValue) then
            swap(items,i,swapIndex)
            swapIndex = swapIndex + 1
            i = i + 1
        else
            i = i + 1
        end
    end
    return swapIndex
end
```

Figure B.5: Generated Quicksort Solution in Lua (Part I)

```
function swap(items,i,j)
    local iVal = items[i]
    local jVal = items[j]
    items[i] = jVal
    items[j] = iVal
end
```

Figure B.6: Generated Quicksort Solution in Lua (Part II)