Embedding Programming Languages: PROLOG in HASKELL

Mehul Solanki

University of Northern British Columbia

solanki@unbc.ca

April 10, 2016

Introduction I

Overview

To provide background, I will discuss:

- Programming languages and HASKELL and PROLOG.
- Classification.
- Programming paradigms and declarative programming.
- Functional and logic programming.
- Bringing programming languages together.

Programming Languages

A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer. For example: C, JAVA.

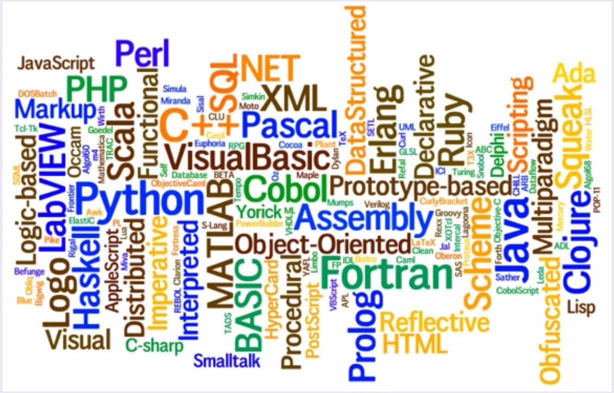

Figure: The Universe of Programming Languages

HASKELL

HASKELL is an advanced purely-functional programming language. In particular, it

- is polymorphically statically typed;
- has type inference;
- is lazy;
- and is purely functional.

Figure: HASKELL Programming Language

PROLOG

PROLOG is a general purpose logic programming language with over 20 distributions. It borrows its basic constructs from logic. However, it has an order is defined for both clauses in the program, and for goals in the body of the clause.

Figure: SWI PROLOG Distribution

Programming Language Paradigms

A programming paradigm is a fundamental style of computer programming, a way of building the structure and elements of computer programs.
For example, Object Oriented Programming is programming language paradigm.

Figure: Programming Paradigms

Classification

Programming languages are classified into paradigms depending on their characteristics and features.
For example, JAVA is an Object Oriented Programming Language.

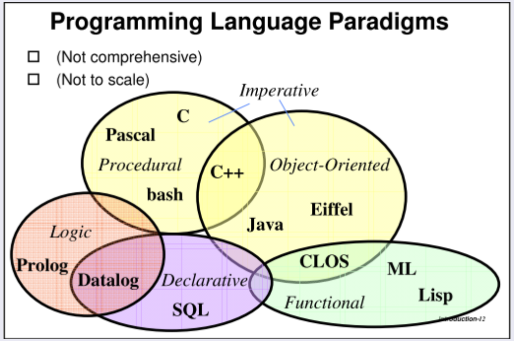

Figure: Classification of Programming Languages

Declarative Programming

- Declarative style of programming, describes (declaratively) what to do and not (operationally) how to do it.
- Programming in a functional language consists of building definitions and using the computer to evaluate expressions.
- In logic programming, a program consists of a collection of statements expressed as formulas in symbolic logic.
- For example, HASKELL (functional language) and PROLOG (logic language).

Functional Programming

- Functional programming is all about functions.
- λ-calculi, is a formal system in mathematical logic for expressing computations.
- For example, HASKELL uses the Damas-Hindley-Milner type system provides the ability to give a most general type to a function or program.

```haskell
add :: Num a => a -> a -> a
add x y = x + y
```

- Functional programs are mostly side-effect free.

Embedding Languages

Implementing a language within another language.

- Foreign Function Interface (FFI)
  A mechanism by which a program written in one programming language can make use of services written in another language.
  For example, JAVA provides a mechanism to embed code from other languages using the JAVA Native Interface (JNI).
- Library or Module Extension
  Replicate the features and characteristics of the target language into the host.
  For example, the `SAX` library is an XML library extension for JAVA.

Embedding PROLOG

- PROLOG has been embedded in host languages such as JAVA, LISP, SCHEME.
- PROLOG in HASKELL :
  - `Mini Prolog` : A micro PROLOG-like language for an older specification for HASKELL 98.
  - `prolog` : A PROLOG-like interpreter which can be interacted through a REPL.
- Logic programming in HASKELL.
  - Propositional logic,
  - Backtracking, and,
  - Unification.

Merging Paradigms

- Merging paradigms means combining different programming paradigms or programming styles into a single environment resulting in a hybrid language.
- The idea of a multi paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms.
- For example, SCALA is an object functional programming language.
- Functional Logic programming languages include:
  - CURRY,
  - MERCURY,
  - others (see Chapter 6).

Choosing a Language

General Purpose Language

Broad scope but problem needs to be moulded according to the capability of the language.

Special Purpose Language

Limited scope but easier to express the problem as the suitable capabilities are readily available.

- For example, "Clarissa", a voice user interface for browsing procedures on the International Space Station.

Programmer's Dilemma

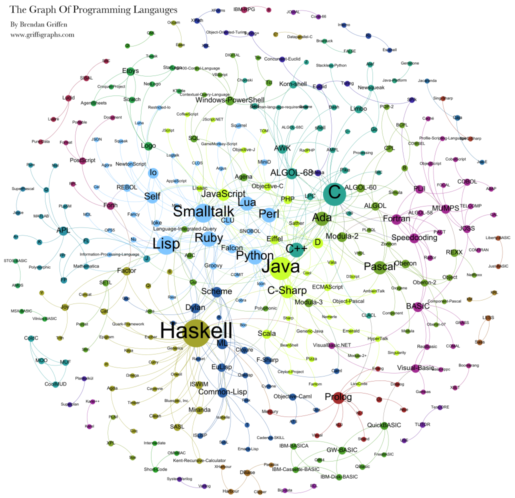

Figure: The Graph of programming Languages

Thesis Statement

Thesis statement

The thesis aims to provide insights into merging two declarative languages namely, HASKELL and PROLOG by embedding the latter into the former and analyzing the result of doing so. . . .

- Replicating functionality.
- PROLOG-like functionality in HASKELL.
- *haskellised* PROLOG-like eDSL

Monads

- Monads in HASKELL can be thought of as composable computation descriptions. They,
  - separate composition and computation execution time line,
  - carry and pass around data(state).
- A monad is a structure that represents computations defined as sequences of steps.
- Monadic computations take place in a “bubble.”

Pattern Matching

- Pattern matching is a feature of HASKELL (and other similar languages).
- In pattern matching, we attempt to match values against patterns and, if so desired, bind variables to successful matches.
- Consider the example of the Fibonacci series in HASKELL.

```haskell
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Unification

- The way in which PROLOG matches two terms is called unification.
  - If term1 and term2 are constants, then term1 and term2 unify if and only if they are the same atom, or the same number.
  - If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 unify, and term2 is instantiated to term1 . (So if they are both variables, theyre both instantiated to each other, and we say that they share values.)
  - If term1 and term2 are complex terms, then they unify if and only if: They have the same functor and arity, and all their corresponding arguments unify, and the variable instantiations are compatible.
  - Two terms unify if and only if it follows from the previous three clauses that they unify.

Unification Examples

- Unifying variables,

```
(X,2) = (1,Y).
X = 1.
Y = 2.
```

- Unifying complex terms,

```
k(s(g),Y) = k(X,t(k)).
X = s(g)
Y = t(k)
```

- Unification is not always easy...

```
h(f(W),S,g(U,y)) = h(U,V,g(f(x),V))
```

...

Unification Examples

- Unifying variables,

```
(X,2) = (1,Y).
X = 1.
Y = 2.
```

- Unifying complex terms,

```
k(s(g),Y) = k(X,t(k)).
X = s(g)
Y = t(k)
```

- Unification is not always easy. . .

```
h(f(W),S,g(U,y)) = h(U,V,g(f(x),V))
W=x
U=f(x)
S=V=y
```

Previously Existing Work

- Implementations.
  - `Mini Prolog` : PROLOG-interpreter with support for variable search strategies.
  - `prolog` : A PROLOG interpreter written in HASKELL.
  - CURRY : A functional logic programming based on HASKELL
- Literature.
  - Translating PROLOG predicates into a HASKELL function.
  - Passing state of a computation.
  - Implementing a typed functional logic programming language.
- Libraries.
  - `unification-fd` : Generic unification algorithms.
  - `logict` : A continuation-based, backtracking, logic programming monad.

Overview of what we did

- Literature review
  - We reviewed literature on embedded languages (Chapter 5)
  - We reviewed literature on merging programming languages (Chapter 6)
- Improvements,
  - Practical features.
  - PROLOG in HASKELL.
- Other Contributions,
  - Modular *functorizing* mechanism.
  - Modular *monadic* unification procedure.
  - Basic working PROLOG implementation.
  - Variable search strategies.
  - Embedding IO in an eDSL.

Functorizing a language

- Traditional abstract grammars are recursive and closed.
- Open languages let other tools add, for instance, unification variables.
- Closed language:

```
data Term = Struct Atom [Term]
          | Var VariableName
          | Wildcard
          | Cut Int deriving (Eq, Data, Typeable)
```

- Open language:

```
data FlatTerm a = Struct Atom [a]
                |  Var VariableName
                |  Wildcard
                |  Cut Int deriving (Show, Eq, Ord)
```

Functorizing a language

- Manual extension,

```
data Term = Struct Atom [Term]
          | Var VariableName
          | Wildcard
          | Cut Int
          | New_Constructor_1 .........
          | New_Constructor_2 .........
    deriving (Eq, Data, Typeable, .....)
```

- Functorized Extension,

```
data Extended f = New_Constructor_1 ...
                | New_Constructor_2 ....
                | Base (f (Extended f))
    deriving (Eq, Data, Typeable, .....)
```

Monadic Unification

- `unification-fd` compatible
- *Unifiable* language.
- Convert the language expressions,
- Extract and convert variables to become `State` compatible,
- Carry out unification in the `Binding Monad`.
- Re translate substitutions.

Prototype 1

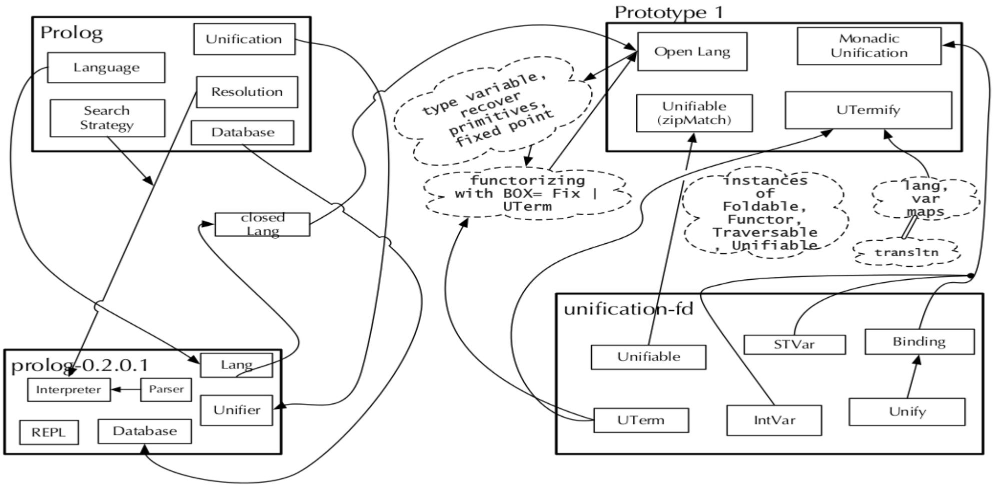

Figure: Architecture of Prototype 1

Contributions

- A query resolver matches a query to the rules in the knowledge base and generate a list of goals which when achieved a result is returned.
- A query resolver consists of:
  - a scheduling policy,
  - a search strategy,
  - unification.
- Prototype 1 is only unification.
- Prototype 2 is a PROLOG-like interpreter.

Prototype 2

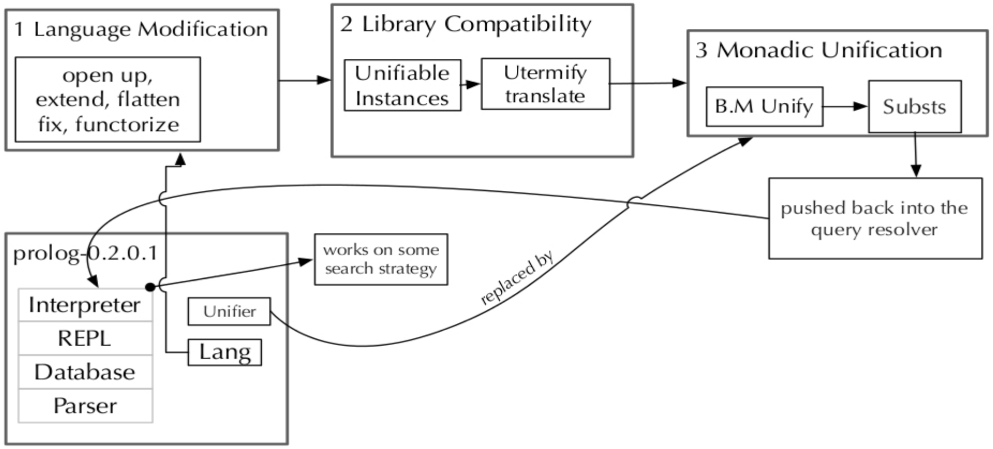

Figure: Architecture of Prototype 2

Contributions

- Variable search strategy:
    - Stack Engine,
    - Pure Engine, and,
    - Andorra Engine.

Prototype 3

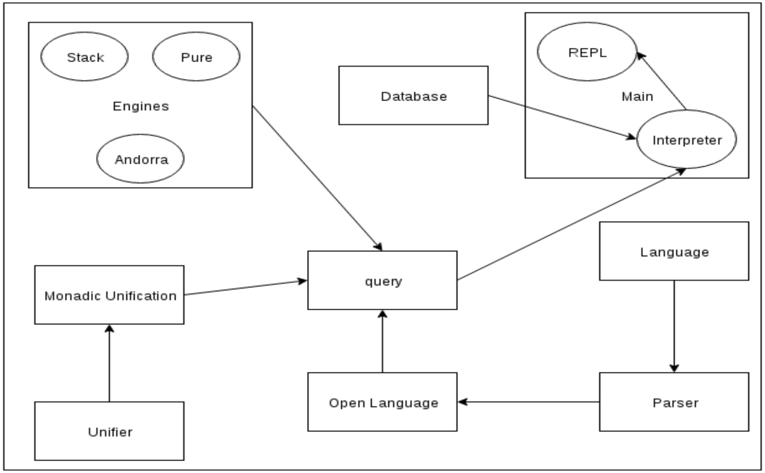

Figure: Architecture of Prototype 3

Contributions

- Embedding input-output capabilities to a domain specific language.
- Constructors for IO operations in the abstract syntax.
- Interpreted program is pure.
- Two-stage interpretation strategy.

Prototype 4

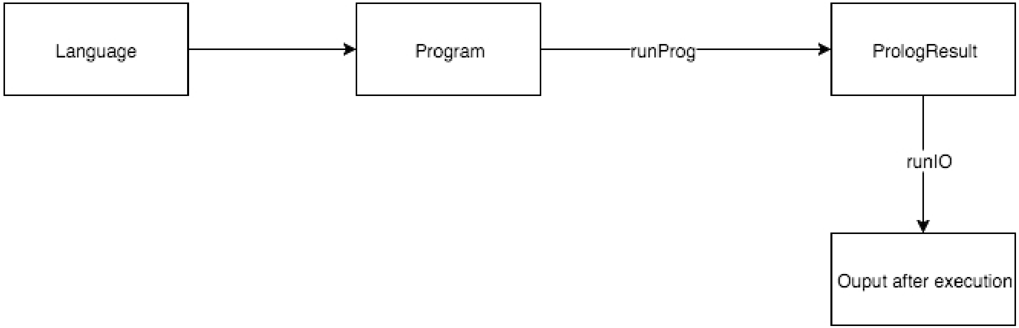

Figure: Architecture of Prototype 4

Future Scope

- Quasi quoter with anti-quotation.
- Variable run time search strategy.
- Additional database capabilities similar to SWI PROLOG.
  - `assert/retract` database manipulation.
  - recorded database.
- Multi type variable language.
- Prototype 4 additions and extension.

Conclusion

In conclusion,

- we explored the various approaches for bringing languages together.
- We looked at the state and support for embedded domain specific languages in HASKELL.
- Specifically, we looked at PROLOG in HASKELL.
- Our ideas were tested by the means of some prototypes to find out that HASKELL is an effective tool for embedding.
- Lastly, we contributed towards solving the programmer's dilemma of choosing programming languages.

The End

Thank you!

The End

Questions?

The End