Christoph Kohlhepp / Int.J.Data.Sci. & Big Data Anal. 2(2) (2022) 18-32
https://doi.org/10.51483/IJDSBDA.2.2.2022.18-32 ISSN: 2710-2599

SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Data Science and Big Data Analytics

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper **Open Access**

Bridging Discrete and Continuous Logicin Automated Reasoning Systems

Christoph Kohlhepp[1*]

[1]Independent. E-mail: chris.kohlhepp@gmail.com

Article Info

Volume 2, Issue 2, November 2022
Received : 22 August 2022
Accepted : 18 October 2022
Published : 05 November 2022
doi: 10.51483/IJDSBDA.2.2.2022.18-32

Abstract

In this paper we will explore the connection between Functional Programming and reasoning. Functional Programmingis often advanced as a means to increase type safety, write correct programs, or simply to pursue brevity and succinctness in program design. Much less emphasized, but just as powerful, is the connection between functional programming and logic—between functional programming and automated reasoning. On our journey we will consider object orientation, relational databases, and a Lisp based system called Powerloom, a Knowledge Representation and Reasoning System that can represent both (A) discrete mathematics as well as; (B) quantitative models such as regression as used instatistics. Powerloom is being developed at the Intelligent Systems Division at the University of Southern Californiaand was funded by Defence Advanced Projects Agency (DARPA). Along the way, we will look at a system called Coq,a theorem prover for OCaml. This is part 2 of a 3 part series. Part 1 is the Anatomy of a Puzzle.

Keywords: *Functional programming, Reasoning, Automated reasoning, Regression, Powerloom*

1. Introduction

Programming languages define the concept of a "First Class Citizen." First class citizens of a language are entities which "support all the operations generally available to other entities." An object-oriented language sets itself apart bydefining objects as first class citizens. This means objects can be created, destroyed, passed as arguments to functions, stored in collections, manipulated etc. For a concept to be a first class citizen means that essentially all of the operations of a language may be applied to that concept. By contrast, objects may be simulated in C and Axel-TobiasSchreiner details how to do this in his book ooc.pdf. But this does not mean objects are first class citizens in C. It means some of the abstractions embodied by objects may be emulated in C. Object Orientation supports precisely two coreabstractions: Classification and Association.

Figure 1 shows examples of classification and association. A dog is classified as an animal. A dog associates with the classes Frisbee and Bone. Virtual methods, a form of dynamic dispatch, are syntactically integrated with classification. Quintessentially every important concept in Object Orientation is bootstrapped on just these two abstractions. This does not mean other abstractions cannot be represented. In C++, we might write the following.

* *Corresponding author: Christoph Kohlhepp, Independent. E-mail: chris.kohlhepp@gmail.com*

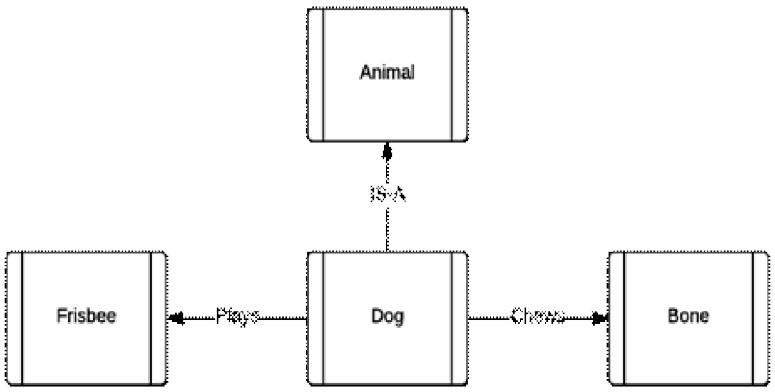
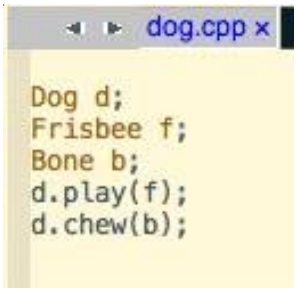


Figure 1: Classification and Association



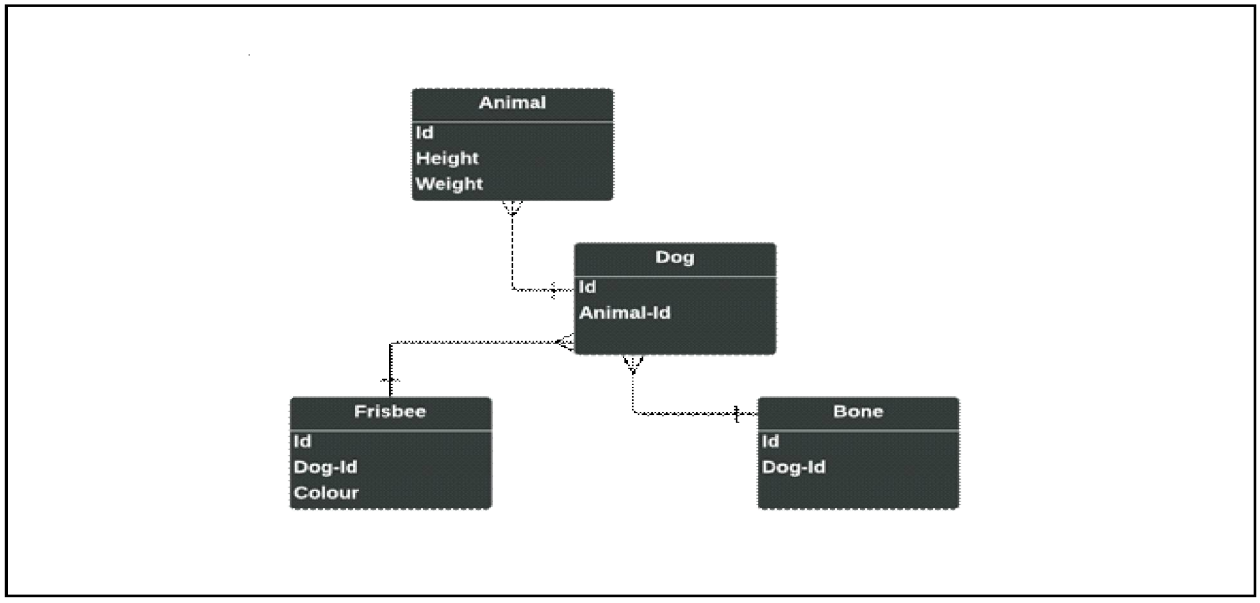
This represents a sequence of code that we have just represented in C++. Three objects are created. Our dog plays with the frisbee and then chews the bone. Yet this sequence is not in its own right a first class citizen of the language. Even ifplaced within a function or method, we cannot manipulate this sequence by sorting it or reversing its order. If I want our"dog d" to chew the bone first, then play with the frisbee, I will have to write different code. To do this correctly, Ineed to know the specifics of the domain - here dogs, frisbees and bones. In broader terms this makes me a knowledgeengineerof the domain.

What if we wanted to know what dogs there are in the system? Right now there is only "dog d." We know this because we wrote the code. But the language does not provide a facility to express this. If we wanted to express this, we might create a collection object which forms an association with dogs (a vector or list perhaps) that functions as a registry. We might then enumerate this registry and query what dogs exist in the system. What we will have done is to restate the problem in terms of one of the two abstractions at our disposal, here: association. Everything in writing an object-oriented program comes back to this simple restatement exercise.

2. Relational Databases

A relational database might be better equipped to handle the last problem. The first class citizens of a relational data-base are "relations." In casual terms, these are connections between tables with rows of data. More formally, they are mappings from elements in a source set to elements in a target set. They are distinguished from mathematical functionsin that an element in a source set may map to more than one element in a target set. As an example, a dog may havemore than one frisbee.

Below is the entity relationship diagram for our dog, frisbee, bone scenario. This looks promising. We may now query"select * from Dog" to answer our previous questions as to what dogs exist in our system.



But wait, something has gone missing. We associate elements from one set with elements from another by duplicating information we call "keys," here denoted with the various "Id" fields. And this mechanism is the same, regardless of whether the semantics embodied is classification or association. Once more, a human is needed to facilitate interpretation. This is our knowledge engineer – yet again. If we need to relate Dog to Animal, a suitable join query is needed. The knowledge engineer will need to reason about how to construct this join. We are operating on sets and have to deal with the eventuality that rows in one set may have missing entries in the other. What is the interpretation of this to be? Clearly we need someone to reason about this who is knowledgeable in the "animal / dog" domain.

We may put all that knowledge into stored procedures and views—yet if in the future we add a "Small Dogs" table whichis meant to be a classification of Dog (read inherits from Dog), then those stored procedures and views may all need tobe updated to reflect this. That is because classification cannot be expressed natively. Where in object orientation, wehad two key abstractions, now we have but one.

3. Knowledge

In both the aforementioned instances it was knowledge of both the syntax (how we express things) and the semantics (what these things mean) that allowed us to work around constraints in the expressiveness of the language. What if we introduced knowledge itself as an abstraction into this eclectic mix?

Firstly, what kind of things are knowable? We are fortunate that the study of knowledge and reasoning is an old one. There is the study of Ontology—study of the nature of being. And there is the study of Logic, from ancient Greek "logike"—the study of valid reasoning. Abstract enough?

Let's make that concrete. It rains. That is a proposition. It is true when precipitation at non freezing temperatures occurs. It is false at other times. The last two sentences form a set of rules that comprise a mini ontology. Formally suchrules are often stated in what is called existential quantification and universal quantification. Let's elaborate our small model. If it rains, Bob carries an umbrella. What is implied is that when ever precipitation does occur, Bob carries his umbrella. That is implication. If I observe that it is raining, I may now apply my implication to infer that Bob must be carrying his umbrella. This rule is known as Modus Ponens. We have also introduced the concept of people into our ontology, here manifested in the persona of Bob. There exist entire calculi around the notion of propositions. We may parameterize propositions like one might parameterize a function and this becomes Predicate Calculus.

4. Why Lisp? Why Powerloom? How about Prolog?

Lisp has an interesting property. It is said that everything in Lisp is a list. I'd rather prefer to think of it as everything in Lisp is a function—including the syntax. It's "turtles all the way down." As an example, where

an arithmetic representation of a function might be f(x), the Lisp representation merely lifts the opening parenthesis in front of the operator f.

```
; Arithmetic:
;
; y = f(x)
;
; Lisp representation of f(x) becomes
(f x)
```

Hence an "if statement" is also a function. It's true and false clauses are functions. A function containg an if statement istherefore a function containing a function that contains functions. This is utterly compositional. Like in our relational example, we have one abstraction to work with. Then we had to create semantics with a restricted syntax. Now it is almost as if instead of a language and a syntax we have a building material for language and syntax.

```
CL-USER> (if *condition*
             (do-something-true)
             (do-something-false))
NIL
CL-USER>
```

A language building material will be welcome when we try to combine concepts as diverse as object orientation, relations, propositions, mixing discrete mathematics with quantitative logic. Syntax allows the construction of semantics. We need a vehicle that offers us the freedom to merge the idioms from these paradigms rather than imposing on us anygiven paradigm as we might expect from more "opinionated" languages such as Haskell.

As for Powerloom, the Powerloom system can be used as a deductive reasoning layer that overlays Lisp. This means Powerloom reasoning can be integrated into an application without a Foreign Function Interface (FFI). To this end, Powerloom embeds a domain specific language, Stella, that looks and feels like Lisp but facilitates Knowledge Interchange Format (KIF). Prolog, by contrast, tends to be a largely standalone eco system.

Powerloom has backend implementations in Java, C++ and Lisp. Powerloom has been ported to the iPhone.

5. Let's Play

Before we do, let us consider what kind of abstraction we will be working with. We already discussed propositions as abuilding block of knowledge. So it is likely we will be seeing a new data type: the proposition. Propositions will need tobe able to capture all the other abstractions we have been discussing. To reason about knowledge we will need logic. So we are likely to see rules formulating constraints upon propositions. Quantification and implication are but a few.

6. Preparation

We load the Powerloom system into Lisp.

```
; SLIME 2014-08-01
CL-USER> (powerloom)

*** This Stella image uses CL-structs instead of CLOS instances ***

STELLA 3.5.29 loaded.
Type `(in-package "STELLA")' to execute STELLA commands.Making required system logic
Making required system powerloom-extensions
PowerLoom 4.0.9.beta loaded.
Type `(powerloom)' to get started.
Type `(in-package "STELLA")' to run PowerLoom commands directly
   from the Lisp top level.
T
CL-USER> (in-package "STELLA")
#<COMMON-LISP:PACKAGE "STELLA">
STELLA> (in-dialect :KIF)
:KIF
STELLA>
```

We will modularize our system. Let's call it the "petting zoo."

```
STELLA> (defmodule "/PL-KERNEL/PL-USER/PETTINGZOO"
           :documentation "Module for a petting zoo."
           :uses ("PL-KERNEL" "LOGIC" "STELLA" "UNIT-KB"))
:VOID
STELLA> (in-module "PETTINGZOO")
|MDL|/PL-KERNEL-KB/PL-USER/PETTINGZOO
STELLA> (clear-module "PETTINGZOO")
:VOID
STELLA> (reset-features)
|l|(:EMIT-THINKING-DOTS :JUST-IN-TIME-INFERENCE)
STELLA>
```

This defines the module for our petting zoo and "constructs" it in a default state.

7. Mindset

Going forward we have propositions, logic rules and a constraints solver. Let's start with an example. Say we have anequation $y = \text{sqrt}(x)$. y is the square root of x. Lets impose a constraint $y = 4$. What x will satisfy this constraint? To find out we create a proposition that says there exist an equality between 4 and the square root of x. We may now ask the constraint solver to retrieve all valid x that satisfy the constraints of this proposition. Let's do just that.

```
STELLA> (retrieve all ?x  (= 4 (sqrt ?x)))
There is 1 solution:
#1: ?X=16
```

What we said here is: retrieve all solutions for the variable *x* where the proposition holds true that 4 equals the square root of *x*. Predictably there is but solution: 16.

8. The Petting Zoo Ontology

Our model is called an ontology (from Greek, the study of being). Let's put some animals in.

Here is our first attempt at definitions of concepts in our zoo ontology.

```
STELLA> (defconcept Animal  (?a)
            :documentation "The class of animal beings")
|c|ANIMAL
STELLA> (defconcept Dog  (?d Animal)
            :documentation "The class of dogs of type animal")
|c|DOG
STELLA> (defconcept Toy (?t)
            :documentation "The class of toys")
|c|TOY
STELLA> (defconcept Frisbee  (?f Toy)
            :documentation "The class of toy frisbees")
|c|FRISBEE
STELLA> (defconcept Bone  (?f Toy)
            :documentation "The class of toy bones")
|c|BONE
STELLA>
```

We have two base concepts, Animal and Toy. Dog is a sub classification of Animal. Bone and Frisbee are toys. In propositional terms, we construct instances of concepts using asserts. Like so.

```
STELLA> (assert (Dog fido))
|P|(DOG FIDO)
STELLA> (assert (Dog sparky))
|P|(DOG SPARKY)
```

We can immediately proceed to solve our first problem. What dogs exist in the system?

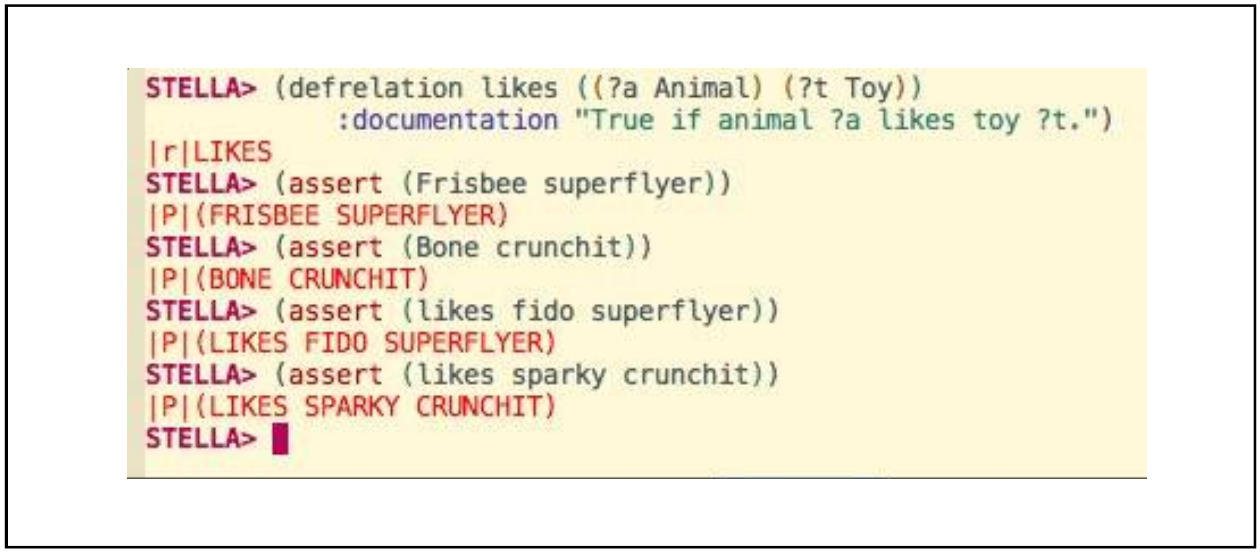
```
STELLA> (ask (Animal fido))
TRUE
STELLA> (ask (Animal sparky))
TRUE
STELLA>
```

This is our first win. Are Sparky and Fido also animals?

```
STELLA> (ask (Animal fido))
TRUE
STELLA> (ask (Animal sparky))
TRUE
STELLA>
```

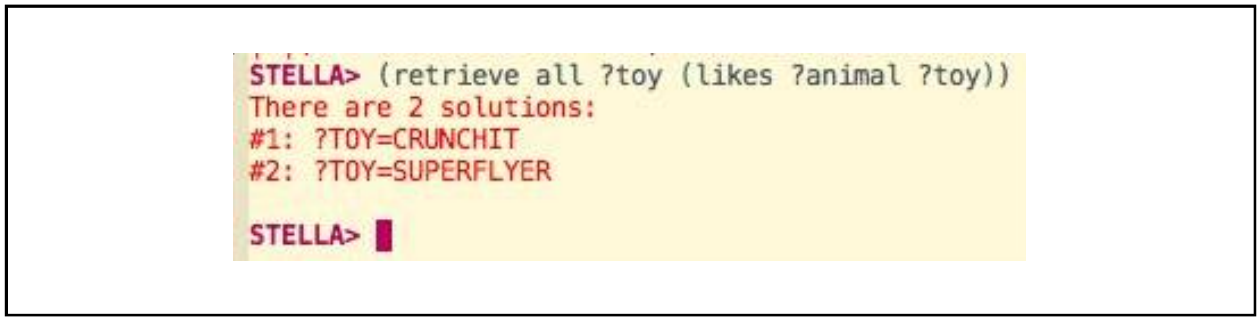
They are. This looks a lot like Prolog. If we can "ask" it, it is a proposition. Let's build some relational abstractions. We begin by defining a relation "likes" for animals and toys, proceed to "construct" some toys

and then defining who likes them. Note that our assertion "fido likes superflyer" type checks in that dogs are animals. In a relational database we would now be juggling keys. As before, what we can "assert" and what we can "ask" must be a proposition. Relations therefore are propositions.



```
STELLA> (defrelation likes ((?a Animal) (?t Toy))
          :documentation "True if animal ?a likes toy ?t.")
|r|LIKES
STELLA> (assert (Frisbee superflyer))
|P|(FRISBEE SUPERFLYER)
STELLA> (assert (Bone crunchit))
|P|(BONE CRUNCHIT)
STELLA> (assert (likes fido superflyer))
|P|(LIKES FIDO SUPERFLYER)
STELLA> (assert (likes sparky crunchit))
|P|(LIKES SPARKY CRUNCHIT)
STELLA>
```

Tell me about every toy that is liked.

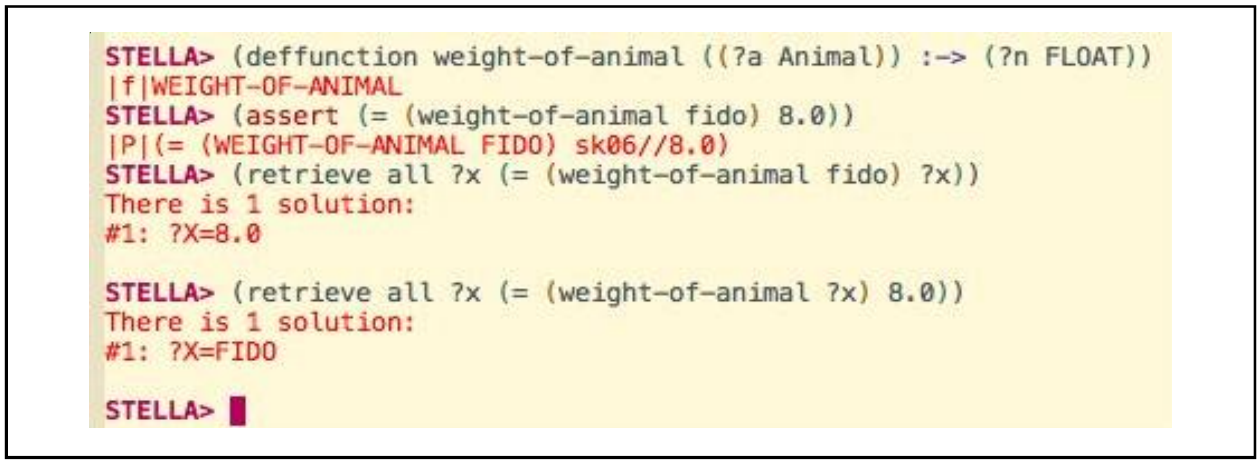


```
STELLA> (retrieve all ?toy (likes ?animal ?toy))
There are 2 solutions:
#1: ?TOY=CRUNCHIT
#2: ?TOY=SUPERFLYER

STELLA>
```

You get the idea, we can perform relational queries as relational database programmers know them. If we can formulate the predicate on the relation, we can query it. Since relations are not restricted to an arity of two, we have covered the territory of SQL joins with out resorting to any additional abstractions. We didn't even need keys to formulate our relational query across "tables." We have also moved into the realm of Predicate Calculus—where propositions have quantifiable variables.

So, we also have classification and sub-classification. And we can query what we assert. But can we perform any usefulfunction? Let's define a function weight-of-animal. We can now set the weight of our animals and retrieve it. As before, our "setter" is assert; our "getter" is retrieve. We can ask questions such "what does Fido weigh?" Or we can ask, who weights 8.0?



```
STELLA> (deffunction weight-of-animal ((?a Animal)) :-> (?n FLOAT))
|f|WEIGHT-OF-ANIMAL
STELLA> (assert (= (weight-of-animal fido) 8.0))
|P|(= (WEIGHT-OF-ANIMAL FIDO) sk06//8.0)
STELLA> (retrieve all ?x (= (weight-of-animal fido) ?x))
There is 1 solution:
#1: ?X=8.0

STELLA> (retrieve all ?x (= (weight-of-animal ?x) 8.0))
There is 1 solution:
#1: ?X=FIDO

STELLA>
```

This does represent a mathematical function in that one element from a source set is mapped to exactly one element of a target set. Yet it still does not represent any useful computation. This merely sets and gets values. It is the analog of a set() and a get() on an object property. Lets make something up. Animals in our petting zoo are well fed. Their weightdoubles every month. So, we formulate a function, weight-in-a-month to be the double of the animals present weight.This in an actual computation such as might be performed in a method of an object-oriented class.

Let us declare such a function.

```
STELLA> (deffunction weight-in-a-month ((?a Animal)) :-> (?n FLOAT))
|f|WEIGHT-IN-A-MONTH
```

But wait, this is only a function signature. What relates weight and weight in a month? A typed, universal quantifica-tionand an implication.

```
STELLA> (assert (forall  (?a Animal)
                (=> (> (weight-of-animal ?a) 0)
                    (= (weight-in-a-month ?a) (* 2 (weight-of-animal ?a))))))
|P|(FORALL (?a)
   (<= (EXISTS (?v16 ?v15)
         (AND (= (WEIGHT-IN-A-MONTH ?a) ?v16) (= (WEIGHT-OF-ANIMAL ?a) ?v15) (= (* 2 ?v15)
 ?v16)))
       (EXISTS (?v13)
         (AND (ANIMAL ?a) (= (WEIGHT-OF-ANIMAL ?a) ?v13) (> ?v13 0)))))
STELLA>
```

What we are saying here is that for all animals with positive weights, the weight in a month will be twice today'sweight.

9. Curry-Howard Correspondence

This is also our first major observation. In the world of propositions, we apply a function by way of a logical implication. We used a typed, universal quantification and a logical implication. What is the universal quantification ? The (forall) statement. What is typed? Forall applies to quantifiable variables?a that satisfy the proposition (Animal?a). The rest is our implication relating the weight of an animal in a month to its current weight with an arithmetic equality constraint. What we are seeing here is a correspondence between function application and logical implication. This is also known as the Curry-Howard Correspondence. The Curry-Howard correspondence creates a correspond-ence between a type system and models of computation. We will see more of the Curry-Howard Correspondence later.

Back on track. What will Fido weigh in a month?

```
STELLA> (retrieve all ?x (= (weight-in-a-month fido) ?x))
There is 1 solution:
#1: ?X=16.0

STELLA>
```

Some advanced programming languages have the notion of dependent types. For example we might have a File type.But if the File is open, it's an OpenFile. This a powerful mechanism for specifying and verifying

invariants. For example, only open files can be closed. If a close operation is attempted on a non open file, the compiler or system oughtto know what to do. Why leave to a programmer, what a specification can handle? Can we do this here? We have already seen that types are introduced as concepts. But so far our concepts were very simple and only specify direct classifications.

We are really worried about our animals doubling in weight every month. Our nutritionist has asked us to track "lightanimals."

```
STELLA> (defconcept LightAnimal ((?a Animal))
          :<=> (< (weight-of-animal ?a) 10.0))

|c|LIGHTANIMAL
```

This is our first dependent type. With types as propositions, we can quantify our type using a predicate. Is Fido a lightanimal?

```
STELLA> (ask (Lightanimal fido))
TRUE
STELLA>
```

Thank goodness, he is. But do we know why? Here comes a really cool feature. I always wanted to ask my computer"why?"

What follows is mathematical proof as to why Fido is classed as "light" according to the logic of our ontology.

```
STELLA> (ask (Lightanimal fido))
TRUE
STELLA> (why)
1 (LIGHTANIMAL FIDO)
    follows by Modus Ponens
    with substitution {?v13/8.0, ?a/FIDO}
    since 1.1 ! (FORALL (?a)
                  (<= (LIGHTANIMAL ?a)
                      (EXISTS (?v13)
                        (AND (= (WEIGHT-OF-ANIMAL ?a) ?v13)
                             (< ?v13 10.0)))))
    and   1.2 ! (= (WEIGHT-OF-ANIMAL FIDO) 8.0)
    and   1.3   (< 8.0 10.0)

1.3 (< 8.0 10.0)
    follows because it was proven by an inference specialist

|kv|(<|i|@PRIMITIVE-STRATEGY,|i|@EXPLANATION-INFO> <|i|@PRIMITIVE-STRATEGY,|i|@EXPLANATION-
INFO> <|i|@PRIMITIVE-STRATEGY,|i|@EXPLANATION-INFO> <|i|@JUSTIFICATION,|i|@EXPLANATION-INFO
>)
STELLA>
```

How about Sparky? Is Sparky a lightanimal? What do we expect here?

```
STELLA> (ask (LightAnimal sparky))
UNKNOWN
```

It is *NOT* false. That would not be logical as we have not said anything about Sparky's weight. This is also calledan Open World Assumption.

10. Relational and Object Abstractions Under One Roof

How far will all this stretch in reasoning about the model of our world? We have seen relations. We have seen classification. We have seen functions. We have seen propositions and predicates glueing these together. This is what we set outto do. We have already covered the basics of object orientation and relational databases in one paradigm. Indeed pro-positions appear to be a perfect vehicle to solve the Object Relational Impedance Mismatch. Java Hibernate solves this—with 1,166,039 lines of code.

11. Transitivity and Recursion

Firstly we define parentage and ancestry.

```
STELLA> (defrelation has-parent ((?p Animal) (?parent Animal))
          :documentation "True if `p' has `parent' as a parent.")
|r|HAS-PARENT
STELLA> (defrelation has-ancestor ((?p Animal) (?ancestor Animal))
          :documentation "True if `p' has `ancestor' as an ancestor.")
|r|HAS-ANCESTOR
```

Then we define rules governing these. Our first rule says that all parents are ancestors.

```
STELLA> (assert (forall ((?x Animal) (?y Animal))
          (=> (has-parent ?x ?y)
              (has-ancestor ?x ?y))))
|P|(FORALL (?x ?y)
   (<= (HAS-ANCESTOR ?x ?y)
       (AND (ANIMAL ?x) (ANIMAL ?y) (HAS-PARENT ?x ?y))))
STELLA> (assert (forall ((?x Animal) (?z Animal))
          (=> (exists (?y Animal)
                (and (has-ancestor ?x ?y)
                     (has-ancestor ?y ?z)))
              (has-ancestor ?x ?z))))
|P|(FORALL (?x ?z)
   (<= (HAS-ANCESTOR ?x ?z)
       (EXISTS (?y)
         (AND (ANIMAL ?x) (ANIMAL ?z) (ANIMAL ?y) (HAS-ANCESTOR ?x ?y) (HAS-ANCESTOR ?y ?z
)))))
STELLA>
```

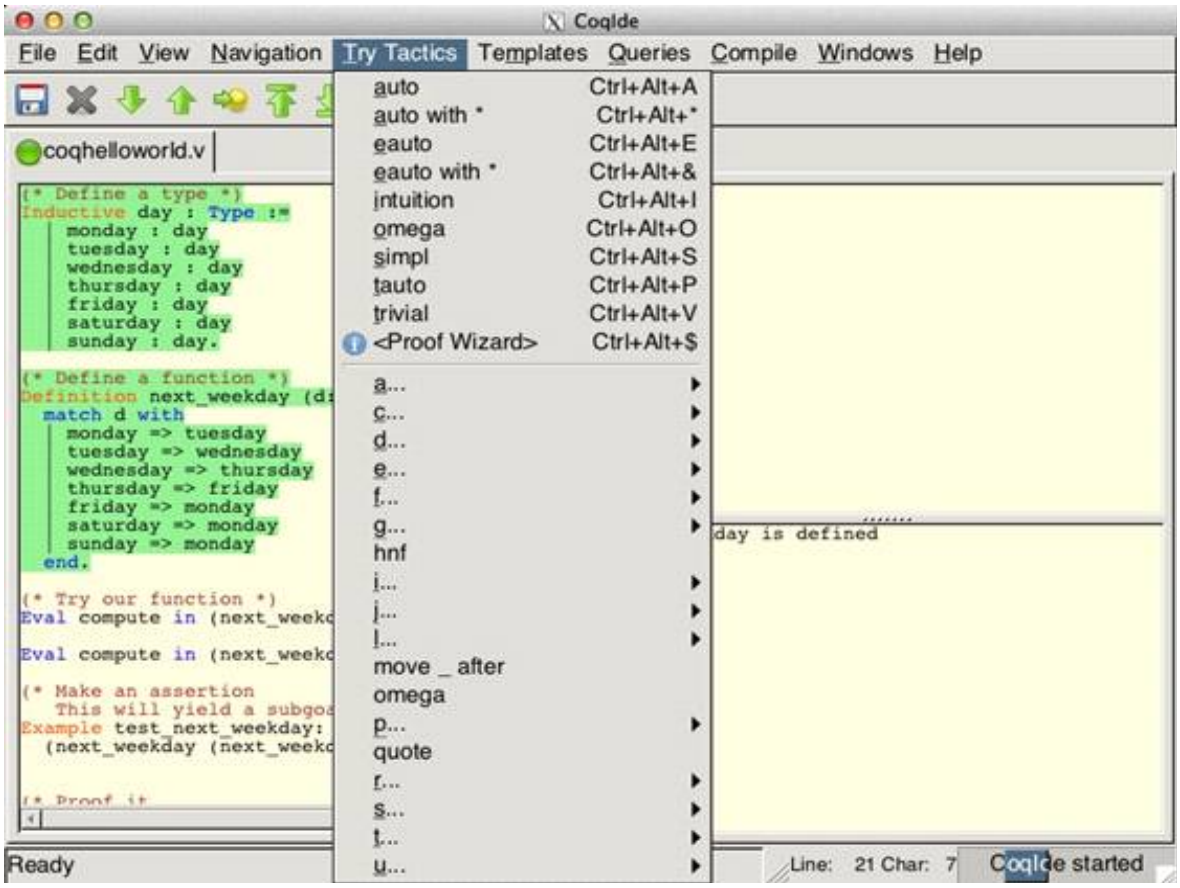
Our second rule is more interesting. It says that for all ?x and ?z which are Animals where there exists an Animal ?y such that ?x is an ancestor of ?y and ?y is an ancestor of ?z then ?x is also an ancestor of ?z.

This demonstrates two things. (1) It makes ancestry transitive. (2) It corresponds to a recursive relationship. It creates acorrespondence between a type system and a model of computation. We have seen this before. It was the Curry-Howard Correspondence. There is more to this than we have covered. For example, the chain rule of inference correspondsto function composition. This is highly intuitive if we consider that we have used implication to formulate function ap-plication and that Modus Ponens govers inference from implication.

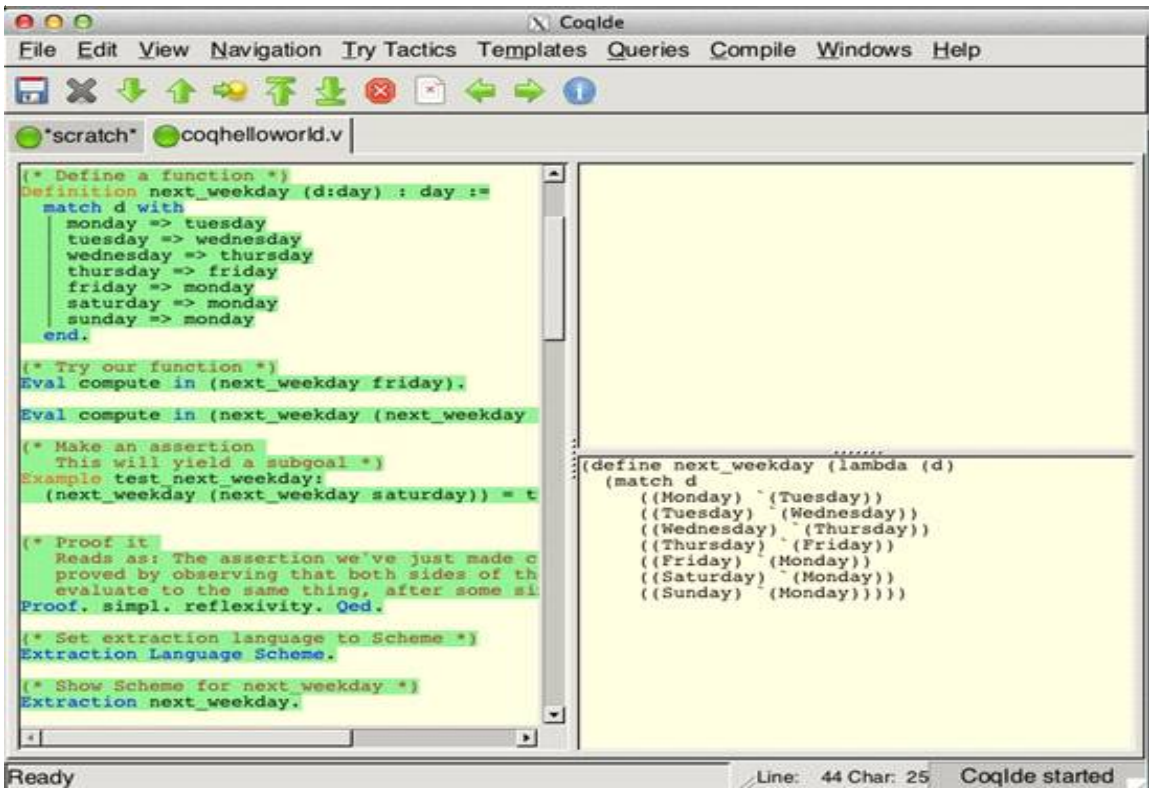
What then can we do here? We might formulate a statement such as all ancestors are worried about descendants whose body weight will be in excess of a certain limit in a month. Firstly we will need a worried-about relation. That exercise is left to the reader, but the point is this: With ancestry already embodying the recursion, we will not have to write recursive logic again when we write logic relating to ancestry. Differently put, we don't have to destructure our data when we handle it. This is profound. As an example, if I define Fido to have a parent and the parent to have a parent who will then be Fido's ancestor by implication—if I now want to know if that ancestor is worried about Fido's body weight in a month, I can just "say so." (ask ...) Likewise, if I want to know what weight that ancestor is worried about, e.g., compute the number please, I can just (retrieve 1 ?x ...) What I don't have to do is to find all of Fido's ancestors by recursively traversing some data structure. We can create lists and trees using relations and propositions. If we have listand trees and we have function application and recursion, we have all of the key concepts of functional programming. We might state this simply as:

12. Functional Programming is Logical: Coq

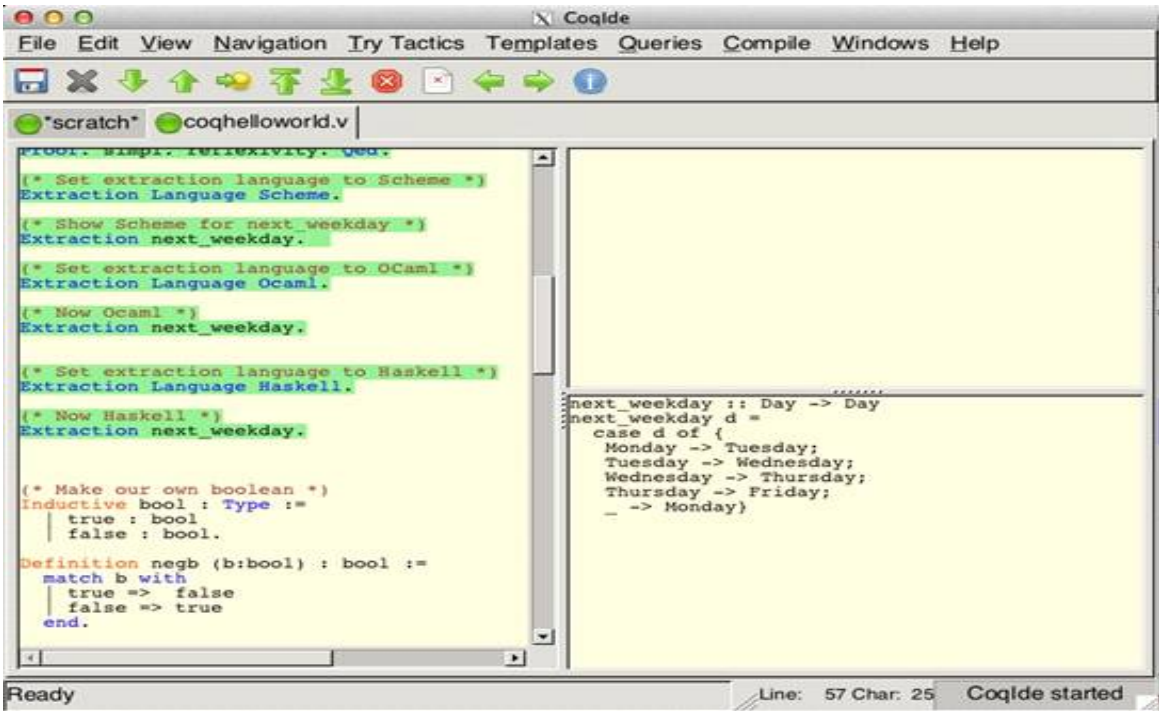
We have already seen an example of mathematical proof. Yet neither Powerloom nor its hosted Lisp dialect Stella a respecifically aimed at mathematical proof. There are other systems which leverage the Curry-Howard correspondence to assist with mathematical proofs. One such system is Coq from the French National Research Institute for Computer Science and Automation, INRIA. Coq hosts a language called Gallina in OCaml, much like Powerloon hosts Stella withinLisp. Coq additionally provides proof tactics. You might think of tactics like assisting the knowledge engineer with propositions as design patterns assist with objected oriented programming. Yet, this is not primarily an article on Coq. It is mentioned here to emphasize that the paradigm of reasoning with functional programming as a vehicle for logic truly does scale.



One particular honorable mention is that Gallina specifications, once proven, may be extracted to Scheme, OCaml and Haskell.



The above demonstrates extraction to Scheme. The below demonstrates extraction to Haskell.



13. Our Coup D'Etat

13.1. Marriage Between Discrete and Continuous Logic

Propositional logic and the entire school of discrete mathematics have a problem. Only certain aspects of reality can be described in terms of discrete logic. The weather is good. The weather is bad. These are propositions. They are qualitative. But actually there is a 70% chance of rain today. This metric is quantitative. So is that good weather or bad? In the real world decisions often have to be made not on discrete data but on data that is inherently continuous. Systems of reasoning that do not cater for this are likely to be not widely applicable. Actually, it's worse than it may at first appear. Often discrete decisions must be made based on continuous data. For example a store may have to choose to stock its shelf space with the most popular brands of a product for the next season based on a forecast of consumer sentiment which in turn is based on market statistics from the previous season. In data science there is both qualitative and quantitative data. We cannot model the world with just one of these.

Let's explore this. Consider a population of people with varying ages. People in a population have relationships, friends,spouses, parent, children—all propositions. Ages have been recorded for people. Powerloom offers regression by wayof a built in neural network. Let's start Powerloom's neural network and train it on the age of people.

```
STELLA> (set-error-print-cycle 1)
1
STELLA> (set-partial-match-mode :nn)
(set-neural-network-training-method :BACKPROP)
(set-learning-rate 0.2)
:NN
STELLA> :BACKPROP
STELLA> 0.2
STELLA> (set-momentum-term 0.8)
0.8
STELLA> (structured-neural-network-regression person age 50)
Generating training examples
Building classification rule with 83 clauses and 18 variables
Training regression networks
Training Networks
Saving networks
Cycle 0 Error: 0.2770387946700297
Cycle 1 Error: 0.26884167381087826
Cycle 2 Error: 0.2478956588673619
Cycle 3 Error: 0.19807569462478763
Cycle 4 Error: 0.1574161373523358
Cycle 5 Error: 0.12509694062010368
Cycle 6 Error: 0.11032486663646277
```

Note that there is a database of samples which our neural network has used to build information about the ages of people. What is remarkable is the ease with which were able to spin up a neural network in under 10 lines of code. Our ontology forms the basis of feature selection for the neural network. Feature selection for neural networks is a science of its own and can be the most critical part of using neural networks effectively. More about this later.

```
(assert (Age Don 82.0))
(assert (Age Grady 91.0))
(assert (Age Gwen 54.0))
(assert (Age Gary 57.0))
(assert (Age Mike 57.0))
(assert (Age Dee 50.0))
(assert (Age Steve 52.0))
(assert (Age Donna 53.0))
(assert (Age Francis 81.0))
(assert (Age David 29.0))
(assert (Age Megan 21.0))
(assert (Age Alec 22.0))
(assert (Age Paul 60.0))
(assert (Age Bart 28.0))
(assert (Age Brent 29.0))
(assert (Age Lisa 28.0))
(assert (Age Carol 51.0))
(assert (Age Aileen 81.0))
(assert (Age Allene 83.0))
(assert (Age Bob 56.0))
(assert (Age Tara 28.0))
(assert (Age Susan 29.0))
(assert (Age Bern 30.0))
...
```

Now we are ready to make some predictions. John is a person. What is John's approximate age expected to be?

```
STELLA> (assert (person John))
|P|(PERSON JOHN)
STELLA> (approximate John Age)
73.9558884057023
```

Ok, this must be a developed country in about 20 years time... Without further knowledge about John, he is likely to be73 years old. Let's make him a parent.

```
STELLA> (assert (person Fred))
|P|(PERSON FRED)
STELLA> (assert (parent John Fred))
|P|(PARENT JOHN FRED)
STELLA> (approximate John Age)
79.49011334314483
```

Unsurprisingly, given the high mean age of this population, making John a parent only alters the prediction by a fewyears. Most everybody in this population seems to be a parent—and rather senior. So then... what happens if we make Fred a parent and in doing so make John a grandparent—without let it be noted saying anything further about John?

```
STELLA> (assert (person Mary))
|P|(PERSON MARY)
STELLA> (assert (parent Fred Mary))
|P|(PARENT FRED MARY)
STELLA> (approximate John Age)
84.79457266187947
```

As a grandparent, John may now expect to be 84 years old. What is remarkable is that Powerloom is able to infer this based on implied structural relationships between the people involved.

This is significant in that this allows interactive exploration of feature selection in our neural network. The prompt from the screenshot above, if it is not apparent, is a REPL — think iPython style agile development. We can make and retract assertions, and probe how the outcome of predictions is affected by these assertions. Entire systems are devoted to the relationship between an ontology and predictive quantitative analytics. Lumify is one such system. With Powerloom and Lisp, you get the power of connecting an ontology with predictive quantitative analytics that Lumify brings to thetable—plus everything else we have been looking at.

This is the marriage between continuous and discrete logic that we sought to solve real word problems. What we have combined is qualitative classification and quantitative regression. We might have equally well achieved this through conventional regression in something like R. Here's the BUT... we would have had to perform a separate regression analysis on every classification applicable to the Person class. Classify the relevant Person instance, then apply the associated regression to that person to derive a prediction. The problem is, while parentage is a classification, being agrandparent is not. Nor is being a great-grandparent. Where would you stop? Parentage is simply a transitive relationship to Powerloom. Regression analysis does not really allow modeling transitivity. Additionally we would have to consider the combination of every possible classification with every other classification. Does being classified as a parent AND a friend impact your own predicted age? Few newborns have friends and there is probably a statistical correlation between the age of your friends and your own. So the answer would have to be a "hesitant yes." It is not so much knowing that you are friend that allows me to gauge your age, but knowing the ages of all your friends collectively that will allow me to make a reasonable prediction. To Powerloom this is just another property of the connections in it's knowledge tree. Parent of parent of you is analogous to age of friend of you. The process of inference is the same.

14. What is Next?

In part 1 of this series (Anatomy of a Puzzle) we looked at specification driven programming using a system called Screamer, a nondeterministic choice-point operator, backtracking and a forward propagation system. In this article we looked at the fusion of functional programming in qualitative and quantitative reasoning using Powerloom. Inpart 3 of this series I would like to apply all this to a substantial problem: The 2015 Christmas Puzzle by GCQH, theUnited Kingdom's equivalent of the United States National Security Agency (NSA). This will be an interesting experiment in that it looks to be an NP-hard cryptanalytic task. The goal will be to write a declarative specification of theproblem and have Lisp derive the solution. As with the (Anatomy of a Puzzle), we will endeavor to do this not through primarily imperative code (C++, Java, C#) but by creating an executable specification of the problem that "solves itself."

The title of this article is a tribute to Daniel Friedman's The Reasoned Schemer available from MIT Press here.

Cite this article as: Christoph Kohlhepp (2022). TBridging Discrete and Continuous Logicin Automated Reasoning Systems. *International Journal of Data Science and Big Data Analytics*, 2(2), 18-32. doi: 10.51483/IJDSBDA.2.2.2022.18-32.