FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploring Visual Programming Concepts for Probabilistic Programming Languages

Gabriel Cardoso Candal

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Hugo Sereno Ferreira

July 22, 2016

Exploring Visual Programming Concepts for Probabilistic Programming Languages

Gabriel Cardoso Candal

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2016

Abstract

Probabilistic programming is a way to create systems that help us make decisions in the face of uncertainty. Lots of everyday decisions involve judgment in determining relevant factors that we do not directly observe. Historically, one way to help make decisions under uncertainty has been to use a probabilistic reasoning system. Probabilistic reasoning combines our knowledge of a situation with the laws of probability to determine those unobserved factors that are critical to the decision. Typically, the way the several observations are combined is through the usage of bayesian statistics, due to its anachronistic interpretation where existing knowledge (priors) are combined with observations in order to gather evidence towards competing hypothesis.

When compared to other machine learning methods (such as random forests, neural networks or linear regression), which take homogeneous data as input (requiring the user to separate their domain into different models), probabilistic programming is used to leverage the data's original structure. Plus, it provides full probability distributions over both the predictions and parameters of the model, whereas ML methods can only give the user some level of confidence on the predictions.

Until recently, probabilistic reasoning systems have been limited in scope and have been hard to apply to many real world situations. Models are communicated using a mix of natural language, pseudo code, and mathematical formulae and solved using special purpose, one-off inference methods. Rather than precise specifications suitable for automatic inference, graphical models typically serve as coarse, high-level descriptions, eliding critical aspects such as fine-grained independence, abstraction and recursion.

Probabilistic programming is a new approach that makes probabilistic reasoning systems easier to build and more widely applicable. A probabilistic programming language (PPL) is a programming language designed to describe probabilistic models, in a such a way we can say that the program itself is the model, and then perform inference in those models. PPLs have seen recent interest from the artificial intelligence, programming languages, cognitive science, and natural languages communities. By empowering users with a common dialect in the form of a programming language, rather than requiring each one of them to the non-trivial and error-prone task of writing their own models and hand-tailored inference algorithms for the problem at hand, it encourages exploration, since different models require less time to setup and evaluate, and enables sharing knowledge in the form of best practices, patterns and tools such as optimized compilers or interpreters, debuggers, IDE's, optimizers and profilers.

PPLs are closely related to graphical models and Bayesian networks but are more expressive and flexible. One can quickly realize this by looking at the re-usable components PPLs offer, being one of them the inference engine, which can be plugged in into different models. For instances, it is easy to replace the exact solution traditional Bayesian networks inference, which requires time exponential in the number of variables to run, with approximation algorithms such as the Markov Chain Monte Carlo (MCMC) or Variational Message Passing (VMP), which make it possible to compute large hierarchical models by resorting to sampling and approximation. PPLs often extend from a basic language (i.e., they are embedded in a host language like R, Java or Scala), although some PPLs such as WinBUGS and Stan offer a self-contained language, with no obvious origin in another language.

There have been successful applications of visual programming among several domains, being it education (MIT's Scratch and Microsoft's VPL), general-purpose programming (NoFlo), 3D modeling (Blender) and data science (RapidMiner and Wek Knowledge Flow). The latter, being popular products, have shown that there is added value in providing a graphical representation for working with data. However, as of today, no tool provides a graphical representation for a PPL.

DARPA, the main backer behind PPLs' research, considers one of the main key points of its Probabilistic Programming for Advancing Machine Learning program to make models easier to write (reducing development time, encouraging experimentation and reducing the level of expertise required to develop such models). The use of visual programming is suitable for this kind of objectives, so building upon the enormous flexibility of PPLs and the advantages of probabilistic models, we want to take advantage of the graphical intuition given by data visualization that data scientists are now accustomed to, and attempt to provide model visualization by rethinking how to capture the (usually textual) programmatic formalisms in a graphical manner.

The goal of this dissertation is thus to explore graphical representations of a probabilistic programming language through the usage of node-based programming. The hypothesis under consideration is that graphical representations (not to be confused with bayesian graphical model), are more intuitive and easy to learn that full-blown PPLs.

We intend to validate such hypothesis by ensuring that classical problems solved in the literature by PPLs are also supported by our graphical representation, and then compare the two regarding understandability and error prevention.

“Convictions are more dangerous foes of truth than lies.”

Friedrich Wilhelm Nietzsche

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
CLI	Common Language Infrastructure
CLR	Common Language Runtime
DARPA	Defense Advanced Research Projects Agency
DOM	Document Object Model
DPL	Dataflow Programming Language
EBNF	Extended Backus–Naur Form
ML	Machine Learning
IDE	Integrated Development Environment
JIT	Just in Time
JVM	Java Virtual Machine
PP	Probabilistic Programming
PPAML	Probabilistic Programming for Advancing Machine Learning
PPL	Probabilistic Programming Language
PR	Probabilistic Reasoning
PVL	Purely Visual Language
VDP	Visual Dataflow Programming
VMP	Variational Message Passing
VP	Visual Programming
VPE	Visual Programming Environment
VPL	Visual Programming Language
XML	eXtensible Markup Language

Chapter 1

Introduction

This first chapter aims to provide the reader with an overview of this dissertation. It starts by introducing the context this work is inserted in, identifying the problem which we aim to solve, how we plan on solving it, and the expected outcome. Lastly, it gives a bird's eye view of this report's structure.

1.1 Context

There is, among several domains with interesting and relevant problems to solve (computer vision [KT15], cryptography, biology, fraud detection, recommender systems [Alp10], ...), the recurring necessity to be able to make decisions in the face of uncertainty using machine learning (ML) methods.

Successful ML applications include Google's personalized advertising and context-driven information retrieval, Facebook's studies of how information spreads across a network or UC Berkeley's AMPLab contributions towards Amazon Web Services and SAP's products [BAF$^+$15].

Typically, there are two alternatives for solving this class of problems: either use an existing machine learning model (such as KNN, neural networks or similar) [Sch08] and try to fit your data into the model, or build a probabilistic model for your own particular problem so you can better leverage domain knowledge [Gri13].

In the second alternative one common way to approach it is by using bayesian reasoning, where you model unknown causes with random variables, feed the model the data you have gathered and then perform inference to query for the desired, and previously unknown, variables [Dow12]. The problem in choosing this method is this last step since it is non-trivial to write an inference method [DL].

The solution to this has been building generic inference engines for graphical models so that modeling and inference can be treated as separate concerns and scientists can focus on the modeling [JW96]. However, not all models can be represented as graphical models, and that's why we now have Probabilistic Programming Languages (PPLs), which are able to represent a larger class of problems [DL13]. Probabilistic Programs let you write your model as a program and have off-the-shelf inference [Pre03].

1.2 Problem

In spite of these examples of applications in the industry, ML has been identified by Gartner, in its Hype Cycle annual review, to be in the "Peak of Inflated Expectations" stage, still far from the "Plateau of productivity" [Sta15].

It has also been said that ML's applications are rarely seen outside the academia, with Wagstaff claiming that there is a "frequent lack of connection between machine learning research and the larger world of scientific inquiry and humanity" [Wag12].

Arguably the scenario is even worse for PPLs, having even less adoption among tech companies than other ML methods, such as neural networks or other supervised learning models. One factor which may be contributing to this lack of usage, despite PP's power and flexibility, is the difficulty for data scientists to adapt to the textual interface these languages provide, which lack the graphical intuition provided by other tools they are accustomed to. In a poll made to find out the most popular data-mining tools among data scientists (see Figure 1.1), half of the top 10 tools are graphically interactive.

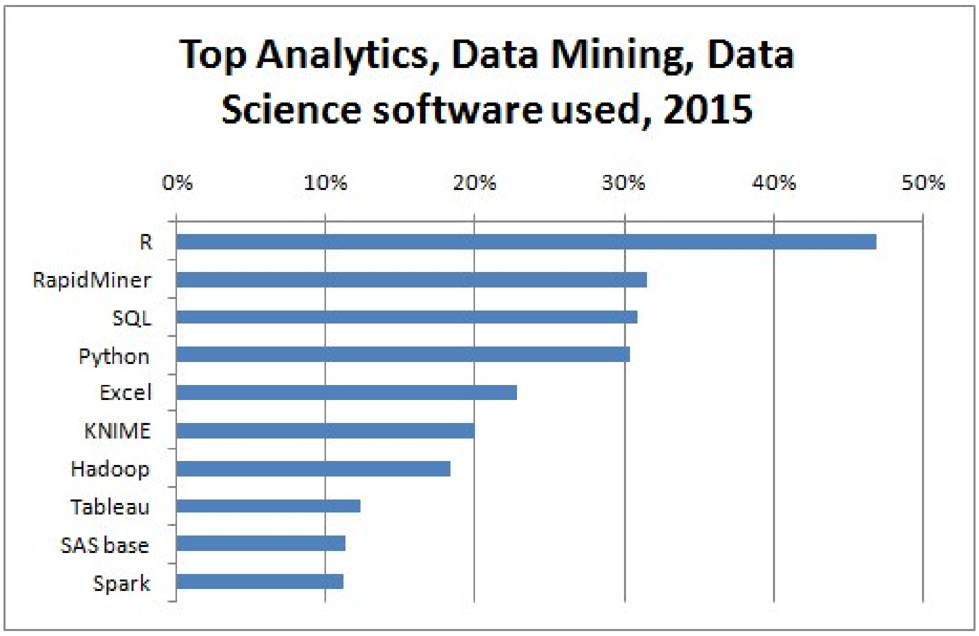


Figure 1.1: Top 10 Analytics Tools [Pia15]

1.3 Motivation and Goals

The Defense Advanced Research Projects Agency (DARPA), one of the backers behind PPLs' research, has recognized some of the problems identified by Wagstaff and started a program called Probabilistic Programming for Advancing Machine Learning (PPAML) to address the shortcomings of current ML methods [Jag13]. It identifies five strategic goals:

- Shorten machine learning model code to make models faster to write and easier to understand
- Reduce development time and cost to encourage experimentation
- Facilitate the construction of more sophisticated models that incorporate rich domain knowledge and separate queries from underlying code
- Reduce the level of expertise necessary to build machine learning applications
- Support the construction of integrated models across a wide variety of domains and tool types

The purpose of this work is to try addressing the first four. In order to do so, we aim to overcome the difficulties in learning a new language, either for inexperienced developers or seasoned ones, such as learning yet another syntax or getting accustomed to the language's idioms. It is known that typical languages are difficult to learn and use [**?**] and that there are advantages in providing a language with a visual interface [AA92]. Also, studies have shown that programmers and data scientists alike resort to mental imagery when solving problems [Das02][PB99], so by providing a visual interface for a programming language we can tighten the gap between how users think and how they express themselves in the language they're using to solve a given problem.

So, the goal of this dissertation will be to develop a Visual Programming Language (VPL) with probabilistic programming capabilities. The targeted audience is programmers and data scientists with background knowledge in statistics which aren't still comfortable with full blown PPLs, but wish to educate themselves on the topic so they can eventually leverage the power of this novel machine learning approach.

The way to do so would be developing a graphical editor, similar to RapidMiner or Blender Composite Nodes, that runs in the browser. The given editor would have the capability to compile the underlying representation beneath the graphical interface (which could be a graph, a tree, or any other data structure we believe to be fit) to the textual representation in the target PPL so that the user can run what he has designed, either directly from the editor itself, as a standalone script or even by integrating it with his existing projects.

The hypothesis under consideration is this graphical representation is more intuitive and easy to learn that a full-blown PPL. We intend to validate such hypothesis by ensuring that classical problems solved in the literature by PPLs are also supported by our graphical representation, and then compare the two regarding understandability and error prevention.

Chapter 2

Background & State of the Art

This chapter has two purposes: describing the foundations on which this work is built on, namely Machine Learning (ML), Probabilistic Reasoning (PR), Probabilistic Programming (PP) and Visual Programming(VP) while enumerating different tools which are based on one or several of these concepts.

2.1 Machine learning

Machine learning (ML) is a field which can be seen as a subfield of artificial intelligence that incorporates mathematics and statistics and is concerned with conceiving algorithms that learn autonomously, that is, without human intervention [Bri16][Sch08]. It has the potential to impact a wide spectrum of different areas such as biology, medicine, finance, astronomy [Ama13], computer vision, sales forecast, robotics [Alp10], product recommendations, fraud detection or internet ads bidding [Gri13].

Learning from data is commercially and scientifically important. ML consists of methods that automatically extract interesting knowledge in databases of sometimes chaotic and redundant information. ML is a data-based knowledge-discovering process that has the potential not only to analyze events in retrospect but also to predict future events or important alterations [Geo16].

2.2 Probabilistic Reasoning

Probabilistic reasoning (PR) is the formation of probability judgments and of informed yet subjective beliefs about the likelihoods of outcomes and the frequencies of events [Las12], it is a way to combine our knowledge of a situation with the laws of probability. There are subjective beliefs because in non-trivial decision-making there is a combination of unobserved factors that are critical to the decision and several sources of uncertainty [?], such as:

- Uncertain inputs, due to missing or noisy data.

- Uncertain knowledge, where multiple causes lead to multiple effects, or there is an incomplete knowledge of conditions, effects and causality of the domain or simply because the effects are inherently stochastic.

So, probabilistic reasoning only gives probabilistic results, and it is one way to overcome cognitive bias and be able to make rational decisions [**?**].

A trial has been made [**?**] where physicians were asked to estimate the probability that a woman with a positive mammogram actually has breast cancer, given a base rate of 1% for breast cancer, a hit rate of about 80%, and a false-alarm rate of about 10%. It reported that 95 of 100 physicians estimated the probability that she actually has breast cancer to be between 70% and 80%, whereas Bayes's rule gives a value of about 7.5%. Such systematic deviations from Bayesian reasoning have been called "cognitive illusions". We will describe both Bayes' rule and Bayesian reasoning in the next section.

2.2.1 Bayesian Reasoning

One way to approach PR is by using bayesian reasoning, which is inspired by the Bayes Theorem (or Rule, or Law). An equivalent formula to the theorem, in its simplest form (applied to a single event) is:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

Where P(A|B) defines the probability of event A given that B occurred. The theorem defines how hidden causes (A) relate to observed events (B), given a causality model (P(A, B) or P(B|A)*P(A)) and our knowledge of the probability of the occurrence of events (P(B)). The inverse is also true, as we will see further ahead in this section. As an example, P(penalty | goal) defines the probability that a penalty kick was scored, knowing that there was a goal.

There are at least two interpretations to the theorem and regarding how one may think about its results [**?**]:

- Frequentist interpretation: probabilities are defined by the relative frequency of events, given a natural sampling. This means that the probability of obtaining 'Heads' when rolling a dice is equal to the number of 'Heads' obtained after rolling the dice a sufficient number of times relative to the total number of times the dice has been rolled.

- Epistemological interpretation: probabilities represent a measure of belief. It can either be a result logical combination of probabilities through the usage of axioms (it's closely related to Aristotelian logic) or it can also reflect a personal belief (which is called a subjective view).

Table 2.1: Alarm system confusion matrix

	alarm	$\neg alarm$
burglary	0.09	0.01
$\neg burglary$	0.1	0.8

2.2.1.1 An example

One example of the application of this theorem is [**?**]: you know your home's alarm is ringing, but you don't know whether that was caused by a burglar or something else (maybe a bird triggered it, or there was a malfunction in the alarm system). How confident are you that you're being robbed? Consider that the alarm company, based on quality trials, defined in the confusion matrix for P(alarm, burglary) (Table 2.1).

You can interpret each table's cell as P(A, B). For instances, the top left cell is the probability that the alarm rings and there is a burglar, while the bottom left cell is the probability that the alarm rang, but there was no burglar (a false positive).

If we substitute the values of Bayes' rule described above, we get:

$$P(burglar \mid alarm) = \frac{P(burglar, alarm)}{P(alarm)}$$

Where results is 0.09 / 0.19 = 0.47. So, even if the alarm is ringing, there is just a 47% probability that the house is actually being robbed.

The previous example illustrates the simplest case of applied BR, but it is also possible to combine several variables. One way to represent this kind of scenario is by expressing the variables in a directed acyclic graph, where the relation "Parent" stands for "May cause" and you can specify the conditional probabilities of a child given a parent's result. This graphical model is called a Bayesian Network.

2.2.1.2 Bayesian Networks

We can extend our alarm example further, by considering not only a burglar can trigger the alarm, but an earthquake also can (while there can still be false positives). Also, consider that we have 2 neighbors (Mary and John) who may call us whether the alarm is ringing or not. This problem is represented in figure 2.1.

Some interesting question we can ask, given this scenario are:

- If John calls saying the alarm is ringing, but Mary doesn't, what are the odds it really is ringing?
- If the alarm is ringing, was there an earthquake?
- What are the chances that both my neighbors call, the alarm is ringing, but there is neither a burglary nor an earthquake?

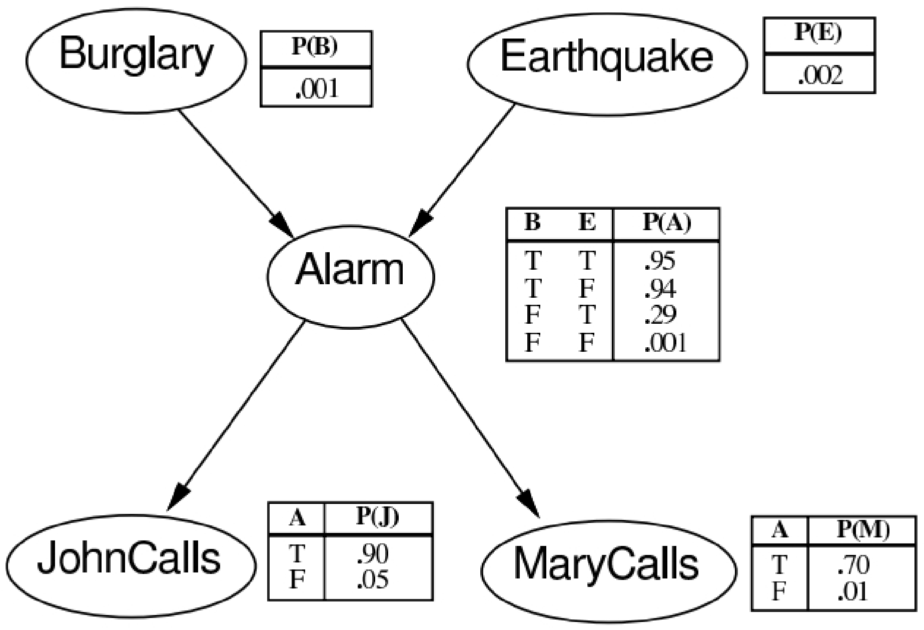


Figure 2.1: Belief network for the alarm problem [Wan02]

This last example, for instances, would be calculated as: P(J, M, A, $\neg B, \neg E) = P(J|A) * P(M|A) * P(A|\neg B, \neg E) * P(\neg B) * P(\neg E) = 0.9 * 0.7 * 0.001 * 0.999 * 0.998 = 0.00062$.

Notice how counter-intuitive this example is: the probability of there being an earthquake is about 32 times larger than there being an earthquake, the alarm ringing and the neighbors calling us, even if the conditional probabilities are reasonably high (0.95, 0.9 and 0.7). This is the result of the calculation of the joint probability being a highly combinatorial problem, which is yet another argument in favor of using PR rather than subjective heuristics.

2.2.1.3 Bayes' and data streams

In practical ML applications, it is often the case that there is an incoming stream of new data, rather than one-time batch calculations. BR can accommodate this way of thinking, which A. Downey called diachronic interpretation [Dow12], where diachronic means that something is happening over time (in this case the probability of the hypotheses, as new data arrives). In order to make sense of this definition, we may rewrite Bayes' rule as:

$$P(H \mid D) = \frac{P(D \mid H) P(H)}{P(D)}$$

Where:

- H: hypothesis
- D: data

- p(H): probability of the hypothesis before the new data is taken into account. Also called **prior**. It can either be calculated using background information or subjectively defined using domain knowledge. Loses significance as new data is added, so its choice is not determinant to the model's performance in the long run.
- p(H|D): what we want to calculate, the probability of the hypothesis after considering the new date. It is called **posterior**.
- p(D|H): probability of the data if the hypothesis was true, called the **likelihood**.
- p(D): probability of the data under any hypothesis, called the **normalizing constant**.

Under this interpretation, you may continuously feed data into the model and see the probabilities getting updated. We will see more practical examples of this in section 2.2.2.

2.2.1.4 Beyond Bayesian Graphical Model

At first glance, someone who is learning for the first time about PR applied to ML, may think that graphical models such as the one presented in Figure 2.1 are the best there can be done in terms of using a graphical interface for solving this kind of problems and that the only thing is missing is an automated way to make the calculations.

While it is true we have still never mentioned techniques or tools that automatically do inference over a Bayesian Network, there are several tools with that capability (including an R package [**?**] or standalone tools [**?**]).

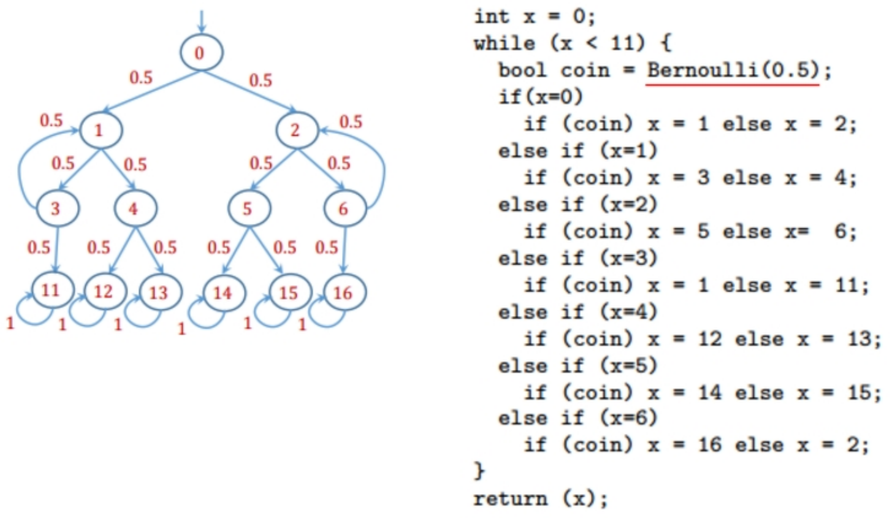
However, not all PR can be done via Bayesian Networks and not all graphical models are enough for complete PR [DL13]. PP are the largest class of models available, and there are also more algorithms for inference than just the calculation of joint probabilities (like we did in the alarm example), as we will discuss in Section 2.2.2.

Bayesian Networks are not the only kind of graphical model. Another one would be Markov Chains, which is yet another example of a model which is not able to represent all PR problems. This is clear when we realize that, while PPLs support numerous distributions (such as Normal, Laplace, Gamma, Half-Cauchy or *t*), all Bayesian Networks and Markov Chain can be represented in a PPL by just using Bernoulli distributions [Gor14]. We can see an example of such a translation in Figure 2.2.

2.2.2 Probabilistic Programming Languages

2.2.2.1 The Probabilistic Program-Model duality

A probabilistic program (PP) is an ordinary program (that can be written in mainstream languages such as C, Java or Haskell) whose purpose is to specify a probability distribution of its variables. This is done by sampling over several executions of the program. The only needed construct the language has to support, in order to be able to write a PP, is having a random number generator



```
int x = 0;
while (x < 11) {
  bool coin = Bernoulli(0.5);
  if(x=0)
    if (coin) x = 1 else x = 2;
  else if (x=1)
    if (coin) x = 3 else x = 4;
  else if (x=2)
    if (coin) x = 5 else x=  6;
  else if (x=3)
    if (coin) x = 1 else x = 11;
  else if (x=4)
    if (coin) x = 12 else x = 13;
  else if (x=5)
    if (coin) x = 14 else x = 15;
  else if (x=6)
    if (coin) x = 16 else x = 2;
}
return (x);
```

Figure 2.2: Translation of Discrete Time Markov Chain to a PPL [Gor14]

[DL13]. This whole concept couldn't be better explained than in this text by Freer and Roy, regarding Church (a PPL, which we describe in Section 2.4.1.2) but common to any PP:

> "If we view the semantics of the underlying deterministic language as a map from programs to executions of the program, the semantics of a PPL built on it will be a map from programs to distributions over executions. When the program halts with probability one, this induces a proper distribution over return values. Indeed, any computable distribution can be represented as the distribution induced by a Church program in this way" [FR12]

One way to think about this notion is by considering that the program itself is the model. An example of the relation between a model (expressed in a PPL) and the implied distribution over its variables (obtained using an inference method) can be seen in Figure 2.3, where a variable *flip* is set to be a Bernoulli distribution and *x* is defined in terms of *flip*. We can then see how the graphic of the inferred distributions of *flip's* and *x's* values looks like and confirm what was to be expected: for *flip's* values lower than 0.5 we see *x* follows a normal distribution, whereas for values greater than 0.5 it follows a gamma distribution instead. The goal of PP (via PPL) is to enable PR and ML to be accessible to most programmers and data scientists who have enough domain and programming knowledge but not enough expertise in probability theory or machine learning.

2.2.2.2 PPLs vs regular PLs

What is then, a Probabilistic Programming Language (PPL)? First of all, it can be a standalone language or an extension to a general purpose programming language. We'll be analyzing exam-

```
flip = rand < 0.5
if flip
    x = randg + 2    % Random draw from Gamma(1,1)
else
    x = randn        % Random draw from standard Normal
end
```

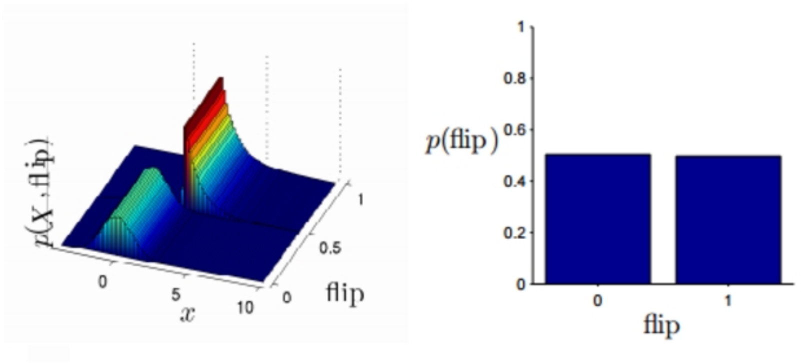


Figure 2.3: Implied distributions over variables [DL13]

ples of languages from either these categories in Section 2.4, but many more exist, such as Figaro [?] (hosted in Scala), webppl [?] (embedded in javascript) or Dimple [?] (which has both a Java and a MATLAB API). The key difference between these languages and a PPL is the latter has the added capability of performing conditioning and inference [?].

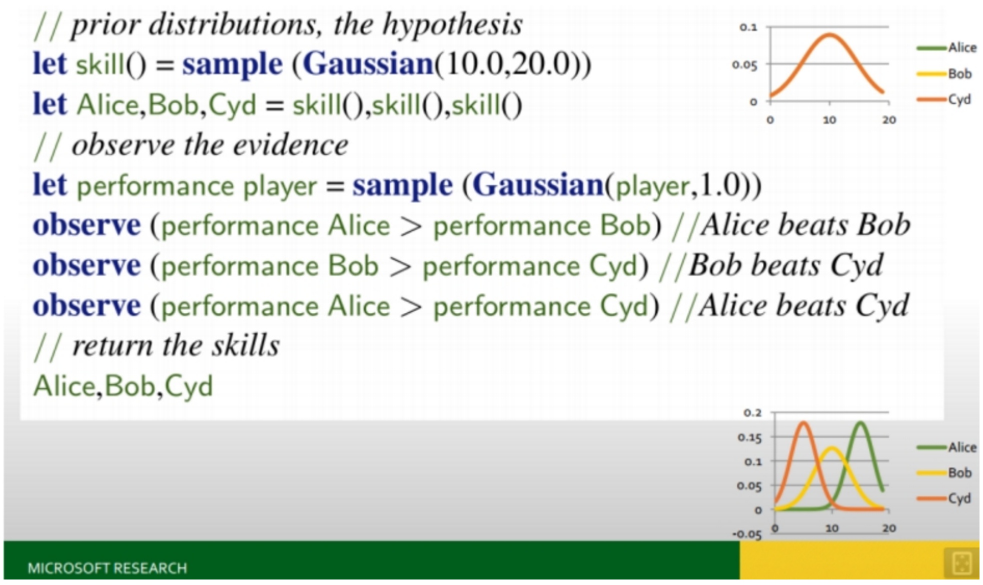
Conditioning is the ability to introduce observations about the real world in the program. That way, you update the prior probability based on those observations. Consider the example in Figure 2.4 (which is a simplified version of how Microsoft applies PP in its Xbox matchmaking algorithm [MWGK12]) where the prior is a normal distribution with equal parameters for all players (shown by the graphic at the top). Then, it defines how the performance of the player is based on his skill (which at the initial point in time, is equal to every one of them) and proceeds to make several observations regarding games between them. Finally, it shows the inferred probability distribution of the posterior on the bottom graphic.

2.2.2.3 Inference

We said that a PPL empowers the user to formalize a model and then query for the probability distribution of its variables, which is automatically done via inference. While general-purpose languages require you to write one-time inference methods that tightly coupled to the PP you are inferring on, PPLs ship with an inference engine suited to most PP programs you can write [?].

An inference engine of a PPL acts similarly to a compiler in a traditional language: rather than requiring each user to engineer its own, which requires significant expertise and is a non-trivial and error-prone endeavor, every PPL has one incorporated.

Having the inference engine work as a separate component, rather than being tightly coupled to each model, opens up a myriad of new possibilities mainly in the form of knowledge and tool



```
// prior distributions, the hypothesis
let skill() = sample (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = sample (Gaussian(player,1.0))
observe (performance Alice > performance Bob) //Alice beats Bob
observe (performance Bob > performance Cyd) //Bob beats Cyd
observe (performance Alice > performance Cyd) //Alice beats Cyd
// return the skills
Alice,Bob,Cyd
```

Figure 2.4: Microsoft Xbox Live True Skill [MWGK12]

sharing, as we have seen in the past in the compiler space. Examples of this would be new compiler and interpreter techniques (such as working towards scalability or parallelization), optimizers, profilers or debuggers.

Another great advantage of having a modular inference engine is that we can try different inference algorithms and pick the one that best fits the problem at hand. When analyzing which algorithm is more suitable for a certain use case, there are certain characteristics worth noting [?]:

- Determinism - in equal initial conditions, an algorithm always yields the same result.
- Exact result or approximation.
- Guaranteed convergence - an algorithm may or not be guaranteed to reach a result at some point in time. If not, it's possible that it will run forever.
- Efficiency - related to how fast can it reach a result.

Microsoft's Infer.NET provides three inference algorithms [?]:

- Expectation Propagation - deterministic, provides exact solutions only in some cases, is not guaranteed to converge and is labeled as "reasonably efficient".
- Variational Message Passing - also deterministic, but always gives approximates results and is guaranteed to converge. It's considered to be the most efficient of the three for most cases.
- Gibbs sampling - non-deterministic, may be able to reach exact result if given enough time to run, has guaranteed convergence and is regarded as not-so-efficient as the other two.

We can divide the three algorithms into two categories: variational bayesian methods [WBJ05][?] and sampling methods (in the case of Gibbs sampling, it's based on Markov chain Monte Carlo) [?]. The main difference, as far as the end-user is concerned, is that variational methods provide faster results but are subject to bias, whereas sampling methods have the potential to produce more accurate results when compared to the ones that resort to sampling (the downside being it's slower and its convergence is hard to diagnose) [?].

Please notice that none of these algorithms provides the same kind of exact solution as the calculation of the joint probabilities we did in Section 2.2.1. The reason for that is that calculating an exact solution takes time exponential in the number of variables to run, even if we have smarter algorithms than the naive calculations we did [?].

2.2.2.4 Openbox models

When compared to traditional machine learning methods (such as random forests, neural networks or linear regression), which take homogeneous data as input (requiring the user to separate their domain into different models), probabilistic programming is used to leverage the data's original structure. This is done by empowering the user to write his own models while taking advantage of a re-usable inference engine. Olivier Grisel called this combination "Openbox models, blackbox inference engine"[Gri13].

Rather than using most of his time performing feature engineering (that is, trying to fit the problem and the data into an existing model), the user will have the tool necessary to design the model that best fits the domain he is working on. Plus, it provides full probability distributions over both the predictions and parameters of the model, whereas ML methods can mostly only give the user a certain degree of confidence in the predictions.

2.2.3 Conclusion

Summarizing, PPLs are a step forward in using PR to solve ML problems since it helps overcome the difficulties in using PR in real world problems. This is done by adding automated general inference on top of a precise specification (the program), where in the past models were communicated using a mix of natural language, pseudo code, and mathematical formulae and solved using special purpose, one-off inference methods.

This encourages exploration, since different models require less time to setup and evaluate, and enables sharing knowledge in the form of best practices and design and development patterns.

However, using a full fledged programming language might still be an entry barrier. We want to help statisticians and data scientists alike to learn faster and be more productive using a PPL in a way similar to the tools they are accustomed to. In order to do so, we'll be combining a PPL with Visual Programming (Section 2.3).

2.3 Visual Programming

Visual Programming (VP) can be defined as "any system that allows the user to specify a program in a two-(or more)-dimensional fashion." [?]. In a textual programming language, even though there are two dimensions (one being the text itself, and the other the optional line-breaking characters), only one of them has semantics, as the compiler processes text as a one-dimensional stream.

Examples of systems with additional dimensions are ones that allow the use of multidimensional objects, the use of spatial relationships, or the use of the time dimension to specify “before-after” semantic relationships [?].

Research has identified several advantages in the use of VP, such as a natural way of expressing semantics, good readability, easy interaction, language independence (though this is not applicable to the work of this thesis, as detailed in 2.3.1), programming at higher levels of abstraction or rapid prototyping [?], which is achieved by providing immediate visual feedback [?].

The advantages of programming at higher levels of abstraction are known, and one of them is it exposes users who are not used completely fluent in textual programming to a reduced number of concepts [?], while decreasing the verbosity of programs, which can even be useful for seasoned programmers [?]. It also reduces the importance of being familiar with syntax, a common cause of difficulty of adaptation among less experienced programmers when learning a new language [?][?]. This difficulty of translating ideas into syntactically correct statements can also be solved by finding alternative ways to communicate instructions to the computer [?].

So, the point of using a VP tool to aid in programming is to overcome the difficulties that many people have when learning a conventional language [?]. It has been shown this approach can help users without prior, or little, programming experience to create fairly complex programs [?]. This is especially true within certain small domains, where the language can be tailored for a subset of tasks rather than trying to be suitable for all kinds of applications [?]. Tools similar to Excel (spreadsheets) are a prime example of this, where the following benefits were identified [?][?]:

- The graphics on the screen use a familiar, concrete, and visible representation which directly maps to the user’s natural model of the data
- They provide immediate feedback
- They supply aggregate and high-level operations
- They avoid the notion of variables (all data is visible)
- The inner world of computation is suppressed
- Each cell typically has a single value throughout the computation
- They are non-declarative and typeless
- Consistency is automatically maintained

- The order of evaluation (flow of control) is entirely derived from the declared cell dependencies

Another example of such a domain would be developing ML applications resorting to probabilistic programming.

Smith and other authors claim that the human thought process is clearly optimized for multidimensional data [**?**][**?**], so all the aforementioned advantages can be explained by how graphical programming is closer to our mental representation of problems when compared to a textual interface [**?**]. As said by Fischer, Giaccardi, Sutcliffe and Mehandjiev:

> “Text-based languages tend to be more complex because the syntax and lexicon (terminology) must be learned from scratch, as with any human language. Consequently, languages designed specifically for end users represent the programmable world as graphical metaphors ... (*such languages aim to*) reduce the cognitive burden of learning by shrinking the conceptual distance between actions in the real world and programming.” [**?**]

This idea as been tested in an empirical study: in a algorithms course of the United States Air Force academy, students have consistently shown a preference for solving problems visually, while it also seems that doing so helped them to achieve better scores in problem-solving exercises and overperform their colleagues who used a regular programming language [**?**]. However, shortcomings of using VP were also identified, as we will discuss in 2.3.4.

2.3.1 Visual Programming Environment

Boshernitsan proposed a classification scheme for VPLs [**?**] that divided VPLs in purely visual languages (PVLs), hybrid text and visual systems, programming-by-example systems, constraint-oriented systems and form-based systems.

In the context of this dissertation, as we want to leverage the advantages of both a VPL and a PPL while avoiding implementing a PPL, so the obvious choice is to use a hybrid text and visual system. We will be calling this system a visual programming environment (VPE). The difference between a VPE and a purely visual language (PVL) is that, while a PVL is a language *per se* (meaning that it there is a direct mapping between graphics and execution), VPEs offer a middle ground between regular textual languages and PVLs: they provide a graphical interface that can be used to generate code for a target language [**?**]. An example of a PVL would be MIT’s Scratch [**?**] and one for a VPE is the Eclipse IDE plug-in WindowBuilder [Fou16].

If conceived correctly, VPEs can help addressing some of the issues raised by critics of VP (detailed in section 2.3.4), such as VP’s inability to solve large-scale real-world problems. This is done by applying VP to only subsets of entire systems, making it possible to combine a general-purpose programming language with the advantages of VP [**?**], since the code generated by a VPE can be seamlessly integrated into any project built with its target language.

Concretely, VPEs are able to overcome the tradeoff of control for simplicity commonly made by VPLs: even if a certain idiom cannot be represented by its graphical form, the user can later edit the generated code to include it. This makes it possible to design scalable programs, both in terms of performance (since the user can still access all the target language's low-level features) and development (because all the advantages of programming visually are still present).

2.3.2 Visual Dataflow Programming

A Dataflow Programming Language (DPL) is built upon the notion that data flows from one node to another. Therefore, in this paradigm, a program is internally represented as a direct graph [?]. This graph is constituted by three kinds of nodes (also called blocks): sources (take no input and produce output), processing/transformation (take an input and produce an output based on it) and sinks (only take input and don't produce output) [?].

The edges behave like unbounded in-order queues and define the data dependencies between nodes [?], connecting input ports to output ports. Nodes process data asynchronously, as soon as all of its inputs are available. There are at least two ways a dataflow program can operate: either from left to right (from higher topological order to lower, also called data-driven or push-based) or right to left (in inverse topological order, called demand driven or pull-based) [?].

Some authors have identified features which are important in a DPL [?][?]:

- Freedom from side-effects: a block can't do more than producing an output based on inputs, being it I/O or variable re-assignment, for instances.
- Locality of effect: this means having well-defined and small scopes. A variable should not be used in more than one block (the one it is connected) to via input port.
- Data dependencies equivalent to scheduling: there is no notion of time in a DPL. However, a block will only run as soon as its inputs are ready, creating an implicit scheduling.
- Lack of history sensitivity in procedures: no global variables. A node can only access its input ports.

Looking at these features, the reader can realize that in a language where all these rules are followed it is also possible to achieve implicit concurrency. Meaning that, once a program is specified, the user gets out-of-the-box concurrency without manually having to program it [?].

As we will see in 2.4, there are several VPL and VPE that resort to the dataflow paradigm. This could be explained by how easily DPL map to a visual representation [?]. This kind of tool can be described as Visual Dataflow Programming (VDP).

VDP shares the same challenges as other VP paradigms: achieve the right level of granularity (abstract neither too much so that the users can't express everything he needs to nor too little, making the representation as complex as its textual counterpart).

A big challenge in VDP that is not present in other VPL paradigms is not how to transpose the dataflow graph to a graphical representation but rather representing control flow semantics within dataflow [?]. Several solutions have been proposed, some of which violate the pure DFP principles by not being completely stateless while others introduce cycles in the graph [?].

2.3.3 Evaluation

Even if the goals of VP are clear (to make programming more understandable, ensure correction and be faster to develop in) the best way to achieve them is still a matter of discussion. Like in the design of any programming language, there are some best practices to guide the design of a VPL, some of which have been identified by Burnett [?] and are listed below.

- Concreteness - it's opposite of abstractness. The program express values and instances of values rather than meta-information such as types or classes.
- Directness - work with concrete values rather than possible ones.
- Explicitness - everything that there is to be known about the program can be easily understood by looking at the graphical representation and the user does not need to infer semantics by himself.
- Immediate visual feedback - every change to the program should be immediately propagated to change in the affected output. Spreadsheets are an example of this.

However, these are just guidelines and do not guarantee the efficacy of a VPL. Some VPEs sacrifice some of these principles for the sake of completeness: Viskell (described in 2.4.3.2) violates directness by allowing to work with the option monad, or concreteness by placing a great emphasis on type definition.

Whitley and Blackwell [?] said that because the design decisions in a VPL lack formal basis, the only way to assess if they really contribute to facilitating the programmer's cognitive processes while programming is through empirical studies. In their studies, they have found that while subjects cited ease of learning as a benefit of VPLs, their opinions differed when asked if using a VPL had a positive impact on productivity during a project.

They also claim that there is a gap between academia and industry programmers. In contrast to the first group who tends to focus on theories of cognition to justify the use of VPL, the second is more interested in potential improvements in "potential improvements in productivity that arise from straightforward usability issues".

Some metrics that can be taken into account when assessing a VPL's efficacy are learning time, execution speed and retention [?].

2.3.4 Criticism

In this section, we'll be trying to summarize some of the criticism made to VP, while proposing solutions to some of the issues mentioned and discard some of the others as non-applicable in the context of this thesis.

There are people who claim VPLs lack visual abstraction mechanisms that are as effective as those offered by text-based languages, so they are not well-suited to develop large applications [?]. Some of the techniques currently used in real-world software development include iterative design and interactive prototyping, two principles that are promoted by the usage of VPLs. Also, it has been shown that the richness of the visual paradigm introduces new ways of approaching programming problems, particularly for those not trained in traditional software development methods [?] (such as data scientists, which constitute the target audience for this work). Studies have also shown that fairly complicated algorithms, such as garbage collection, could be described graphically [?].

Green also discusses, contrary to some other evidence we discussed before [?][?], how lab studies failed to collect evidence in favor of the productivity gain of VPLs, even though he admits users like and use them [?]. He also points how that VP systems "do nothing that can't be done as well or better with straight text" and identifies the real issues as "how layout and locality can be used to convey meaning". According to our definition of VP, every system that allows the user to express himself in more than one dimension (such as using layout and locality, as proposed by Green), so it seems that the proposed alternative could be a VPL.

In his *"No silver bullet - essence and accidents of software engineering"* paper, Brooks says that *"A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design ... Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will. In the first place, ... the flowchart is a very poor abstraction of software structure.... It has proved to be useless as a design tool.... Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram.... More fundamentally, ... software is very difficult to visualize."* [?]. While one may be tempted to be convinced by Brooks' initial rhetoric, what he wrote does not seem to apply to the work of this thesis because: a) we won't be using executable flowcharts, but rather a dataflow VPE approach, as described in 2.3.1 and b) as time passes screens are getting bigger and with higher resolutions, a low-end screen by today's standards would be state of the art in 1986, when Brooks wrote that. The final claim that software is hard to visualize, backed by Dijkstra's letter where he states that *"I was recently exposed to ... what ... pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was ... an obvious case of curriculum infantilization ... We must expect from that system permanent mental damage for most students exposed to it."* [?], is contrary to many studies done in cognitive science [?][?][?].

Myers admitted that the key for a successful application of VP to a real-world problem was to identify "appropriate domains and new domains to apply these technologies to" while recognizing that we have already witnessed how VP can help non-programmers work in limited domains [?].

We do believe that data mining can be one of those domains (as shown by the popularity of RapidMiner) [Pia15], so it would make sense to extend the current state of the art with a tool that combines VP and PP. This view is aligned with Burnett's claims that VP makes programming easier for specific audiences [?]. It is also our belief that, by only applying VP to a specific part of a larger system (in this case, the ML module that uses PP) we can overcome the problems that usually arise when scaling up VPL [?]; successful examples of such an approach include the design of GUIs via a VPE (such as WindowBuiler [Fou16] or the Android Studio's layout editor [?]).

Myers also identified some of the problems that are yet to be solved regarding VP [?]:

- **Difficulty with large programs or large data**. Almost all visual representations are physically larger than the text they replace, so there is often a problem that too little will fit on the screen.

  We intend to solve this issue by studying the application of some techniques that provide a greater level of abstraction in order to be able to transmit more semantics while avoiding the layout to get cluttered with details [?].

- **Poor representations**. Programs are hard to understand once created and difficult to debug and edit. The larger the size, the worse this problem becomes.

  This is related with the previous point and the proposed solution is similar.

- **Need for automatic layout**. When a program gets too large the layout becomes too hard to manage, a single addition of a piece may oblige the user to move a great number of blocks in order to avoid collisions and preserve readability. One way to deal with this would be to automatically generate an attractive layout.

  This problem is out of the scope of this thesis, although it certainly seems important for the future of VPL.

- **Lack of formal specification**. Currently, there is no formal way to describe a VPL such as the Backus-Naur Form, even if some work towards one has already been made [?].

  Again, this is out of scope, even if we will make an attempt to specify a grammar that defines the boundaries of what would be a valid graph.

- **Lack of Portability of Programs**. A program written in a textual language is as portable as a text file but VPLs require special software to view and edit.

  While the implementation of the VPE's frontend to be done in this thesis is still a matter of study and is undecided, there is the possibility of building one using the HTML/CSS/-javascript stack, making it portable across browsers for view and edition. It is also our aim to provide an intermediate representation in a format such as JSON or XML, so that a program built in a certain frontend can be processed by any other.

- **Tremendous difficulty in building editors and environments**. In 1990, each graphical language required its own editor and environment, and there were no general purpose VPL editors.

  Currently, there are alternatives of re-usable frontends for VPLs, such as Blockly [fE15] or GoJS [?].

- **Lack of evidence of VPLs' worth**.

  This issue was discussed in section 2.3.3, and we conclude that within certain domains (such as VP applied to PP) and target users (inexperienced programmers), there may be benefits in productivity when using a VPE.

Another question that arises is how to represent and manipulate arrays. In the empirical study made by the creators of RAPTOR, students performed statistically significantly worse on the array question when using a VPL [**?**]. This is something to consider in the future, to investigate if handling arrays functionally (by considering them as immutable values where common functions over iterable data structures could be performed, such as map, filter and reduce) could improve users' usage of the VPL.

One of the concerns of another empirical study's respondents was that high-level VPLs might deny them access to the low-level facilities of the machine that are so important in PC programming [**?**] but, as stated before, this problem is alleviated (if not completely eliminated) by using a VPE rather than a purely visual VPL, since the user can later edit the generated code, where he has access to all the language's features.

2.4 State of the Art

The purpose of this section is to try giving an overview over the existing tools currently used in either VP (purely visual VPLs or VPEs) or PPLs.

2.4.1 PPLs

There is a great number of PPLs, so our criteria to pick which of them to analyze was popularity, which we assessed based on a rough estimate of the number of papers that referenced each language. At the same time, we tried to pick languages from different ends of the spectrum: functional/imperative/object-oriented, statically/dynamically typed, one that runs in the JVM, one from the .NET stack and another one written in javascript which runs in the browser.

One advantage of the languages that do not require compilation (such as WebPPL or Church) is that we can display results to the user faster and without having to re-compile every time we change the model, making it as close to immediate visual feedback as possible.

The way we're planning to validate this thesis hypothesis is by converting models in the literature from a textual form to a graphical one (more on this in section 4), so having a reasonable amount of models in a given language acts as an incentive to use it as a target for code generation in the tool we'll develop.

2.4.1.1 WebPPL

WebPPL was written as part of a course on PP and inference in PPLs [**?**]. Therefore it serves more as an educational tool rather than a language that aims to be production-ready, despite including several inference algorithms (such as MCMC and variational ones). It is hosted in javascript, meaning that it extends the language's capabilities and is totally cross-compatible; in short, it acts as a library.

Its main advantage and the reason why it would be interesting to be the target of our tool is that, because it is written in javascript, it runs in the browser without having to be ran externally, reducing the time between designing and getting visual feedback.

2.4.1.2 Church

We started by looking into Church because it is based on pure Lisp (therefore it is also based on lambda calculus and purely functional programming) [?] and believed it could be more expressible (in the sense that we could express equally complex models with less code) than its non-functional counterparts. While maintaining the same Lisp-like syntax, it also has an implementation in javascript called webchurch [?], with all the associated advantages (described in 2.4.1.1).

The Church language is being replaced by VentureScript, a language that is part of the Venture platform for PP and that can be written both in a Church or javascript-like syntax [?]. Having been written by the same authors as WebPPL but with the purpose of being used to solve real problems instead of just acting as an educational tool, if we were to choose a PPL hosted in javascript to support in our tool, the natural choice would be VentureScript.

2.4.1.3 Infer.NET

Infer.NET is being developed by Microsoft Research Cambridge and intends to empower .NET languages (not only C#, but all languages in the stack, including F#, Visual Basic and Iron Python) with PP capabilities [?].

Similarly to C#, it requires a compile step and is statically typed. Unlike Church/VentureScript, it cannot represent all kinds of PR models, as it fails to handle non-parametric models.

It has several advantages that make it a prime candidate to be the target of this dissertation: extensive documentation (not only a reference manual of the API), several examples with various levels of complexity (which is very useful for our hypothesis validation's process) and an active community, with the contributors actively participating in a discussion forum and available to clear any doubts users might have. These reasons, as well as being part of the popular .NET stack, make Infer.NET a strong candidate to be used as the target language of our tool.

2.4.1.4 Figaro

Figaro is similar to Infer.NET: a statically typed language that requires compilation and runs a popular environment (in this case, the JVM, since Figaro is hosted in Scala) [?]. It is used by Charles River Analytics in production, so in spite of being still under development it seems to be useful for end-users and is not restricted to research groups.

Unlike Infer.NET, it can represent arbitrary models without restrictions, but examples using the language are scarce so using at as a basis for the thesis would require extra work converting examples from other languages to Figaro before developing them in a graphical manner.

2.4.2 Tools using VDP

2.4.2.1 NoFlo

NoFlo is a visual open-source implementation in JavaScript of Flow-Based Programming [Ber] (which is a form of dataflow programming) and was designed for general-purpose programming, even if it is more suitable for web programming (being written in JavaScript it has easy access to DOM manipulation capabilities), there are examples of it even being used for controlling a drone.

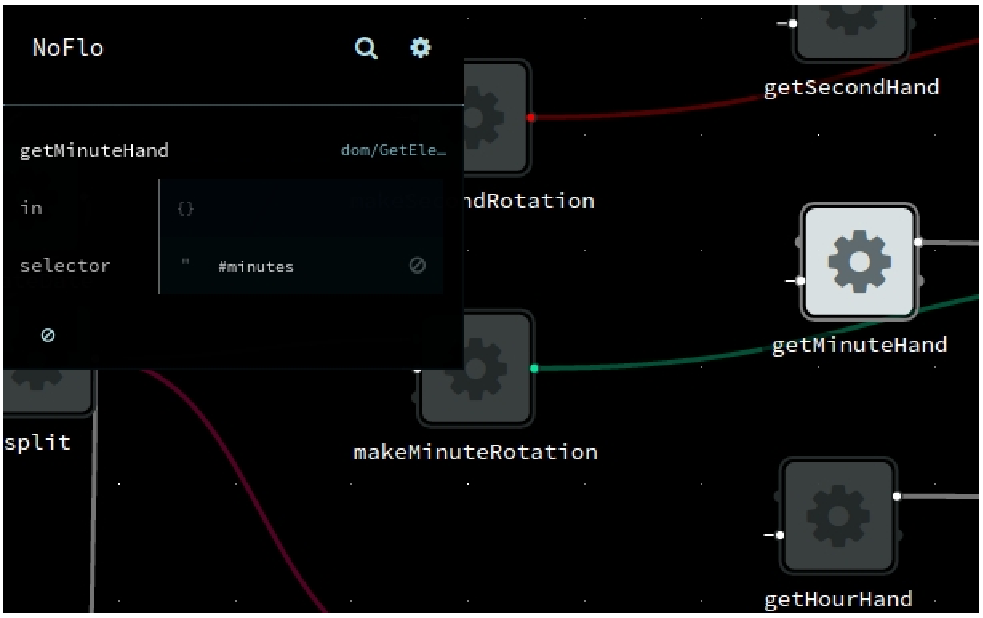
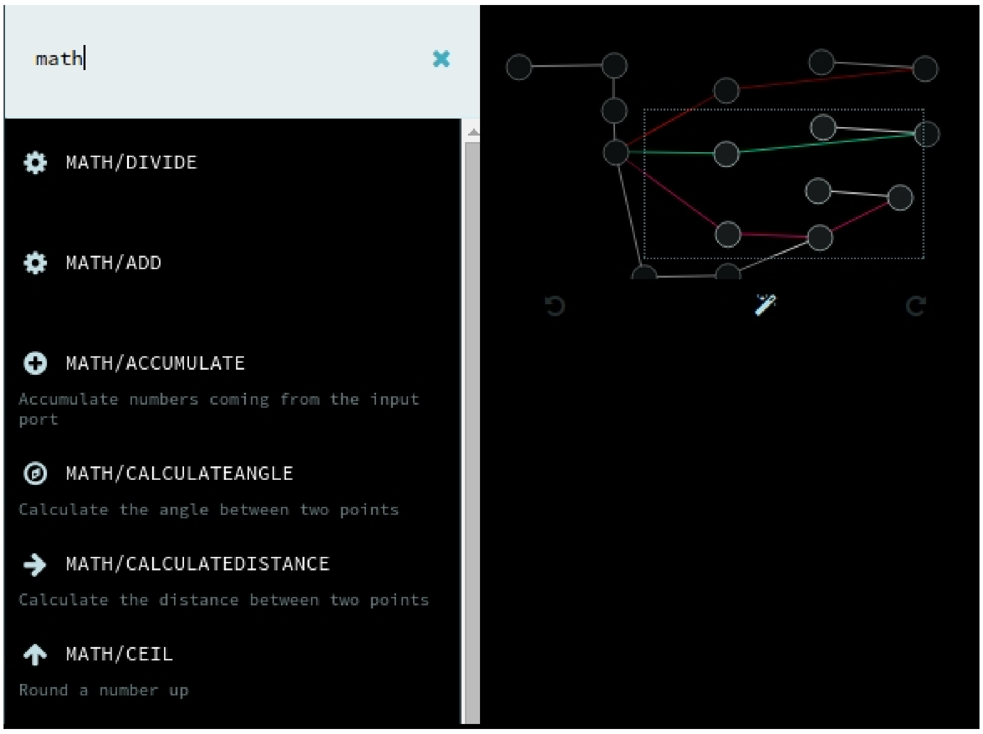


Figure 2.5: Example of node expansion in NoFlo [Ber]

From experimenting with the tool, we found some characteristics that serve as a lesson of what works well and what doesn't:

- A node has the least information possible, yet it can be clicked to show details, such as some usage notes or seeing some inputs or parameters. An example of a common NoFlo layout with a node expanded can be seen in Figure 2.5. This contributes to saving space in the layout while maintaining flexibility.
- You can search for a block by its name (see Figure 2.6).
- There is a minimap that helps to navigate the screen, so we can have a higher level view of the layout 2.6).


math

MATH/DIVIDE

MATH/ADD

MATH/ACCUMULATE

Accumulate numbers coming from the input port

MATH/CALCULATEANGLE

Calculate the angle between two points

MATH/CALCULATEDISTANCE

Calculate the distance between two points

MATH/CEIL

Round a number up

Figure 2.6: NoFlo node search and minimap navigator [Ber]

- Weak typing makes it confuse to use some of the nodes, it is unclear when we can connect input and outputs with mismatching labels because sometimes they are compatible and other times they aren't.

- Sometimes errors appear immediately, others we can just see them in runtime. This would be fine as long as it was clear that there was an error, but that is not the case because the messages blend in with the UI and are not easily noticeable.

- When choosing new blocks, it is often hard to understand what is their functionality because most nodes have no description other than their name.

- Blocks have icons, which presumably provide some visual information about their functionality (see Figure 2.6), but is hard to understand what the difference between the various icons is. What seems to be a good idea, to have icons in blocks to identify them by functionality, as the opposite effect and negatively affects usability.

The last four items can convince the reader of the following conclusion: in VPLs, explicit is better than implicit. Clarity can undoubtedly make all the difference between a language that can be intuitively used and one that, despite all its virtues, could take some more time to master.

2.4.2.2 RapidMiner

According to the popular data mining portal KDnuggets' latest tool popularity poll [Pia15], RapidMiner is the most popular visual tool to use in applied machine learning. It is also the most complete, featuring 1500 different blocks [Rap].

The product has too many features to be able to analyze deeply each one of them, including cloud and repository capabilities for storing and sharing work, mechanisms for deploying models developed locally to a production server or an API for developing new blocks. RapidMiner is actually more of a suite rather than just a tool, so in our analysis, we'll restrict ourselves to the design of a model.

The most differentiating factor between RapidMiner and the other VPLs we studied is that it splits its view between process and results. In the first one, the user can design the model and then he must run it and go to results to see the output. Even if goes against the principle of providing immediate feedback to the user advocated by some VPLs' designers (as discussed in 2.3), it does seem to make sense in its use case. The reason for that is that production machine learning applications running batch jobs can take a reasonable amount of time to run, and the analysis of its results before changing the model can also take some time. This is different from people using 3D editors (like Blender), where they the trial and error cycle is much shorter, and users need to get feedback about their changes much more often.

Having no need to be continuously looking at the output reduces the importance of it being shown in the same space as the program we're designing, and we can save space in the layout by moving it to a different tab. On the other hand, RapidMiner is flexible enough that allows the user to place both the process and the results windows side by side, so it really is up to the user to decide how important is it to be seeing everything at the same time. A tool which doesn't support moving windows and tabs must make a choice whether or not to have the output shown alongside with the program.

Other strengths of RapidMiner include having error indicators in each block and thorough descriptions of each node's functionality, input, output and parameters, including examples. It also provides visual clues about a node's functionality by using not only meaningful icons but also colors. This can be observed in Figure 2.7, where file reading nodes are gray, model-related ones green and those related with data preparation are in pink.

Something that doesn't seem to work particularly well for someone using the tool for the first time are the abbreviated labels in the ports. They are cryptic enough that an inexperienced user has to click the node and read the description to understand what they are. Maybe it would make sense to either write the full label for the sake of clarity or just don't show anything at all.

2.4.2.3 Weka Knowledge Flow

Weka Knowledge Flow is one of the available frontends (others being APIs and a GUI Explorer) to WEKA's core algorithms [HR08].

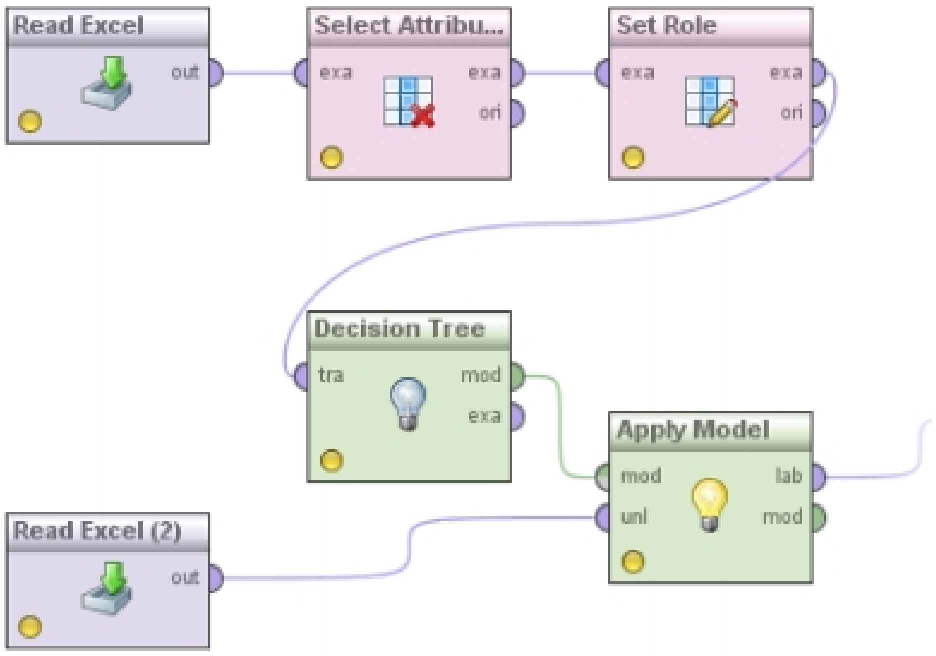


Figure 2.7: Classifier in Rapidminer [Rap]

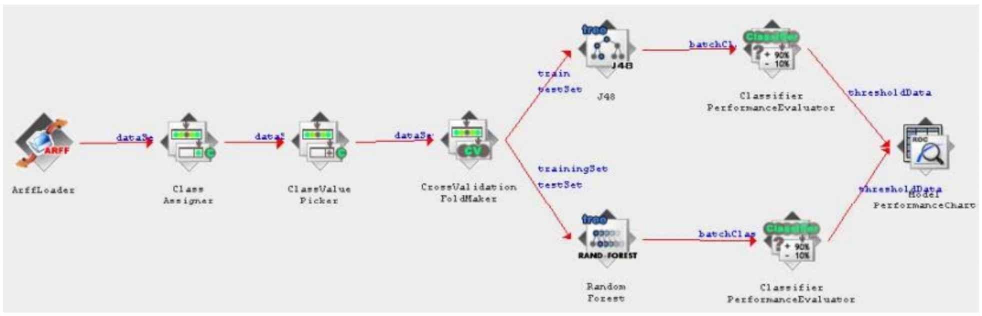


Figure 2.8: ROC curve in Weka Knowledge Flow [HR08]

In this tool, inputs are not connected to specific ports, but rather to a block. The semantics of the connection is expressed on the edge's label. This can save some space in the layout as the nodes can be smaller, but in order to understand a node's input, the user has to waste time following all the edges backward until he finds the label, which doesn't seem a reasonable tradeoff.

Weka Knowledge makes a good use of icons to differentiate between different node types. Not only is it easy to the user to distinguish between source, processing and sink nodes, but also between the different processing nodes' categories (such as filters, classifiers and clusterers).

2.4.2.4 Blender Composite Nodes

Blender Composite Nodes is part of the Blender open-source 3D computer graphics software and provides a way for the user to build visually (through a dataflow representation) a script that can be applied to images or Blender scenes [Wik].

It incorporates common VDP idioms such as source nodes (such as render layers or RGB), processing nodes (where you can apply blur, shift an RGB curve, etc) and sink nodes (it's the way to finalize a script and save, or display, the result). It is interesting to note that Blender Composite Nodes does not incorporate processing nodes with filter semantics [1], which is an example of a VPL that intentionally did not incorporate a common idiom in VDP because it does not make sense for its domain.

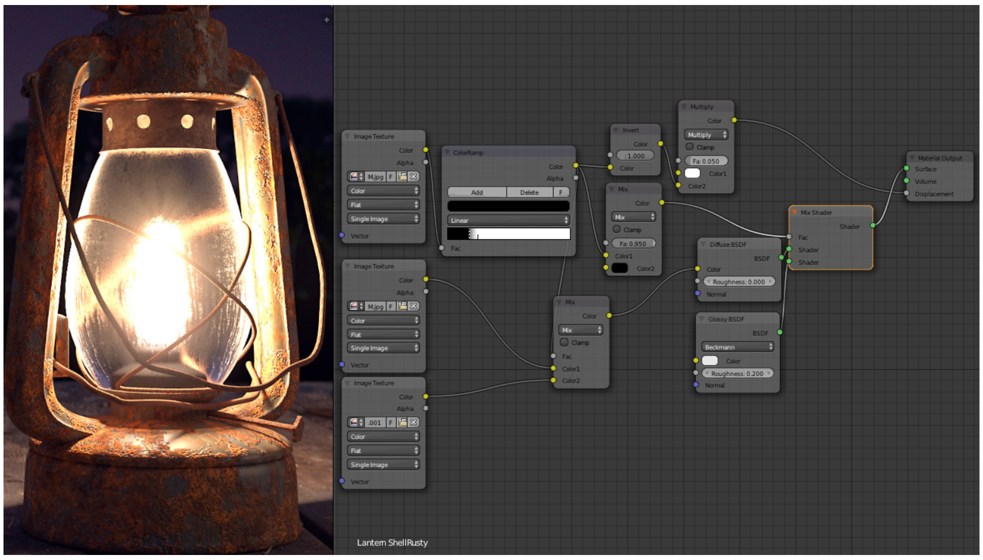


Figure 2.9: Script for texture transformation in Blender [Wik]

There are several interesting characteristics that are worth highlighting:

- Composite Nodes assigns a color to each type, so that the user can easily understand the type of each input or output (see Fig. 2.9).

[1]We mean by filter that the result is a subset of the input. Blender has graphical filters, which are analogous to a functional map operation.

- Implicit conversions for some of the types. For instances, a conversion from a vector (which are always three-dimensional) to a value yields the mean of the vector's elements. In figure 2.9 we can see that some values are connected to a different type (among others, the first input node's color is connected to a simple value named 'Fac'), which is valid behavior.

- Incorporating a custom interface within each node. This helps to reduce the space occupied by the program, since there is no need to create inputs for every single input in a processing node. It also simplifies the VPL's usage, because since some types can only exist within certain nodes, and by not being able to instantiate them outside those nodes, we reduce the possible choices for input nodes. An example would be the image type enumeration, that we can see in figure's 2.9 input nodes.

- Node groups which are essentially an encapsulation of a set of nodes, resulting in another node that can be re-used, help the user create his own set of abstractions. Our texture transformation example could be saved into a node group, so that we can use it in another script without needing to copy everything from one place to the other or cluttering the new script's interface with several new nodes. The concept is analogous to functions in programming.

2.4.3 VPEs

In this section we'll be describing three visual programming environments. By VPEs we mean tools where the user can express the program in a graphical manner that will later be translated into valid code, regardless of it being executable directly on the VPE or not.

2.4.3.1 Blockly

Blockly is more than a VPE, it actually is a library for building VPEs [fE15]. It is fully open source, runs on the browser (in a fully client-side manner) and was developed to be extensible.

Blockly does not adopt the VDP paradigm, it instead uses a system of blocks similar to a puzzle, as you can see in Figure 2.10. Another useful feature is basic type checking, so that users can't connect blocks that wouldn't make sense, such as applying an uppercase function to a number. The way Blockly is built, it is possible to stop the user from producing an invalid program.

According to its authors, the reason why they didn't choose to adopt VDP is that they believe the generated code doesn't look like the VDP graph, making it harder from a user who later transitions from the tool to textual programming to adapt to the new format.

It is possible to define new blocks as well completely controlling how generated code will look like, as well as defining new types. These features seem to be enough that Blockly can be considered a framework complete enough to be suitable for building most VPEs and, because it is open-source, it can be unlimitedly extended.

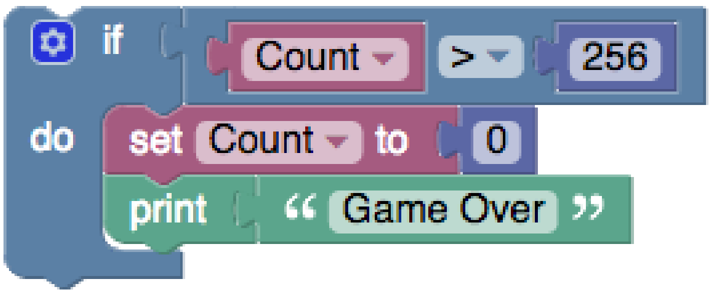


Figure 2.10: Blockly example [fE15]

2.4.3.2 Viskell

Viskell is *"an experimental visual programming environment for a typed (Haskell-like) functional programming language"* [Boe16]. Its authors assume their main goals are, among others, to create a readable and compact visualizations for functional programming and addressing the scalability issues of larger visual programs. They state that, in order to achieve the latter goal and being able to scale up, Viskell should combine a simple core language with a good type system.

This VPL does not resemble the most popular ones, having some main differences: the program is designed from top to bottom rather than left to right, blocks don't have icons (using explicit names instead) and all ports have explicit types. This class of design choices doesn't seem to be clearly superior to other state of the art tools, so probably we won't be including them in our VPE.

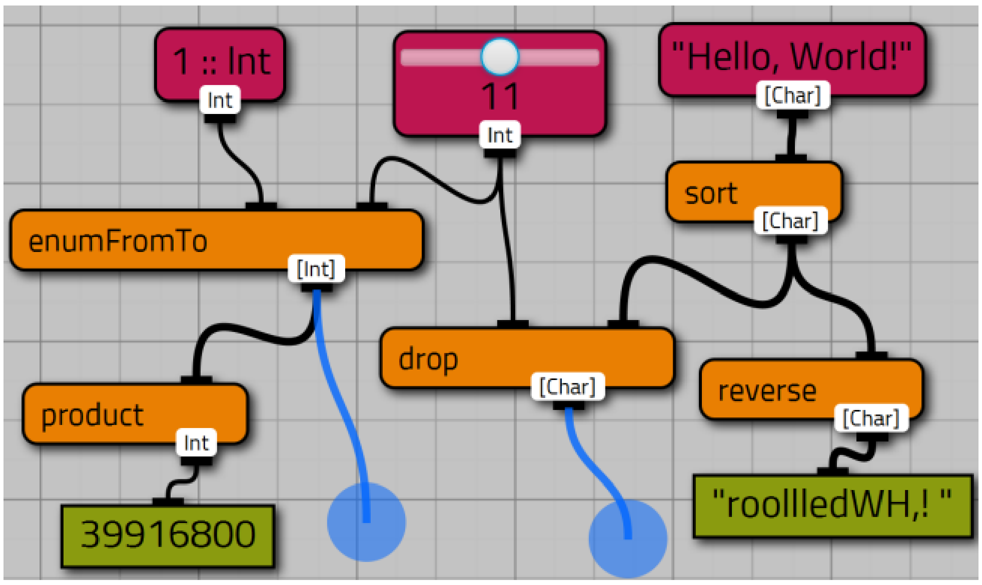


Figure 2.11: Viskell example [Boe16]

Nevertheless, there are other interesting features in Viskell, many of which are tightly coupled to functional programming (such as how it supports currying, anonymous functions or case expressions). The exception is the capability to have an incomplete graph and allowing the user to insert code textually to fill in the gaps (see the blue areas in Figure 2.11).

Being a work in progress, its authors have pointed out ideas of how to deal with increased program complexity that they have still not implemented in the language. Some of them are shown in Figure 2.12 and outlined below:

- Vertical composition of blocks: in the example it is extensively used in the last ("concatMap") block.
- Inline display of constants: as seen in the first block "enumFrom", where 1 is written inline rather than being a separate source block. It is a form of horizontal composition.
- Functions as constants to higher order functions: represented by the right-hand side of the 3 blocks in the middle ("map", "groupOn" and "concatMap").

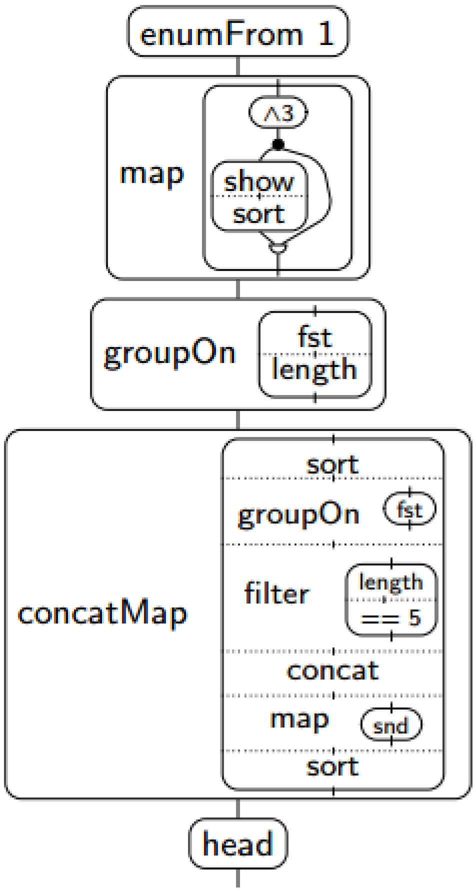


Figure 2.12: Compressed visual representation example [Boe16]

This way, we eliminated most of the connections between nodes and achieved a more compact (we have 4 arcs in a representation that would use more than 20 in its original format) and readable representation.

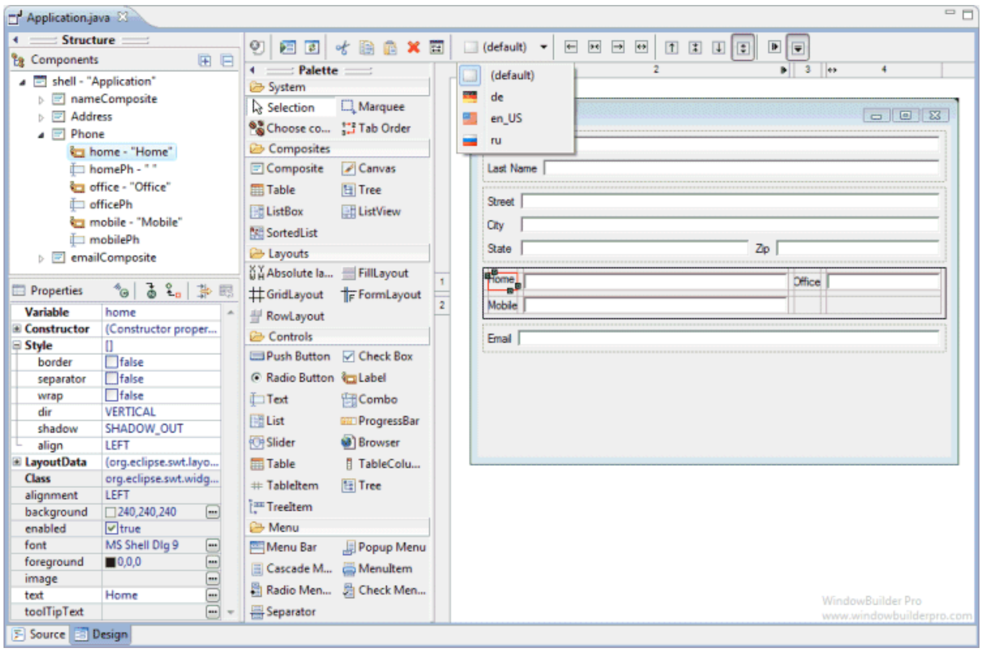
2.4.3.3 WindowBuilder

One domain where VPLs have been successfully applied is UI design. Instead of going through the tedious task of writing boilerplate code (often having to memorize complicated function signatures) and needing to memorize APIs, users can just drag and drop elements on the screen.

The Eclipse IDE provides several plugins on its marketplace, one of which is the WindowBuiler plugin [Fou16] to build Java SWT/Swing UIs.

WindowBuilder is a bi-directional and WISIWIG tool (What You See Is What You Get), meaning that it generates code that faithfully represents the drawn UI and it is also possible to directly edit the code and see the updated result back in the design view.

Since defining UIs is a very different domain from PP there are not many features that we can use from WindowBuilder in our future VPE. The exception is the bi-directionality of code, which could be useful in certain scenarios.

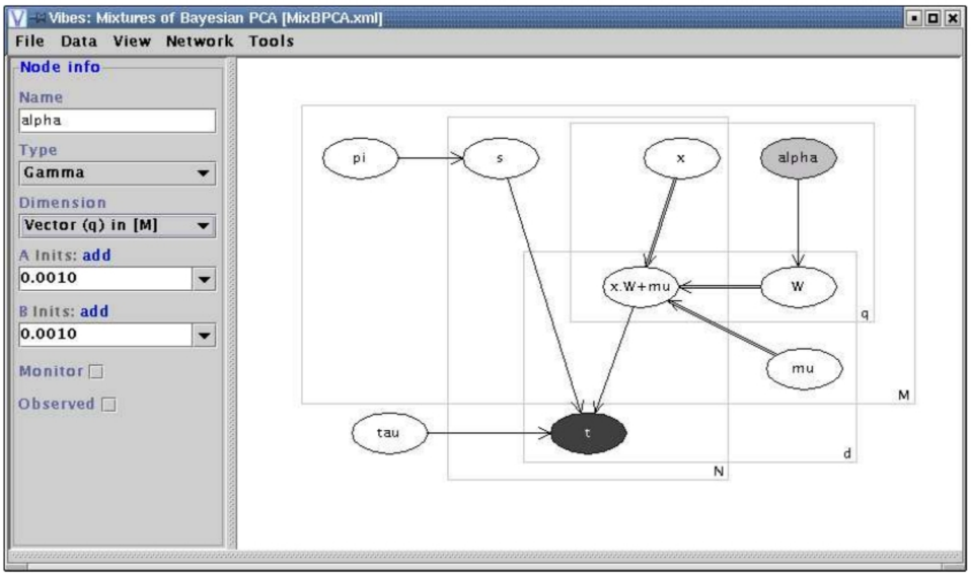


Variable	home
Constructor	(Constructor proper...
Style	[]
border	false
separator	false
wrap	false
dir	VERTICAL
shadow	SHADOW_OUT
align	LEFT
LayoutData	(org.eclipse.swt.layo...
Class	org.eclipse.swt.widg...
alignment	LEFT
background	240,240,240
enabled	true
font	MS Shell Dlg 9
foreground	0,0,0
image	
text	Home
toolTipText	

Figure 2.13: WindowBuilder's design view [Fou16]

2.4.4 VIBES

VIBES is a tool, developed by John Winn during his PhD, that allows variational inference to be performed automatically on a Bayesian network [WBJ05] defined graphically. It was written to serve as proof of concept for the variational message passing (VMP) algorithm, so it has reduced focus on human-computer interaction. Other than its representation of deterministic inputs with a different edge style than undeterministic ones and usage of plates to represent loops, from a usability point of view it's not very interesting.


Vibes: Mixtures of Bayesian PCA [MixBPCA.xml]
File Data View Network Tools
pi
s
x
alpha
x.W+mu
W
q
mu
M
tau
t
d
N

Node info
Name: alpha
Type: Gamma
Dimension: Vector (q) in [M]
A Inits: add
0.0010
B Inits: add
0.0010
Monitor
Observed

Figure 2.14: Example of a model using VIBES [WBJ05]

2.4.5 WinBUGS

Similarly to VIBES (described in 2.4.4), WinBUGS is a framework with a graphical component which s part of the Bayesian inference Using Gibbs Sampling (BUGS) project. It allows the user to, graphically or textually, define a model and automatically perform inference on it (unlike VIBES, the inference algorithm is Gibbs sampling, a method that resorts to a Markov chain Monte Carlo technique) [?]. Its interface is very similar to VIBES and the only difference between them, besides the algorithm used, is that WinBUGS provides a bi-directional conversion between graphical and textual representations.

$$\left\{ \begin{array}{rcl} \alpha,\ \beta & \sim & \mathrm{NORMAL}(0, 10^{-6}) \\ \tau & \sim & GAMMA(10^{-3}, 10^{-3}) \end{array} \right. , \quad \left\{ \begin{array}{rcl} Y_i & \sim & \mathrm{NORMAL}(\mu_i, \tau) \\ \mu_i & = & \alpha + \beta(x_i - \bar{x}) \end{array} \right.$$

```
model
    {for (i in 1:N)
        {Y[i]~dnorm(mu[i],tau)
         mu[i]<-alpha+beta*(x[i]-mean(x[]))}
     alpha~dnorm(0,1.0E-6)
     beta~dnorm(0,1.0E-6)
     tau~dgamma(1.0E-3,1.0E-3)
     sigma<-1/sqrt(tau)
     }
```

Figure 2.15: Example of a model translated to WinBUGS textual form [Cra04]

In Figure 2.15, we can see that WinBUG's textual representation of a simple model can already be harder to understand than the mathematical formalism it represents. This is exactly the problem we want think it can be solved with a proper VPE.

2.5 Conclusions

In this section we have seen the application of VP concepts to tools, confirming the usefulness of many of the concepts studied in the Secton 2.3.

By comparing the different VP applications, we have identified useful features such as the use of color and icons to transmit meaning, block compression (either hiding block details or even the grouping of several blocks in different manners), zooming in and out, minimap navigation, explicit errors, bi-directionality (being able not only to convert graphics to code, but also vice-versa) and strong typing.

Equally useful to knowing what features are useful is to understand which pitfalls to avoid, such as not including labels or icons for the sake of having a simpler interface, resulting being an interface where semantics are too implicit and hard to capture quickly. One of main goals is to avoid having a VPL that is harder to understand than the original representation of the model we are developing, similarly to what happens with WinBUGS textual representation.

Chapter 3

Problem statement

As already described in Section 1.3, this dissertation aims to solve the problem of PPLs having too much of a steep learning curve for someone who is inexperienced in programming, even if that person would've enough knowledge in statistics to leverage PPLs' power in applied machine learning. We propose to do so by showing how we can develop or extend an existing VPL (more specifically a VPE) so it can capture common PPL semantics so that it is user-friendly and yet flexible enough to be able to express solutions for non-trivial problems that could be solved via a PPL. Since the existing work joining VP with PP is almost nonexistent, the notable exception being VIBES and WinBUGS (even if they have their shortcomings, as described in 2.4.4), we have identified margin for improvement.

We will be comparing alternatives and proposing solutions to the common issues that arise when trying to create a VPL with PP semantics. However, this does not mean that the VPL must, or should, act as a mirror to the underlying PPL. The objective is not to have a VPL that can express all of the language's syntax, acting as a mirror or a complete interface for the textual format it represents, but rather to enable the change of paradigm from object-oriented, procedural or declarative programming to a visual one. nonetheless, if this scenario happens and we happen to be able to represent fully a PPL, that is not a problem by itself, but just a consequence of the choices made during the creation of the VPL.

VPEs have been successfully applied to other domains, and considering previous studies in VP that suggest they are well suited for both limited domains (such as PP) and inexperienced programmers, it is the author's belief that the PPL toolset would benefit from such a tool. Therefore, we intend to validate the following hypothesis:

> "When a user produces a probabilistic model via a graphical representation that automatically translates itself into executable code, instead of specifying it textually, he will do so in a shorter amount of time, make fewer errors (both during development and regarding the final solution) and will reach a final representation that is more understandable and thus easier to maintain."

3.1 Validation Procedure

Because there isn't a standard method for evaluating if a language is easier to work with than another, the question arises on how to evaluate success. The optimal way to do so would be to make an empirical study. However, this requires finding a significant amount of people from the target audience (described below) willing to participate in the study (which is not an easy task, because it represents a rare profile) as well as having a tool mature and stable enough so that the study's results could be considered reliable and representative of an underlying idea (in this case, that a VPE can boost user's performance when developing models with a PPL). The goal of this dissertation is to assess several ways how such a tool could be built, pick one of them and develop a prototype; is not to develop production-grade software.

An alternative to this empirical evaluation is to gather examples of probabilistic programs expressed in a PPL (either the one who chose to serve as backend, or another with similar capabilities) in its traditional textual form, translate them into our graphical language, and compare the two regarding how easy would it be for someone other than the program's author to understand it and the safety nets provided for the programmer (runtime errors, compiler warnings or other forms of preventing errors).

As such, our validation procedure consists of a model-evaluate-extend loop where we start by trying to model a problem in our VPL and then look at the result: if we couldn't model the problem by lack of semantics, or the representation lacked essential characteristics (such as type safety), we extend the VPL; after this is done we continue to the next example. This iteration continues until we are able to represent a significant amount of examples, so that we can show how to extend a VPL to support a variety of PP constructs with different semantics.

3.2 Target audience

The resulting tool is aimed at people with knowledge in statistics who are inexperienced programmers. This may include data scientists, researchers, mathematicians or statisticians. In short, anyone who would apply PP to problem solving but is not fluent in textual programming.

3.3 Expected contributions

As the result of this work, we built a VPE for PPLs that can be extended with more PPLs, even if we are only implementing the adapter for one. By doing so we: (1) provided a platform that enables other people to experiment and make usability research on, (2) define a visual language that can be applied to PP in general, (3) drawn some lessons learned while implementing such visual language, and (4) initiate the study on the viability of applying VP to PP to enhance end-user's productivity.

Chapter 4

Implementation details

In this section we will be explaining what is our VPE capable of doing and why we made certain choices, such as the target PPL, the front-end framework or what features to include or not.

4.1 Outline

As we have seen in Chapters 1 and 2, there is a rising awareness on the power of ML methods. PPLs, despite still being mostly unknown to data scientists, provide an interface for defining personalized probabilistic models and possess a built-in inference engine. The benefits of VPLs, when compared to a textual format, are well-known, but there is no VPL applied to a PPL, so we aim to create one.

A big requirement we wished to fulfill was, not only ease of use but also ease of installation. This means having portability across operating systems as well as being able to execute without installing dependencies. For this reason, we decided our VPE should run in the browser.

It was also important that we chose a target PPL for which there is a decent amount of programs written on, so we could thoroughly test our hypothesis. Because it was the one that best matched this criterion we chose Infer.NET to work with, a framework that can be used in CLI languages (such as C#, F# or C++). Because none of these languages can, at least at the time of writing (that might change when WebAssembly [?] gets widespread support), be directly compiled to JavaScript we will be sacrificing immediate visual feedback and build a hybrid text and visual system (a VPE) rather than a purely visual language (see Section 2.3.1 for more details). This means that our tool won't be able to run immediately the model the user defines, but will generate Infer.NET (hosted in C#) code and allow execution via a remote service.

The user can take the generated code and run it in a separate runtime (such as CLR). While not the ideal scenario, because it complicates the development and debugging cycles (in order to see each change, users must first copy the code to a file in a machine that can compile and run it), it has the advantage of seamlessly integrating with larger projects. An example would be an enterprise application written in C# that has a small component that the team who's developing it feels it would make sense to use a PPL there. They can use our VPE to develop that portion of the program, and then just place the code in the desired place. Using a purely visual programming language in the same scenario, despite having a faster development time, would then require a rewriting in a textual form.

The other alternative would be to use the tool to run the generated code in an external service, with the disadvantage that it exposes the user to both network delays and the availability and performance of such a service. This is also not ideal since it makes debugging step-by-step impossible.

4.2 Picking a target PPL

Perhaps one of the most important decisions we had to make concerns the PPL we would be offering a VPE to, since the visual programming concepts applied are highly dependent on the language's own capabilities. For instances, we cannot study how VP can be helpful in an object-oriented design if we are using a purely functional PPL such as Church which does not have that concept.

As we seen on Section 2.4.1, the PPL landscape is heterogeneous enough that we could pick a language from any paradigm or runtime we wanted to, as there are many alternatives available. So we could narrow down the scope, we decided to avoid purely functional and logical programming languages, because we wanted to build a VPE that was not an end in itself: meaning that it was not only useful to build PP but also to help our target audience eventually transition from a graphical environment to the textual form (or, at least, be able to use both if they need a more fine-grained control in a certain scenario). Even if it is a matter of debate which paradigm is more suitable to people learning how to program, we feel that because imperative and object-oriented are more common in data science (the prime examples being Python and R), they would be a better choice.

After applying this restriction, there were still many PPLs to choose from, so we decided upon the main criteria on which to make our decision: the language should have comprehensible and extensive documentation, plenty of code examples with a variety of complexities, an active and helpful community, be hosted in a mainstream programming language (as opposed to being a standalone language by itself) and it should run on the browser. The first three points are related to how well can we learn the language well enough to design a VPE for it, having no prior experience with it, in a short amount of time. Being executable on the browser would allow providing instant visual feedback of the results since the VPE would not be limited to defining a model, but it could execute it.

On a technical side the strongest candidate is VentureScript [**?**], due to the fact it is hosted in javascript (with the aforementioned advantages). Despite this advantage, it is still on Alpha and both the code examples and documentation are still scarce, even taking into consideration its predecessor's examples [**?**] and tutorials [**?**].

The three stronger candidates, after some selection, were PyMC [**?**], Figaro [**?**] and Infer.NET [**?**]. All three are hosted in popular languages (Python, Scala and C/F, respectively) and have comprehensive tutorials and documentation available [**?**][**?**][**?**].

We ended up choosing Infer.NET because it had the widest range of examples available, which is pivotal to design a VPE as complete as possible, as well as evaluate it under different scenarios. Besides its complete documentation and tutorial, its website features more than 20 code examples, as well as a list of papers that resorted to Infer.NET. Some of these papers are about PP or PPL theory, but others are about applications of it, such as "Automatic analysis and identification of verbal aggression and abusive behaviors for online social games" [**?**], which gives us confidence that Infer.NET is mature enough so we can focus in the VPE design rather than struggle with learning the language. Bottom line, it was chosen due to the abundant number of programs written on it publicly available.

4.3 Picking a front-end

After knowing which would be the language we would be targeting, there was still the need to define both the visual paradigm and the framework, if any, that we would use to develop the front-end (visual interface) of our VPE.

Regarding the paradigms, the two competing alternatives are VDP and the one offered by Blockly. While the first approach is what we typically see in similar tools, such as RapidMiner or NoFlo, the second one is a fairly new approach, unseen in tools other than those which were built using Blockly. It is not trivial to assess the suitability of either of the options because there was never a study directly comparing the merits of each of them, so we will try to make our own analysis by comparing their strengths and weaknesses while assessing what is more valuable in the context of a VPE for a PPL.

4.3.1 Blockly

Similarly to what happens with VP and VPEs in general, there are plenty of references in the literature that Blockly-based tools boost productivity [**?**] and ease the process of learning how to program [**?**].

When compared with tools that are based in a VDP approach, it has a representation that is more similar to how code is written [fE15], which can be an advantage in a tool that aims to serve as a visual interface that generates code in a textual format, such as the one we're developing. This can be seen as an advantage because, not only does it facilitate fully transitioning to a textual format as soon as the user is comfortable enough at programming, but it also makes it easier to fine-tune textually code that was developed using the VPE, because the generated code is more similar to the one a human would write when compared with a VDP representation.

Among the features we analyzed in Section 2.4.2 that deal with helping the user to achieve program correction, we saw syntax checking and type checking. Blockly addresses both situations by visually modeling the program as a puzzle where only compatible pieces fit together. This is an intuitive way to enforce correctness, since it maps to an activity the majority of people are familiarized with: assembling puzzles. This seems a better way to cope with errors rather than showing an error message at compile or run-time because often it is not even clear that there is an error, as we have seen that happens with NoFlo in Section 2.4.2.1.

Another great advantage of Blockly is that, because it runs on the browser, it is compatible across devices and operating systems while not requiring any kind of setup. If the user chooses to download Blockly, it can even work with it without an internet connection. The same applies to our VPE if we choose to build it on top of Blockly.

However, this VPE of VPEs lacks features that enable it to scale up effectively such as the ones we have seen with NoFlo or Blender Composite Nodes (Section 2.4.2). Examples of this include being able to zoom in and out in the UI, searching for blocks by their name or create group blocks to encapsulate a certain group of nodes (a concept analogous to functions in regular programming). It enables the creation of vertically and horizontally compressed blocks, similarly to Viskell, by using block composition and dropdowns rather than requiring to plug-in extra blocks (see Figure 4.1 for an example of each case).

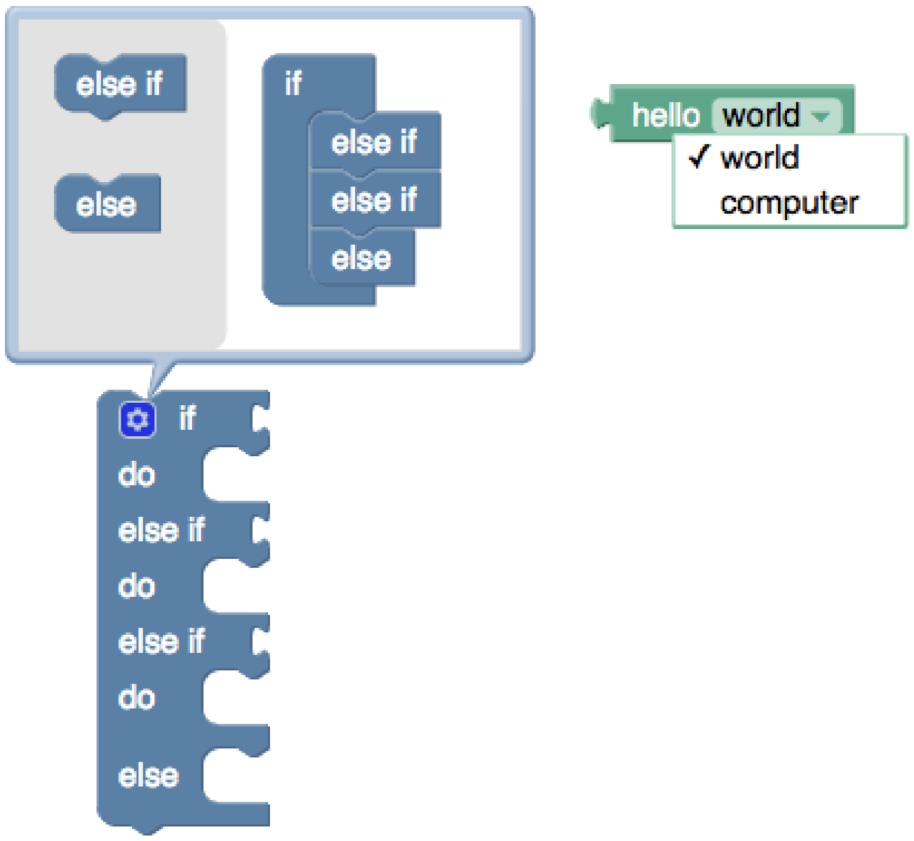


Figure 4.1: Blockly example of horizontal and vertical compression [fE15]

4.3.2 Eclipse Toolset

The Eclipse Foundation has some tools, which are accessible as plugins for the Eclipse IDE, that we thought that could serve as a foundation for the VPE we're aiming to develop.

Damos [?], a project still in the "Proposal" phase of the Eclipse Development Process, is a framework for building data-flow graphical languages and their respective code generator. It includes both graphical and textual editors, as well as support for continuous, synchronous and asynchronous computations. Even if it already includes enough blocks so it can be used by engineers and scientists as a replacement for tools like MATLAB/Simulink or LabVIEW, it was designed to be customizable and extensible with new blocks, so we could use it to build our VPE on.

Damos uses another Eclipse tool as its graphical editor: the Graphical Modeling Framework (GMF). GMF can be used to build graphical editors for multiple visual languages [?], such as UML modeling, application interfaces such as the one described in Section 2.4.3.3 or, eventually, a visual language for a PPL. GMF is often used in conjunction with the Eclipse Modeling Framework (EMF), "a modeling framework and code generation facility for building tools and other applications based on a structured data model" [?].

In spite of their advantages, all of these have two common problems. First, they require users to download both Eclipse and the used plugins and their dependencies which not only constitutes an entry barrier but hinders collaboration, because it makes it difficult to share the model among colleagues. And secondly, they were built to generate code to specific target languages (C for Damos and Java for EMF), neither of which has a PPL implementation with a strong community (Probabilistic-C [?] serves as a proof of concept, and we already explained in Section 4.2 why we discarded Figaro). Even if we could eventually use the generated C or Java as an intermediary representation, we believe the extra burden (both in development time and in runtime performance) of having an extra transformation does not justify for the added benefits when compared with other front-ends.

4.3.3 Custom implementation

At this point the strongest candidate for a front-end was Blockly, but it would also be possible to fully develop a new front-end, instead of using a framework such as Blockly or the tools from the Eclipse toolset. The purpose of doing so would be overcoming Blockly's limitations that we mentioned earlier in Section 4.3.1. In order to keep Blockly's portability, we would have an implementation targeting the browser, where the visual elements would be represented by DOM tree elements and drawn using the browser's render engine, just like a regular HTML page; for performance reasons, we could use the HTML *canvas* element [?].

The reason why we won't be choosing this option is that we believe that this flexibility of implementing extra features (that essentially have to do with navigation, block filtering and block selection) are not worth the risk of having a VPE that does not feel as fluid and as natural as Blockly or GMF implementation would, since all those aforementioned features are secondary when compared to the look and feel of the core VPE (by core we mean the workspace where we drag the blocks, edit their inline properties and connect them).

Examples of such core usability features, besides having a layout that has already been tested (proven to deliver the benefits of VP) that Blockly already provides include magnetic connections (you don't have to drag and drop a block to a strictly defined area, doing so near a valid connection highlights that connection and makes it possible to drop it there) and block mutations (a block may change when you connect certain blocks to it, this allows for horizontal and vertical compression).

Using a third-party framework also has the advantage of automatically getting new features as soon as that framework is updated, such as the zoom in and out that Blockly is planning to deliver [fE15].

4.4 Architecture

The VPE's architecture is fairly straighforward, as you can confirm in Figure 4.2. We have leveraged Blockly's already existing blocks for common programming language constructs, such as as conditional clauses or loops, and added new capabilities that we believe to be fundamental to using a VPE for designing probabilistic models (we'll be discussing them in detail in Section 4.5).

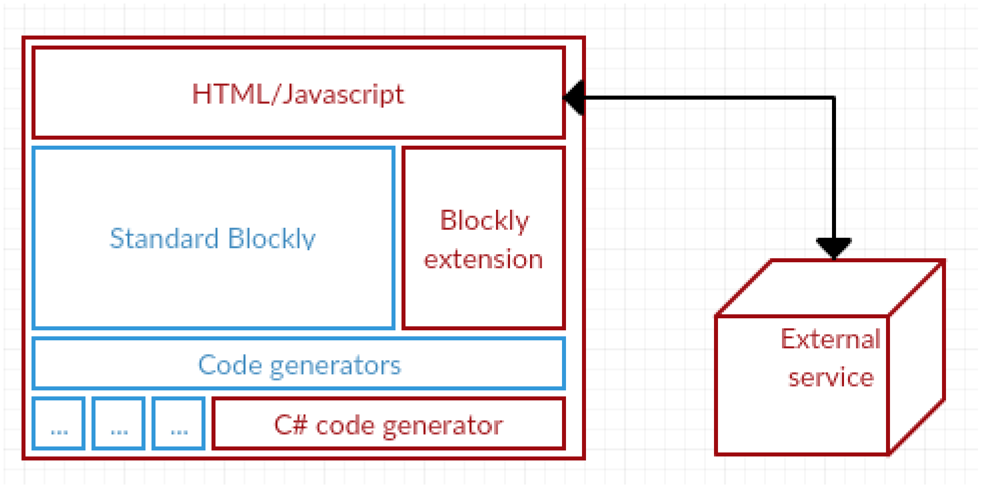


Figure 4.2: High-level overview of the VPE's architecture

4.5 Development steps

The purpose of this section is to enumerate the difficulties encountered when developing a VPE that could support the design of programs similar in functionality to the ones the reader can find in Infer.NET's tutorials [?] and the solution we adopted to overcome them.

4.5.1 Generating basic C

Since Infer.NET is hosted in C but Blockly does not include a generator for the language, the first thing to do was to be able to provide code generation for Blockly's default blocks, that provide a language's most basic constructs such as variables, lists and control structures.

We decided to adopt an open-source implementation to solve this issue [?] but since it was not compatible with Blockly's versions since January 2016, it was necessary to work on making it compatible, so we could have the latest features available, such as zooming in and out, which we had previously identified as a major feature required in a VPE to be able to scale up programs visually.

This work lead to the pull request #8 to that open-source project [?].

4.5.2 Mandatory declarations

Similarly to Java and C/C++, in order to run a program, you must provide a method as an entry point, where the control flow will start and end. However, a textual representation generated by Blockly does not contemplate this possibility, so we had to figure out how to deal with this issue. Figure 4.3 shows a simple program, and this is its generated code:

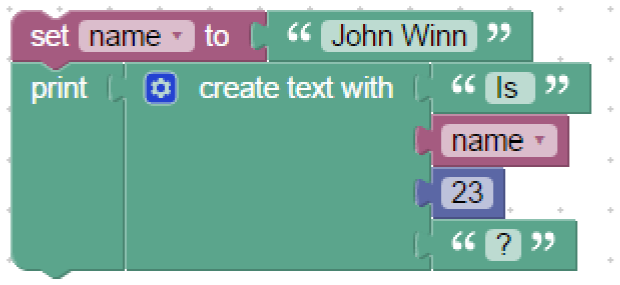


Figure 4.3: Blockly simple example

```
dynamic name;


name = "John Winn";
Console.WriteLine(String.Concat("Is ", name, 23, "?"));
```

If you try to run this code, you will get a compile-time error as expected.

Another issue that arises when representing Infer.NET programs in Blockly is how to specify inference engine settings, such as the algorithm is should use, if it parallelizes for-loops, if it re-allocates internal messages between inference runs, etc. Unlike other kinds of statements, such as loops, variable declarations or prints, these settings are usually only set once per program and remain unaltered.

Having both these situations in mind, a possible solution is to use a singleton block (one that must be present in every representation and can't be deleted), where we aggregate both these concerns and which contains the remainder of the program. Such an example can be seen in Figure 4.4, which generates the following code:

```
using System;
using System.Collections.Generic;
```

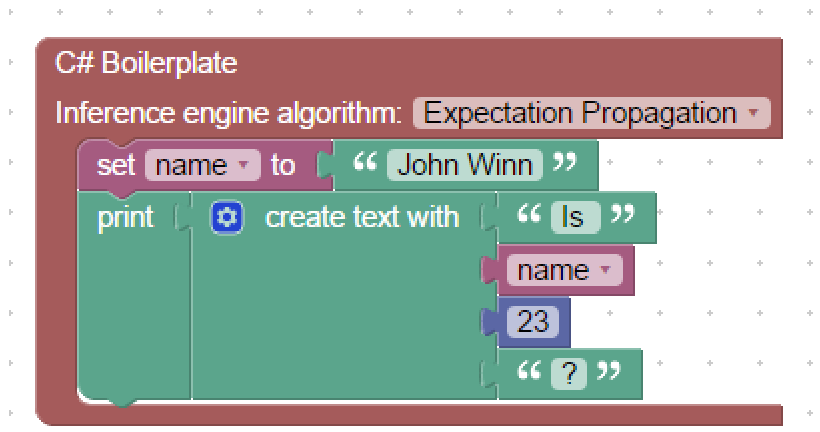

C# Boilerplate
Inference engine algorithm: Expectation Propagation
set name to " John Winn "
print create text with " Is "
name
23
" ? "

Figure 4.4: Blockly simple example with C boilerplate

```
using System.Text;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer;

namespace InferBlockly
{
  public class InferBlockly
  {
    public void Run()
    {
      InferenceEngine engine = new InferenceEngine();
      engine.Algorithm = new ExpectationPropagation();
      dynamic name;


      name = "John Winn";
      Console.WriteLine(String.Concat("Is ", name, 23, "?"));
    }
  }
}
```

Despite losing flexibility, since it is impossible to change the settings throughout the program (even if that is valid in Infer.NET), we believe that this increased simplicity is more intuitive and therefore beneficial to an inexperienced programmer than allowing the settings to be changed at any point in time.

4.5.3 Variable type checking

Every PPL has the concept of random variables, which are the building block of any PP. Some examples in Infer.NET are listed below.

```
        Variable<bool> firstCoin = Variable.Bernoulli(0.5);
        Variable<double> x = Variable.GaussianFromMeanAndVariance(0, 1);
        Variable<double> threshold = Variable.New<double>();
        Variable<double> precision = Variable.GammaFromShapeAndScale(1, 1);
        VariableArray<double> x = Variable.Array<double>(dataRange);
        VariableArray<bool> y = Variable.Observed(willBuy);
```

Because there are so many different types (we are showing 4, but there are more), we believe our VPE should help the user avoid mistakes related with using wrong types, such as trying to apply logical operators to numeric random variables or greater than comparisons between boolean variables.

When choosing how to represent variables we had to take into account that Blockly was conceived to target dynamic languages, as it can be seen by the languages it targets: JavaScript, Python, PHP and Dart (which supports static type-checking, but it's not the default mode [**?**]). The negative consequence of this is that it's not ready to perform type checking on variables, meaning that a user can produce invalid programs when using variables (as seen in Figure 4.5, we have drawn a program that will produce a runtime exception since we can't evaluate the square root of a text string), although the same doesn't happen when directly connecting the blocks, since we can't connect text input blocks to the right connector of the square root block.

There have been discussions among the open-source projects that rely on Blockly on how to incorporate type checking for variables (such as a VPE for Arduino [**?**] or one for Kiwi.js [**?**], an HTML5 game framework) as well as in Blockly's discussion forum [**?**], but none reached a final proposal, let along a working implementation.

Therefore, we have decided to extend Blockly in order to be able to support different types of variables with a similar type check to the one used in other kinds of blocks, meaning that if they don't type check, the pieces don't fit and the user can't connect them. This was done by adding an extra field to FieldVariable (the class Blockly class that represents variables) that identifies its type and modifying the behavior of each component that deals with variables to take that into account [1].

This way, variables from different types are kept isolated, so that if the user declares a boolean variable named *bool*, it won't appear in numeric variable's dropdowns or menus. This segregation between different variable types, besides allowing to have the aforementioned type checking at the block level, it also allows to have different colors and labels in the blocks for different types of variables (making it clearer to the user they should be used differently) and to declare explicit C types in the generated code, rather than using the dynamic keyword to avoid static type-checking [**?**]. The advantage is that the generated code is closer to the one that would be written by a programmer, so it eases the transition from a VPE to a textual format, since both representations are similar.

[1]These changes can be found in github.com/gcandal/blockly/commit/8f5fa0

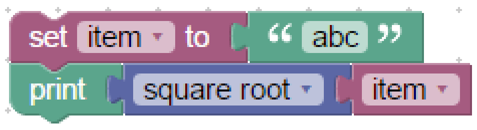


Figure 4.5: Blockly example of representable invalid program

4.5.4 Serialization

It is important in a VPE to be able to save and load programs, so the user can not only work on a given program at different points in time but also share it with other users. Blockly makes it possible to do so by converting a graphical representation to XML, which can later be imported.

We have extended Blockly so this functionality could be easily accessible in the GUI, and the XML can be saved and loaded into the computer that's running the VPE.

4.5.5 Remote execution

A problem that arises when building a VPE that runs in a browser is that it can't execute code from any language other than Javascript. Since we are using Infer.NET we also face this problem.

A basic approach to this issue is to assume the user can't run the program directly on the VPE and that he must copy the code to a local file so that he can compile and run it. The downside with this approach is obvious: it places a great burden on the user and negatively affects the development process, since it increases the time between designing and getting the results.

Our solution to this problem was to develop a web service that provides a single HTTP endpoint. This endpoint receives the textual representation of the code as input and return's that program's output. This way even if we still don't have instant visual feedback because of the network round-trip time, that delay is often negligenciable when compared to the time needed for the program to run, particularly for larger programs.

Chapter 5

Practical examples

All the examples presented here are drawn from Infer.NET's tutorial [?]

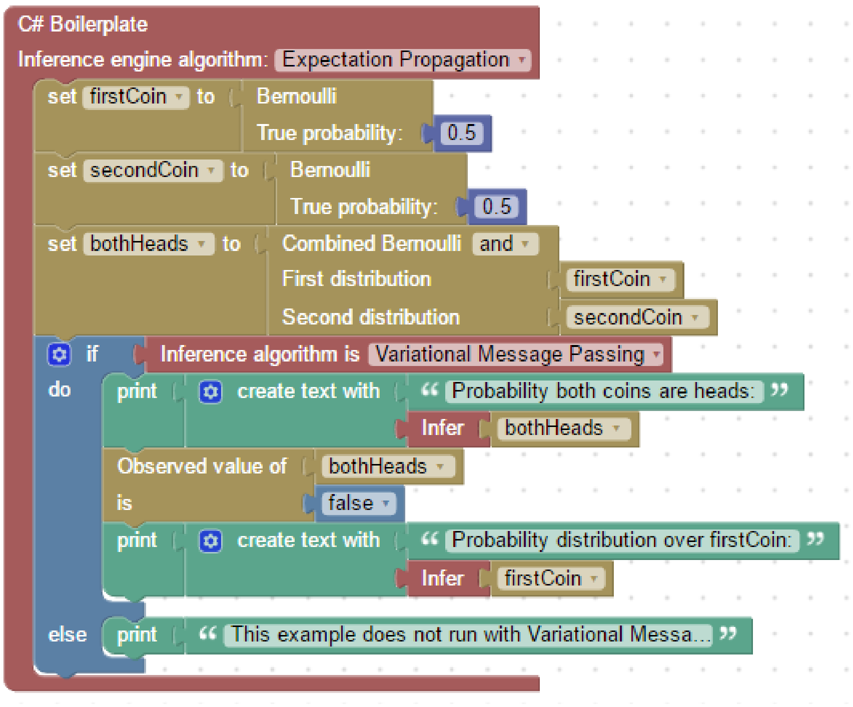
5.1 Two coins

This example is a simple one, where we want first to determine the probability of two coins being heads, and then say that the two weren't heads and ask for the probability of the first one being heads. We can see the resulting representation in Figure 5.1, which generates the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer;

namespace Examples
{
  public class FirstExample
  {
    public void Run()
    {
      InferenceEngine engine = new InferenceEngine();
      engine.Algorithm = new ExpectationPropagation();
      Variable<bool> firstCoin;
      Variable<bool> secondCoin;
      Variable<bool> bothHeads;

      firstCoin = Variable.Bernoulli(0.5);
      secondCoin = Variable.Bernoulli(0.5);
      bothHeads = (firstCoin & secondCoin);

      if (!(ie.Algorithm is VariationalMessagePassing)) {
        Console.WriteLine(String.concat("Probability both coins are heads: ", ie.
            Infer(bothHeads));
```

```
          bothHeads.ObservedValue = false;
          Console.WriteLine(String.concat("Probability distribution over firstCoin: ", ie.Infer(firstCoin)));
        } else {
          Console.WriteLine("This example does not run with Variational Message Passing");
        }
    }
  }
```

In this fairly simple example, we represent some of the most important semantics of a PPL: declaring random variables, the dependencies between them and plugging in observed values. In the next examples, we will be adding more constructs on top of these basic ones, so the reader can be convinced that this representation is easily extensible.



```
C# Boilerplate
Inference engine algorithm: Expectation Propagation
  set firstCoin to  Bernoulli
                    True probability: 0.5
  set secondCoin to  Bernoulli
                     True probability: 0.5
  set bothHeads to  Combined Bernoulli and
                    First distribution   firstCoin
                    Second distribution  secondCoin
  if  Inference algorithm is Variational Message Passing
  do  print  create text with  " Probability both coins are heads: "
                               Infer bothHeads
      Observed value of  bothHeads
      is  false
      print  create text with  " Probability distribution over firstCoin: "
                               Infer firstCoin
  else  print  " This example does not run with Variational Messa... "
```

Figure 5.1: First example of a PP with our Blockly-powered VPE.

5.2 Truncated Gaussian

In this example, we define a Gaussian distribution and then query for the result of that distribution when we restrict its values to be greater than a certain threshold. The generated code is below, while the graphical representation is in 5.2.

```
using System;
using System.Collections.Generic;
using System.Text;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer;
using MicrosoftResearch.Infer.Distributions;
using MicrosoftResearch.Infer.Maths;

namespace BlocklyInfer
{
  public class BlocklyInfer
  {
    public static void Main()
    {
      InferenceEngine engine = new InferenceEngine();
      engine.Algorithm = new ExpectationPropagation();
      dynamic thresh;
      Variable<double> x;


      for (thresh = 0; thresh <= 1; thresh += 0.1) {
          x = (Variable.GaussianFromMeanAndVariance(0, 1));
          Variable.ConstrainTrue((x > thresh));
          if (engine.Algorithm is ExpectationPropagation) {
            Console.WriteLine(String.Concat("Dist over x given thresh of ", thresh,
                 "=", (engine.Infer(x))));
          } else {
            Console.WriteLine("This example only runs with Expectation Propagation"
                );
          }

        }
    }
  }
}
```

There is yet another version similar to the first one, but where we use a random variable to represent a number rather than a normal number, which leads to a more efficient inference process. This shows how Blockly's type system is good enough to be strict but also flexible; in this case, the observedValue function can either receive numbers or random numbers as part of the input. The code is shown below and is generated from the model of Figure 5.3.

```
using System;
using System.Collections.Generic;
using System.Text;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer;
using MicrosoftResearch.Infer.Distributions;
```

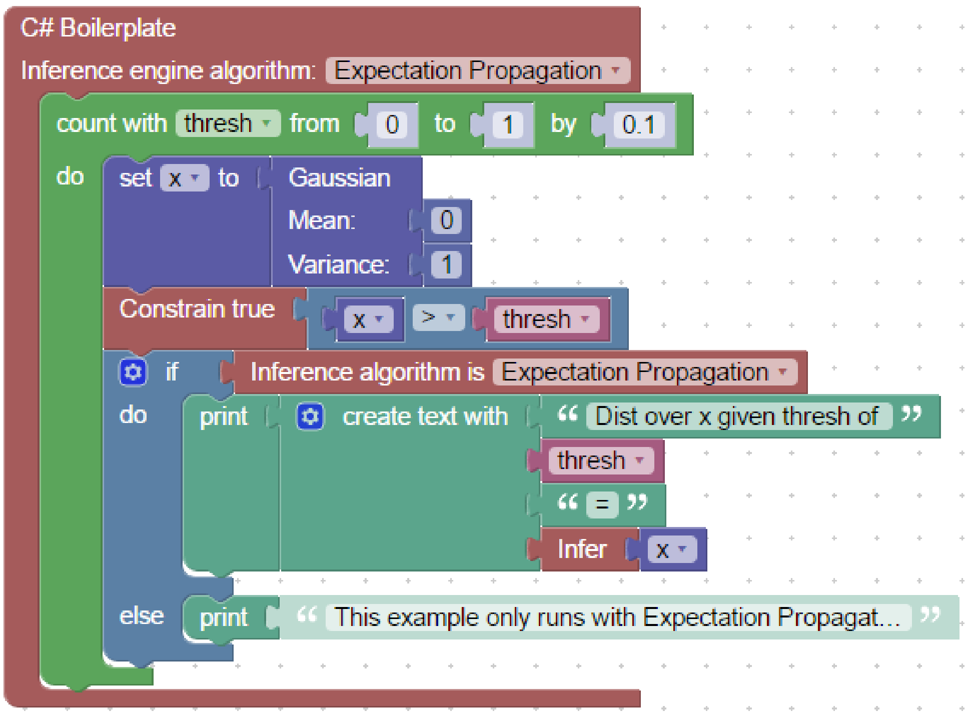
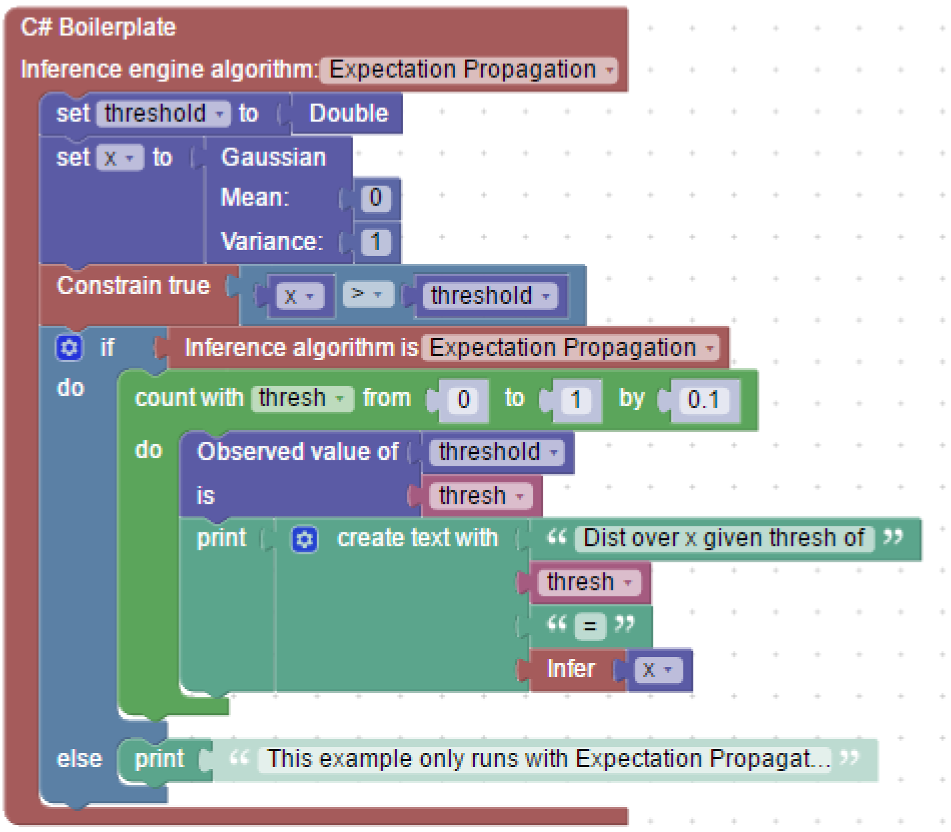

C# Boilerplate
Inference engine algorithm: Expectation Propagation
count with thresh from 0 to 1 by 0.1
do set x to Gaussian Mean: 0 Variance: 1
Constrain true x > thresh
if Inference algorithm is Expectation Propagation
do print create text with "Dist over x given thresh of " thresh "=" Infer x
else print "This example only runs with Expectation Propagat..."

Figure 5.2: Truncated Gaussian PP

```
using MicrosoftResearch.Infer.Maths;

namespace BlocklyInfer
{
  public class BlocklyInfer
  {
    public static void Main()
    {
      InferenceEngine engine = new InferenceEngine();
      engine.Algorithm = new ExpectationPropagation();
      dynamic thresh;
      Variable<double> threshold;
      Variable<double> x;


      threshold = (Variable.New<double>());
        x = (Variable.GaussianFromMeanAndVariance(0, 1));
        Variable.ConstrainTrue((x > threshold));
        if (engine.Algorithm is ExpectationPropagation) {
          for (thresh = 0; thresh <= 1; thresh += 0.1) {
            threshold.ObservedValue = thresh;
            Console.WriteLine(String.Concat("Dist over x given thresh of ", thresh,
                 "=", (engine.Infer(x))));
          }
        } else {
                Console.WriteLine("This example only runs with Expectation Propagation");
            }
        }
    }
}
```



```
C# Boilerplate
Inference engine algorithm: Expectation Propagation
  set threshold to Double
  set x to Gaussian
             Mean: 0
             Variance: 1
  Constrain true  x > threshold
  if  Inference algorithm is Expectation Propagation
  do  count with thresh from 0 to 1 by 0.1
      do  Observed value of threshold
          is thresh
          print  create text with  " Dist over x given thresh of "
                                   thresh
                                   " = "
                                   Infer x
  else  print " This example only runs with Expectation Propagat... "
```

Figure 5.3: More efficient version of Truncated Gaussian PP

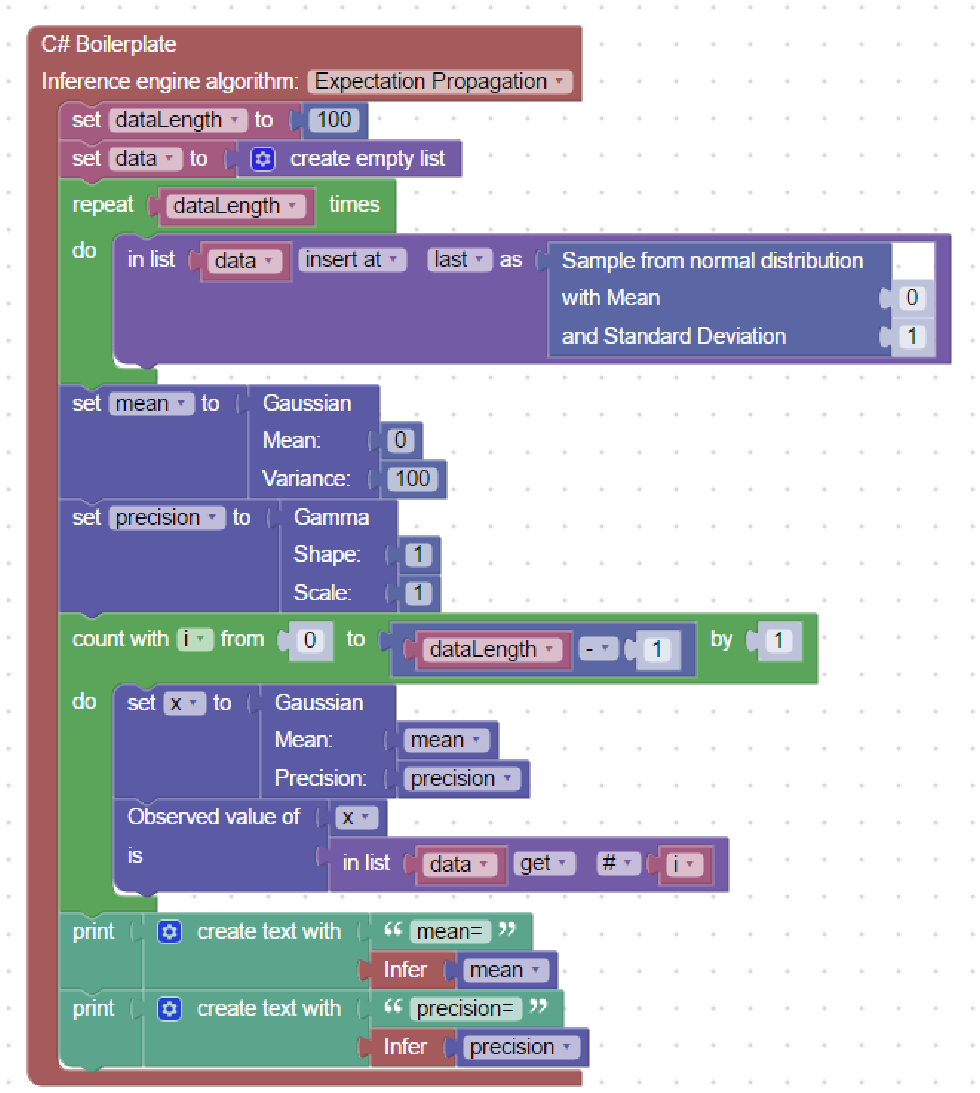
These examples are an extension of the first one with added loop constructs, the possibility of defining a single random number and the possibility to constrain numeric distributions.

5.3 Learning a Gaussian

In this model, we're trying to infer the mean and precision of a given dataset (generated randomly). The most distinguishing aspect of this example is that it deals with lists. The resulting code from the model of Figure 5.4 is shown below.

```
using System;
using System.Collections.Generic;
using System.Text;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer;
using MicrosoftResearch.Infer.Distributions;
using MicrosoftResearch.Infer.Maths;

namespace BlocklyInfer
{
  public class BlocklyInfer
  {
    public static void Main()
    {
      InferenceEngine engine = new InferenceEngine();
      engine.Algorithm = new ExpectationPropagation();
      dynamic dataLength;
      dynamic data;
      dynamic i;
      Variable<double> mean;
      Variable<double> precision;
      Variable<double> x;


      dataLength = 100;
        data = new List<dynamic> {};
        for (var count = 0; count < dataLength; count++) {
          data.Add((Rand.Normal(0, 1)));
        }
        mean = (Variable.GaussianFromMeanAndVariance(0, 100));
        precision = (Variable.GammaFromShapeAndScale(1, 1));
        var i_end = dataLength - 1;
        var i_inc = 1;
        if (0 > i_end) {
          i_inc = -i_inc;
        }
        for (i = 0;
             i_inc >= 0 ? i <= i_end : i >= i_end;
             i += i_inc) {
          x = (Variable.GaussianFromMeanAndPrecision(mean, precision));
          x.ObservedValue = (data[i]);
        }
        Console.WriteLine(String.Concat("mean=", engine.Infer(mean)));
        Console.WriteLine(String.Concat("precision=", engine.Infer(precision)));
    }
  }
}
```

This is the longest generated code of all the examples, and the reason why the second loop looks so different from the one that would be written by a programmer is a consequence of using a variable (dataLength) in a loop where we want to re-use the variable used to count the loop's iterations. If you look at the operations before the second loop, you can notice that they exist because they give the flexibility to the user of looping in the reverse order. Aside from this, the example is yet another natural extension of the first one, where we added more types of distributions.



```
C# Boilerplate
Inference engine algorithm: Expectation Propagation
  set dataLength to 100
  set data to create empty list
  repeat dataLength times
  do  in list data insert at last as Sample from normal distribution
                                     with Mean 0
                                     and Standard Deviation 1
  set mean to Gaussian
              Mean: 0
              Variance: 100
  set precision to Gamma
                   Shape: 1
                   Scale: 1
  count with i from 0 to dataLength - 1 by 1
  do  set x to Gaussian
                Mean: mean
                Precision: precision
      Observed value of x
      is in list data get # i
  print create text with " mean= "
                         Infer mean
  print create text with " precision= "
                         Infer precision
```

Figure 5.4: Learning a Gaussian distribution

5.4 Conclusions

The generated code is fairly similar to one that a human would write, with the exceptions of string concatenation (in this case it would be more natural to join the text with the inference result via the plus operator rather than using String.concat), variables being declared and set in different lines and some types of loops (which have been justified in the last example).

It is our belief that the graphical representation is clearer than its textual counterpart, mainly because parameters are named (it is clear that 0.5 is the probability of true when defining a

Bernoulli distributions) and due to the use of colors: red for statements related with the inference engine, dark yellow for boolean distributions, green for text and I/O, blue for booleans and control structures and purple for numbers.

Chapter 6

Conclusions

The generated code is fairly similar to one that a human would write, with the exceptions of string concatenation (in this case it would be more natural to join the text with the inference result via the plus operator rather than using String.concat), variables being declared and set in different lines and some types of loops (which have been justified in the last example).

It is our belief that the graphical representation is clearer than its textual counterpart, mainly because parameters are named (it is clear that 0.5 is the probability of true when defining a Bernoulli distributions) and due to the use of colors: red for statements related with the inference engine, dark yellow for boolean distributions, green for text and I/O, blue for booleans and control structures and purple for numbers.

The main reason to use a VPE rather than the original textual representation is because of the safety net it provides and which prevents the user from making either syntax or type errors.

For these reasons, we believe this kind of visual programming environment can be useful to help data scientists who are inexperienced programmers take the first steps in using a PPL.

However, if we don't look at the VPE as an educational tool but rather as something that won't be transitional, we think that productivity-wise it falls short when compared to tools such as RapidMiner, Excel or even textual representations. The main rationale behind this is that, because we chose to adopt a Blockly representation, the resulting VPE is not a completely visual language since it violates concreteness and does not fully encapsulate textual programming concepts (such as variable declarations). If there was someone considering to develop a VPL to be used as a permanent substitute to the textual form of PPLs, we believe he should explore a dataflow representation, which is at an higher-level of abstraction than Blockly and has been used in successful applications both in industry and academia.

6.1 Limitations

In this section, we will describe some limitations that our chosen toolset (using Blockly to represent Infer.NET C) imposes in functionality.

6.1.1 Inverse compilation

By inverse compilation we mean the act of converting a program in a textual form to its Blockly representation. This could be useful because it would help users easily switch between representations, similarly to Window Builder (as discussed in Section 2.4.3.3) and unlike what happens with our current approach. With our VPE, if you textually make a change in the program by directly writing in the text-area where it appears, and since that won't be reflected in the graphical representation, every time the code is re-generated (which happens frequently and unexpectedly, in situations such as opening and closing menu tabs) your changes will be overridden.

Even if parsing C and converting it to an intermediate representation would be simple (Microsoft provides an EBNF grammar for the language [?]), there are a number of questions that should be explored before implementing this feature. For instances, our graphical representation guarantees type safety and a valid program so it would not make sense to allow specifying an invalid program that would be translated into a graphical form, it would break one of the most important guarantees, the one that makes it suitable for beginners in programming. In order to ensure a valid textual program, we would need to be able to either mimic the compiler's verifications or run the compiler itself; the first option is a big challenge while the second still requires further investigation on how to handle users' errors.

6.1.2 Instant visual results

As we discussed before in several parts of this work, a big tradeoff we made was to sacrifice the flexibility of using any language needed in favor of the portability of a VPE that runs in the browser, which (as of this moment) only runs JavaScript.

Perhaps the greatest limitation of this approach is not being able to run Infer.NET code, which makes it impossible not to violate the VP principle of instant visual results. The development cycle is greatly affected by forcing the user to copy the text into a location where a compiler can access and run it. A workaround for this is to adopt a client-server architecture where our server would run the code and send the results back to the user. Even if that creates a need for internet connection, which currently does not exist, we believe it to be a worthwhile tradeoff, and so we have chosen to implement it.

The downside of this approach is that we have to wait for the round-trip time of the program to be sent to the server plus the compilation and execution times. During this process, the user does not get any feedback; this could be minimized by providing push notifications to the client and streaming the program's output, rather than waiting for the execution to finish to get a response.

6.1.3 Object-oriented programming

You may notice that, even while we're targeting an object-oriented programming language, our VPE has no support for representing classes. It also does not support file management (splitting the program into several files) or namespaces.

The reason for this is that our target audience is not concerned with those features. As discussed in Section 2.3 VP is best suited for small domains, and our VPE is meant to help data scientists and statisticians with little or no programming experience start getting familiar with PP through a PPL, and not provide a fully featured IDE.

Incorporating this kind of features, although feasible, would significantly make the VPE more complex both in terms of cluttering the interface and the user experience by providing mostly unnecessary options, thus having little added benefit to the majority of users.

6.2 Future Work

During the work of this thesis, we have certainly identified areas where there could be improvements. The first one is an addition to the validation process, while the second concerns a feature that could be added; we describe both below.

Besides these two, the concepts we explore in this text could be further improved, and even extended, by continuing to perform the example modeling loop. Even if we feel that we have tackled the most significant challenges, and provided a decent number of examples, there is always still more work to be done in this area.

Using examples from a different PPL could be interesting, even if the semantics are usually very similar, so we could access the generality of the concepts and transformation patterns we have been studying.

Another feature that would highly value the VP tools for PP in respect to their textual counterparts would be to include more visual feedback. This includes not only graphics of the resulting distributions, but could even show progress while the model is running, so the user could debug their program.

6.2.1 An empirical study

As discussed in Chapter 4, an interesting approach to further validate the hypothesis that a change in paradigm from traditional procedural or object-oriented PP to visual PP makes a certain class of users more productive would be to perform an empirical study whose participants would belong to the target audience we defined in Section 3.2.

The way the study would work would be by compare how fast a user can define a model for a given set of problems when using a PPL in a regular way or through our graphical interface. We'd then count the number of syntax and type errors done with each representation. By selecting users who never used the chosen PPL, we could not only measure execution speed but learning time.

It would also be valuable to assess, not only how fast can someone develop with either of the alternatives, but also the quality of the output. That could be done in two steps: starting by verifying if the program correctly models the problem and then asking the participants in the study if they believe the model they have developed graphically is easier to understand than its textual counterpart (and vice-versa). Although subjective, we believe getting the participants' opinion regarding the output quality could provide valuable insights in order to understand if VP can really enhance a user's experience when using a PPL. Another method we could use to help us make an assessment of the validity of the hypothesis would be asking participants questions regarding usability. Even if it may seem redundant, since we would already have the time measurements, it is a way of identifying strengths and weaknesses of the visual representation.

6.2.2 Function blocks

A feature that was left out during development but could have a significant impact in the VPE's usability, mainly during the development of large models (scaling up) would be to provide the user with the capability to create his own blocks, made up of smaller blocks. This is analogous to functions in regular programming and has been successfully applied to widely used VPE, as we have seen in 2.

References

[AA92] K S R Anjaneyulu and John R Anderson. The Advantages of Data Flow Diagrams for Beginning Programming. 1992.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, second edition, 2010.

[Ama13] Xavier Amatriain. Beyond data: from user information to business value through personalized recommendations and consumer science. In *Proceedings of the 22nd ACM international conference on Conference on information &#38; knowledge management*, CIKM '13, pages 2199–2200, New York, NY, USA, 2013. ACM.

[BAF$^+$15] Andrei Broder, Lada Adamic, Michael Franklin, Maarten de Rijke, Eric Xing, and Kai Yu. Big data: New paradigm or "sound and fury, signifying nothing"? In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 5–6, New York, NY, USA, 2015. ACM.

[Ber] Henri Bergiu. Noflo. Available at http://noflojs.org/, accessed in 7th Feb 2016.

[Boe16] Arjan Boeijink. Experimenting with an interactive visual functional programming environment. 2016.

[Bri16] Encyclopædia Britannica. machine learning. Available at http://www.britannica.com/technology/machine-learning, accessed in 3rd Jan 2016, 2016.

[Cra04] Ciprian Crainiceanu. A short introduction to winbugs. 2004.

[Das02] M. Dastani. The role of visual perception in data visualization. *Journal of Visual Languages and Computing*, 13(6):601–622, 2002.

[DL] David Duvenaud and James Lloyd. Introduction to probabilistic programming.

[DL13] David Duvenaud and James Lloyd. Talk on 'introduction to probabilistic programming'. 2013.

[Dow12] Allen B. Downey. *Think Bayes*. Green Tea Press, 2012.

[fE15] Google for Education. Blockly is a library for building visual programming editors. Available at https://developers.google.com/blockly/, accessed in 2nd Feb 2016, 2015.

[Fou16] Eclipse Foundation. Windowbuilder - is a powerful and easy to use bi-directional java gui designer. Available at https://eclipse.org/windowbuilder/, accessed in 2nd Feb 2016, 2016.

[FR12] Cameron E. Freer and Daniel M. Roy. Computable de Finetti measures. *Annals of Pure and Applied Logic*, 163(5):530–546, 2012.

[Geo16] Pétia Georgieva. Machine learning for big data processing. 2016.

[Gor14] Sriram K. Gordon, Andrew D. and Henzinger, Thomas A. and Nori, Aditya V. and Rajamani. Probabilistic Programming. *International Conference on Software Engineering (ICSE Future of Software Engineering)*, pages 267–, 2014.

[Gri13] Olivier Grisel. Keynote on 'trends in machine learning and the scipy community'. Available at https://youtu.be/S6IbD86Dbvc, accessed in 3rd Jan 2016, 2013.

[HR08] Mark Hall and Peter Reutemann. Weka knowledgeflow tutorial for version 3-5-8. Available at http://software.ucv.ro/ eganea/AIR/KnowledgeFlowTutorial-3-5-8.pdf, accessed in 7th Feb 2016, 2008.

[Jag13] Suresh Jagannathan. Probabilistic programming for advancing machine learning (ppaml). Available at http://www.darpa.mil/program/probabilistic-programming-for-advancing-machine-Learning, accessed in 3rd Jan 2016, 2013.

[JW96] Michael I Jordan and Yair Weiss. Probabilistic inference in graphical models. *Lauritzen, S. L*, 16(510):140–152, 1996.

[KT15] Tejas D Kulkarni and Joshua B Tenenbaum. Picture : A Probabilistic Programming Language for Scene Perception. *Cvpr*, 2015.

[Las12] Daniel Lassiter. Probabilistic reasoning and statistical inference : An introduction (for linguists and philosophers). pages 1–51, 2012.

[MWGK12] Tom Minka, John Winn, John Guiver, and David Knowles. Infer .net 2.5. *Microsoft Research Cambridge*, 2012.

[PB99] Marian Petre and Alan F Blackwell. Mental imagery in program design and visual programming. *Int. J. Hum.-Comput. Stud.*, 51(1):7–30, 1999.

[Pia15] Gregory Piatetsky. 16th annual kdnuggets software poll. Available at http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html, accessed in 4th Jan 2016, 2015.

[Pre03] Probabilistic Programming. *Handbooks in Operations Research and Management Science*, 10(C):267–351, 2003.

[Rap] RapidMiner. Rapidminer studio manual. Available at http://docs.rapidminer.com/downloads/RapidMiner-v6-user-manual.pdf, accessed in 7th Feb 2016.

[Sch08] Rob Schapire. Lecture notes in 'cos 511: Theoretical machine learning'. Available at http://www.cs.princeton.edu/courses/archive/spr08/cos511/scribe$_{n}otes/0204.pdf, accessedin3rdJan2016,$

[SP93] P Szolovits and S G Pauker. Categorical and probabilistic reasoning in medicine revisited. *Artificial Intelligence*, 1993.

[Sta15] Conn Stamford. Gartner's 2015 hype cycle for emerging technologies identifies the computing innovations that organizations should monitor. Available at http://www.gartner.com/newsroom/id/3114217, accessed in 3rd Jan 2016, 2015.

[Wag12] Kiri Wagstaff. Machine Learning that Matters. *Proceedings of the 29th International Conference on Machine Learning*, pages 529–536, 2012.

[Wan02] DeLiang Wang. Ohio state cis 730, lecture notes on 'probabilistic reasoning - belief networks'. 2002.

[WBJ05] John Winn, Cm Bishop, and T Jaakkola. Variational Message Passing. *Journal of Machine Learning Research*, 6:661–694, 2005.

[Wik] Blender Wiki. Composite nodes. Available at http://wiki.blender.org/index.php/Doc:2.4/Manual/Composite$_{Nodes}$,