PRISM Revisited: Declarative implementation of a probabilistic programming language using multi-prompt delimited control and CLP

Samer Abdallah

Jukedeck Ltd.

PLP Workshop, Orleans, France,
September 7, 2017

Outline

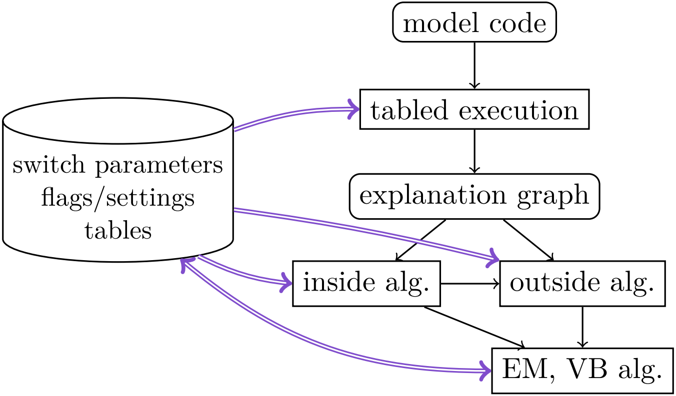
Outline

PRISM: PRogramming In Statistical Models

Early versions: Sato (1995) and Sato and Kameya (1997).

- Prolog-like syntax augmented with 'switches' representing parameterised discrete distrubutions; and *msw*/2 for probabilistic choice.
- Subsumes Markov models, (discrete) HMMs, pCFGs, graphical models.
- Sampling execution.
- Tabled execution (Sato and Kameya, 2000) to get explanation graph (Earley deduction, generalises efficient parsers).
- Efficient algorithms on the graph: Viterbi, inside, inside-outside, EM for parameter learning.
- Further elaborations: variational Bayes (Kurihara and Sato, 2006), MCMC (Sato, 2011).

PRISM


model code
tabled execution
explanation graph
switch parameters
flags/settings
tables
inside alg.
outside alg.
EM, VB alg.

Explanation graphs

Example model: *dice*(N,Z) means N throws of tetrahedral die sum to Z.

values(*die*,[1,2,3,4]).

dice(0,0).
dice(N,Z) $\leftarrow$
  msw(*die*, X),
  $N > 0$, M is $N-1$, *dice*(M,Y),
  Z is $X+Y$.

Enter a top goal *dice*(3,4).

Explanation graphs

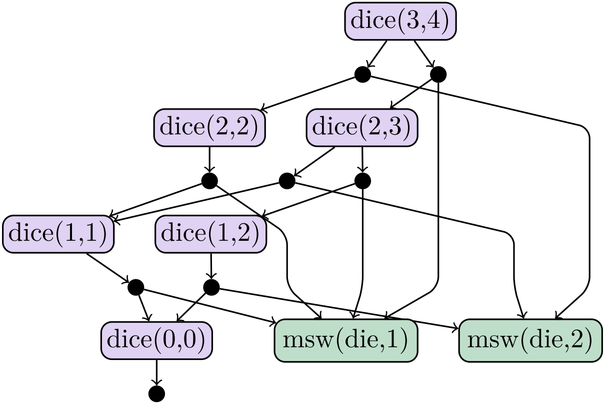
PRISM represents the explanation graph textually as:

```
dice(3,4) <=> dice(2,3) & msw(die,1)
            v dice(2,2) & msw(die,2)
dice(2,3) <=> dice(1,2) & msw(die,1)
            v dice(1,1) & msw(die,2)
dice(1,2) <=> dice(0,0) & msw(die,2)
dice(2,2) <=> dice(1,1) & msw(die,1)
dice(1,1) <=> dice(0,0) & msw(die,1)
dice(0,0)
```

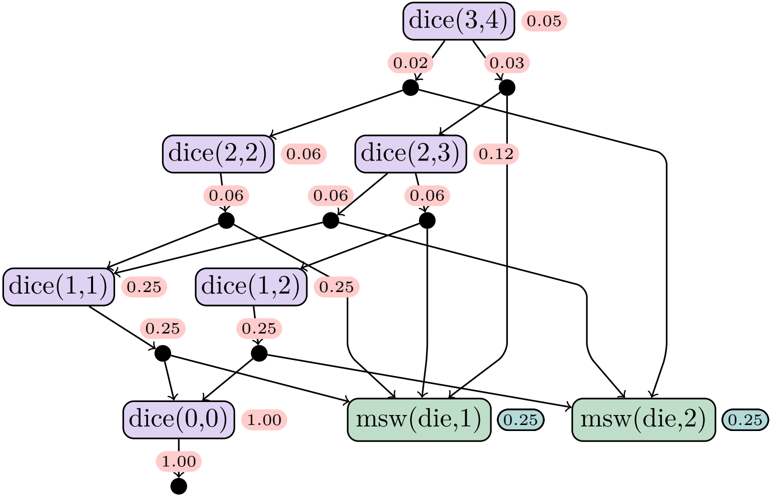
Each subgoal is logically equivalent (`<=>`) to a disjunction (`v`) of conjunctions (`&`). In the rest, we will refer to each conjunct as a *factor* and a conjunction of factors as an *explanation*.

Explanation graphs

Can think of as either a heterogenous graph or a *hypergraph* (Klein and Manning, 2004) where black circles are hyperedges.



Inside algorithm



Implementation size

Comparison between *ccprism* and PRISM version 2.1 (closest in feature set to *ccprism*). Some code implementing general purpose services has been excluded in order to compare like with like.

	Prolog	C	Total
PRISM	6,463	8,010	14,473
ccprism	673	0	673

Although comparison is far from perfect (both implementations include some features not found in the other) PRISM contains roughly 20 times as much code.

Outline

Continuations

Why consider continuations?

- Delimited continuations are really powerful: can implement many kinds of computational effects, including state, nondeterminism, all monads (Filinski, 1999) and tabling equivalent to OLDT or SLG resolution (Desouter, Van Dooren, and Schrijvers, 2015; Abdallah, 2017b; Abdallah, 2017c).

Continuations

Why consider continuations?

- Delimited continuations are really powerful: can implement many kinds of computational effects, including state, nondeterminism, all monads (Filinski, 1999) and tabling equivalent to OLDT or SLG resolution (Desouter, Van Dooren, and Schrijvers, 2015; Abdallah, 2017b; Abdallah, 2017c).
- First used for probabilistic programming (in OCaml) by Kiselyov and Shan (2008): programs yield a lazy search tree over probabilistic choices.

Continuations

Why consider continuations?

- Delimited continuations are really powerful: can implement many kinds of computational effects, including state, nondeterminism, all monads (Filinski, 1999) and tabling equivalent to OLDT or SLG resolution (Desouter, Van Dooren, and Schrijvers, 2015; Abdallah, 2017b; Abdallah, 2017c).
- First used for probabilistic programming (in OCaml) by Kiselyov and Shan (2008): programs yield a lazy search tree over probabilistic choices.
- Getting more interest on the functional side (Stuhlmüller and Goodman, 2012) and now in Anglican (Tolpin, Meent, and Wood, 2015).

Continuations

Why consider continuations?

- Delimited continuations are really powerful: can implement many kinds of computational effects, including state, nondeterminism, all monads (Filinski, 1999) and tabling equivalent to OLDT or SLG resolution (Desouter, Van Dooren, and Schrijvers, 2015; Abdallah, 2017b; Abdallah, 2017c).
- First used for probabilistic programming (in OCaml) by Kiselyov and Shan (2008): programs yield a lazy search tree over probabilistic choices.
- Getting more interest on the functional side (Stuhlmüller and Goodman, 2012) and now in Anglican (Tolpin, Meent, and Wood, 2015).
- Why should they have all the fun? Delimited control recently introduced into Prolog by Schrijvers et al., 2013.

Continuations

A *continuation*, at any point during program execution, is the 'rest of the program'. Focus on the expression $3{*}4$ in the small program below:

$$\textit{print}\ (1 +\ 3{*}4\ )$$

Continuations

A *continuation*, at any point during program execution, is the 'rest of the program'. Focus on the expression $3*4$ in the small program below:

$$\textit{print}\ (1 + \boxed{3*4})$$

The green region is the evaluation *context* for the expression. The continuation represents what happens next—'take the result, add 1 and print it.'

Continuations

A *continuation*, at any point during program execution, is the 'rest of the program'. Focus on the expression $3*4$ in the small program below:

$$\textit{print}\ (1 + \boxed{3*4})$$

The green region is the evaluation *context* for the expression. The continuation represents what happens next—'take the result, add 1 and print it.'

But why stop at the print? Taken to its logical conclusion, the *undelimited* continuation includes the whole OS and only ends when the computer crashes or you switch it off.

Hence, undelimited continuations don't return anything—they are only used for their side effects; they are not functions.

Delimited continuations

In order to manipulation continuations, we need a boundary, implicit or explicit, to create a *delimited continuation*, for example.

$$\textit{print}\ (1 +\ \ 3*4\,)$$

Delimited continuations

In order to manipulation continuations, we need a boundary, implicit or explicit, to create a *delimited continuation*, for example.

$$\textit{print}\ (1 + \boxed{3 * 4})$$

The purple region is a delimited evaluation context for the focussed expression. Now we can usefully 'reify' it (turn it into a thing) as pure function of type $int \to int$. Here, it is the function $\lambda x. 1 + x$.

Delimited continuations

In order to manipulation continuations, we need a boundary, implicit or explicit, to create a *delimited continuation*, for example.

$$\mathit{print}\ (1 + \boxed{3 * 4})$$

The purple region is a delimited evaluation context for the focussed expression. Now we can usefully 'reify' it (turn it into a thing) as pure function of type $int \to int$. Here, it is the function $\lambda x.1 + x$.

How can we control where the context boundaries are? How can we get hold of the continuations?

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\ (reset\ (\textbf{fun}\ ()\ \to\ (1 + 3 * 4)))$$

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\ (reset\ (\textbf{fun}\ () \to (1 + 3 * 4)))$$

shift allows us to capture a delimited continuation k and pass it to, e.g., a function h:

$$print\ (reset\ (\textbf{fun}\ () \to (1 + shift\ h)))$$

This *replaces* the entire delimited context with the return value from h.

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\ (reset\ (\mathbf{fun}\ ()\ \to\ (1 + 3 * 4\ )))$$

shift allows us to capture a delimited continuation k and pass it to, e.g., a function h:

$$\mathbf{let}\ h\ k = k\ (k\ 12)\ \mathbf{in}$$

$$print\ (reset\ (\mathbf{fun}\ ()\ \to\ (1 +\ shift\ h\ )))$$

This *replaces* the entire delimited context with the return value from h. h can do whatever it likes with the continuation.

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\,(reset\,(\mathbf{fun}\ () \to\ (1 + 3 * 4)\,))$$

shift allows us to capture a delimited continuation k and pass it to, e.g., a function h:

$$\mathbf{let}\ k = \mathbf{fun}\ x \to 1 + x\ \mathbf{in}$$

$$print\,(reset\,(\mathbf{fun}\ () \to\ k(k\,12)\,))$$

This *replaces* the entire delimited context with the return value from h. h can do whatever it likes with the continuation.

Finally, delimited contexts can be *nested*; then *shift* captures the continuation out to the innermost *reset*.

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\ (reset\ (\mathbf{fun}\ () \to (1 + 3 * 4)))$$

shift allows us to capture a delimited continuation k and pass it to, e.g., a function h:

$$\mathbf{let}\ k = \mathbf{fun}\ x \to 1 + x\ \mathbf{in}$$

$$print\ (reset\ (\mathbf{fun}\ () \to k\ 13))$$

This *replaces* the entire delimited context with the return value from h. h can do whatever it likes with the continuation.

Finally, delimited contexts can be *nested*; then *shift* captures the continuation out to the innermost *reset*.

Delimited control

In functional languages, delimited control often expressed using $reset : (unit \to \alpha) \to \alpha$ and $shift : (\beta \to \alpha) \to \beta$. Below, *reset* defines the delimited context (or *prompt*) in purple:

$$print\,(reset\,(\textbf{fun}\;()\to\;(1+\;3*4\,)))$$

shift allows us to capture a delimited continuation k and pass it to, e.g., a function h:

$$\textbf{let}\;k = \textbf{fun}\;x \to 1 + x\;\textbf{in}$$

$$print\,(reset\,(\textbf{fun}\;()\to\;14\,))$$

This *replaces* the entire delimited context with the return value from h. h can do whatever it likes with the continuation.

Finally, delimited contexts can be *nested*; then *shift* captures the continuation out to the innermost *reset*.

Delimited control (nondeterminism)

One more example—you should be able smell Prolog on the horizon. . .

$$\textbf{let}\ \mathit{choose}\ \mathit{xs} = \mathit{shift}\ (\textbf{fun}\ k \to \mathit{concat}\ (\mathit{map}\ k\ \mathit{xs}))\ \textbf{in}$$

$$\mathit{print}\ (\mathit{reset}\ (\textbf{fun}\ ()\ \to\ [1 + \mathit{choose}\ [1;2;3]\ ]))$$

Delimited control (nondeterminism)

One more example—you should be able smell Prolog on the horizon…

$$\textbf{let}\ \mathit{choose}\ \mathit{xs} = \mathit{shift}\ (\textbf{fun}\ k \to \mathit{concat}\ (\mathit{map}\ k\ \mathit{xs}))\ \textbf{in}$$

$$\mathit{print}\ (\mathit{reset}\ (\textbf{fun}\ ()\ \to [1 + \mathit{choose}\ [1;2;3]\ ]))$$

Delimited control (nondeterminism)

One more example—you should be able smell Prolog on the horizon. . .

$$\textbf{let } k : \textit{int} \to \textit{int list} = \textbf{fun } x \to [1 + x] \textbf{ in}$$

$$\textit{print } (\textit{reset } (\textbf{fun } () \to \textit{concat } (\textit{map } k \ [1;2;3])))$$

Delimited control (nondeterminism)

One more example—you should be able smell Prolog on the horizon. . .

$$\textbf{let } k : \mathit{int} \to \mathit{int\ list} = \textbf{fun } x \to [1 + x] \textbf{ in}$$

$$\mathit{print}\ (\mathit{reset}\ (\textbf{fun}\ () \to \mathit{concat}\ [[2];[3];[4]]\ ))$$

Delimited control (nondeterminism)

One more example—you should be able smell Prolog on the horizon...

$$\textbf{let } k : \mathit{int} \to \mathit{int\ list} = \textbf{fun } x \to [1 + x] \textbf{ in}$$

$$\mathit{print}\ (\mathit{reset}\ (\textbf{fun}\ () \to [2;3;4]\ ))$$

The result is a *list* of alternatives introduced by the *choose* operator, which has type $\alpha\ \mathit{list} \to \alpha$. This is one way to introduce nondeterminism into a functional language.

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$$X{=}1\ ,\ Y \text{ is } 3{*}4\ ,\ Z \text{ is } X{+}Y\ ,\ \mathit{writeln}(Z)$$

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$X{=}1$, Y is $3{*}4$, Z is $X{+}Y$, $writeln(Z)$ $\{\}$

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$Y \text{ is } 3*4$, $Z \text{ is } X+Y$, $writeln(Z)$ $\{X = 1\}$

Can put evaluation contexts around a subgoal in a similar way.

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$\boxed{\boxed{Z \text{ is } X{+}Y},\ \mathit{writeln}(Z)}\ \{X = 1,\ Y = 12\}$

Can put evaluation contexts around a subgoal in a similar way.

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$writeln(Z)$ $\{X = 1, Y = 12, Z = 13\}$

Can put evaluation contexts around a subgoal in a similar way.

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$writeln(Z)$ $\{X = 1, Y = 12, Z = 13\}$

Can put evaluation contexts around a subgoal in a similar way.
Schrijvers et al. (2013) use *reset*/3 and *shift*/1 to provide control.
I'm going to use *p_reset*/3 and *p_shift*/2, a very thin wrapper providing a better API and named prompts.

$p\_reset(nd, (X{=}1, p\_shift(nd, get(Y)), Z \text{ is } X{+}Y), Status)$ $\{\}$

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$\mathit{writeln}(Z)$ $\{X = 1, Y = 12, Z = 13\}$

Can put evaluation contexts around a subgoal in a similar way.
Schrijvers et al. (2013) use *reset*/3 and *shift*/1 to provide control.
I'm going to use *p_reset*/3 and *p_shift*/2, a very thin wrapper providing a better API and named prompts.

$\mathit{p\_reset}(\mathit{nd}, (\mathit{p\_shift}(\mathit{nd}, \mathit{get}(Y)), Z \text{ is } X+Y), \mathit{Status})$ $\{X = 1\}$

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$writeln(Z)$ $\{X = 1, Y = 12, Z = 13\}$

Can put evaluation contexts around a subgoal in a similar way. Schrijvers et al. (2013) use *reset*/3 and *shift*/1 to provide control. I'm going to use *p_reset*/3 and *p_shift*/2, a very thin wrapper providing a better API and named prompts.

true $\{X = 1, Status = susp(get(Y), Z \text{ is } X{+}Y)\}$

Unlike functional *reset*/*shift*, Prolog *shift*/1 doesn't decide what to do with continuation—it just sends a 'signal' $get(Y)$ along with continuation for later code to deal with. This is more like the *algebraic effect handlers* of Plotkin and Pretnar (2013).

Delimited control in Prolog

That's great for functional languages. What about Prolog?

$writeln(Z)$ $\{X = 1,\ Y = 12, Z = 13\}$

Can put evaluation contexts around a subgoal in a similar way.
Schrijvers et al. (2013) use *reset*/3 and *shift*/1 to provide control.
I'm going to use *p_reset*/3 and *p_shift*/2, a very thin wrapper providing a better API and named prompts.

true $\{X = 1,\ Status = susp(get(Y),\ Z \text{ is } X{+}Y)\}$

Unlike functional *reset*/*shift*, Prolog *shift*/1 doesn't decide what to do with continuation—it just sends a 'signal' $get(Y)$ along with continuation for later code to deal with. This is more like the *algebraic effect handlers* of Plotkin and Pretnar (2013).

N.B. real continuation is a bit more complex, but still an ordinary term.

Effect handlers in Prolog

Let's write a simple effect handler which responds to $get(Y)$ by consuming values from a list.

$$\boldsymbol{get}(Y) \leftarrow p\_shift(rdr,\ get(Y)).$$

$$\boldsymbol{run\_reader}(Goal,\ Values) \leftarrow$$
$$\quad p\_reset(rdr,\ Goal,\ Status),\ \ handle(Status,\ Values).$$

$$\boldsymbol{handle}(susp(get(Y),\ Cont),\ [Y\,|\,Ys]) \leftarrow run\_reader(Cont,\ Ys).$$
$$\boldsymbol{handle}(done,\ \_).$$

Effect handlers in Prolog

Let's write a simple effect handler which responds to *get*(Y) by consuming values from a list.

$$\textbf{\textit{get}}(Y) \leftarrow p\_shift(rdr,\ get(Y)).$$

$$\textbf{\textit{run\_reader}}(Goal,\ Values) \leftarrow$$
$$\quad p\_reset(rdr,\ Goal,\ Status),\ \ handle(Status,\ Values).$$

$$\textbf{\textit{handle}}(susp(get(Y),\ Cont),\ [Y \mid Ys]) \leftarrow run\_reader(Cont,\ Ys).$$
$$\textbf{\textit{handle}}(done,\ \_).$$

- Unifying Y with head of list sends data into *Cont*.

Effect handlers in Prolog

Let's write a simple effect handler which responds to $get(Y)$ by consuming values from a list.

$\boldsymbol{get}(Y) \leftarrow p\_shift(rdr,\ get(Y)).$

$\boldsymbol{run\_reader}(Goal,\ Values) \leftarrow$
$\quad p\_reset(rdr,\ Goal,\ Status),\ handle(Status,\ Values).$

$\boldsymbol{handle}(susp(get(Y),\ Cont),\ [Y|Ys]) \leftarrow run\_reader(Cont,\ Ys).$
$\boldsymbol{handle}(done,\ \_).$

- Unifying Y with head of list sends data into $Cont$.
- $handle/2$ invokes continuation using $run\_reader/2$.

Effect handlers in Prolog

Let's write a simple effect handler which responds to $get(Y)$ by consuming values from a list.

$$\textbf{get}(Y) \leftarrow p\_shift(rdr,\ get(Y)).$$

$$\textbf{run\_reader}(Goal,\ Values) \leftarrow$$
$$\quad p\_reset(rdr,\ Goal,\ Status),\ \ handle(Status,\ Values).$$

$$\textbf{handle}(susp(get(Y),\ Cont),\ [Y\,|\,Ys]) \leftarrow run\_reader(Cont,\ Ys).$$
$$\textbf{handle}(done,\ \_).$$

- Unifying Y with head of list sends data into $Cont$.
- $handle/2$ invokes continuation using $run\_reader/2$.
- Though $Goal$ may look 'impure', with $get/1$ as a computational effect, $run\_reader/2$ is pure. Effect is 'reified' using extra parameter $Values$.

Outline

Effects for a probabilistic program

$\leftarrow$ meta_predicate :=(3,−), *cctabled*(:,0), *sample*(3,−).

$\boldsymbol{dist}(Ps,Xs,X) \leftarrow p\_shift(prob,\ dist(Ps,Xs,X)).$
$\boldsymbol{uniform}(Xs,X) \leftarrow p\_shift(prob,\ uniform(Xs,X)).$
$\boldsymbol{sample}(P,X) \leftarrow p\_shift(prob,\ sample(P,X)).$
$SW \mathrel{\boldsymbol{:=}} X \leftarrow p\_shift(prob,\ sw(SW,X)).$

$\boldsymbol{cctabled}(Head,Work) \leftarrow p\_shift(tab,\ tcall(Head,Work,Inj)),\ call(Inj).$

- Effects are addressed to two different prompts *prob* and *tab*, which handle probabilistic choice and tabling respectively.

Effects for a probabilistic program

$\leftarrow$ meta_predicate :=(3,−), *cctabled*(:,0), *sample*(3,−).

$\boldsymbol{dist}(Ps,Xs,X) \leftarrow p\_shift(prob,\ dist(Ps,Xs,X)).$
$\boldsymbol{uniform}(Xs,X) \leftarrow p\_shift(prob,\ uniform(Xs,X)).$
$\boldsymbol{sample}(P,X) \quad \leftarrow p\_shift(prob,\ sample(P,X)).$
$SW \mathrel{\boldsymbol{:=}} X \qquad \leftarrow p\_shift(prob,\ sw(SW,X)).$

$\boldsymbol{cctabled}(Head, Work) \leftarrow p\_shift(tab,\ tcall(Head, Work, Inj)),\ call(Inj).$

- Effects are addressed to two different prompts *prob* and *tab*, which handle probabilistic choice and tabling respectively.
- SW identifies (it's actually a predicate) a parameterised distribution over terms, equivalent to PRISM switches.

Effects for a probabilistic program

$\leftarrow$ meta_predicate :=(3,−), *cctabled*(:,0), *sample*(3,−).

dist$(Ps,Xs,X) \leftarrow p\_shift(prob,\ dist(Ps,Xs,X))$.
uniform$(Xs,X) \leftarrow p\_shift(prob,\ uniform(Xs,X))$.
sample$(P,X) \leftarrow p\_shift(prob,\ sample(P,X))$.
SW **:=** $X \leftarrow p\_shift(prob,\ sw(SW,X))$.

cctabled$(Head, Work) \leftarrow p\_shift(tab,\ tcall(Head, Work, Inj)),\ call(Inj)$.

- Effects are addressed to two different prompts *prob* and *tab*, which handle probabilistic choice and tabling respectively.
- SW identifies (it's actually a predicate) a parameterised distribution over terms, equivalent to PRISM switches.
- Tabling effect allows effect handler to inject and arbitrary goal just after tabled call.

Effects handlers

Handler for a prompt named *prob*, implemented as a DCG to handle state threading, and delegating the actual handling to an arbitrary predicate *H*.

$\leftarrow$ meta_predicate *run_prob*(3,0,?,?).
run_prob(*H*,*Goal*) $\longrightarrow$ {*p_reset*(*prob*, *Goal*, *Stat*)}, *cont_prob*(*Stat*,*H*).

cont_prob(*susp*(*Req*,*Cont*),*H*) $\longrightarrow$ *call*(*H*,*Req*), *run_prob*(*H*,*Cont*).
cont_prob(*done*,_) $\longrightarrow$ [ ].

This is a very general handler—we could have called it *run_state_handler* and put it in a general purpose library.

Sampling execution without tabling

$\boldsymbol{sample}(P,\ sw(SW,X)) \longrightarrow\ !,\ call(P,SW,X).$
$\boldsymbol{sample}(\_,\ dist(Ps,Xs,X)) \longrightarrow\ !,\ pure(discrete(Xs,Ps),X).$
$\boldsymbol{sample}(\_,\ uniform(Xs,X)) \longrightarrow\ !,\ pure(uniform(Xs),X).$
$\boldsymbol{sample}(\_,\ sample(P,X)) \longrightarrow\ call(Q,X).$

$\boldsymbol{run\_notab}(Goal) \leftarrow p\_reset(tab,\ Goal,\ Stat),\ cont\_notab(Stat).$
$\boldsymbol{cont\_notab}(susp(tcall(\_,Work,Work),\ Cont)) \leftarrow run\_notab(Cont).$
$\boldsymbol{cont\_notab}(done).$

$\leftarrow$ meta_predicate $run\_sampling(4,0,+,-).$
$\boldsymbol{run\_sampling}(Sampler,Goal,S_1,S_2) \leftarrow$
$\quad run\_notab(run\_prob(sample(Sampler),Goal,S_1,S_2)).$

Threaded state includes state of pseudorandom generator.

Sampling execution without tabling

$\boldsymbol{sample}(P,\ sw(SW,X)) \quad\longrightarrow\ !,\ call(P,SW,X).$
$\boldsymbol{sample}(\_,\ dist(Ps,Xs,X)) \longrightarrow\ !,\ pure(discrete(Xs,Ps),X).$
$\boldsymbol{sample}(\_,\ uniform(Xs,X)) \longrightarrow\ !,\ pure(uniform(Xs),X).$
$\boldsymbol{sample}(\_,\ sample(P,X)) \quad\longrightarrow call(Q,X).$

$\boldsymbol{run\_notab}(Goal) \leftarrow p\_reset(tab,\ Goal,\ Stat),\ cont\_notab(Stat).$
$\boldsymbol{cont\_notab}(susp(tcall(\_,Work,Work),\ Cont)) \leftarrow run\_notab(Cont).$
$\boldsymbol{cont\_notab}(done).$

$\leftarrow$ meta_predicate $run\_sampling(4,0,+,-)$.
$\boldsymbol{run\_sampling}(Sampler,Goal,S_1,S_2) \leftarrow$
$\quad run\_notab(run\_prob(sample(Sampler),Goal,S_1,S_2)).$

Threaded state includes state of pseudorandom generator. *Sampler* encapsulates the distribution parameters for each switch—there is no global mutable state.

Sampling execution without tabling

$\boldsymbol{sample}(P,\ sw(SW,X)) \quad\longrightarrow\ !,\ call(P,SW,X).$
$\boldsymbol{sample}(\_,\ dist(Ps,Xs,X)) \longrightarrow\ !,\ pure(discrete(Xs,Ps),X).$
$\boldsymbol{sample}(\_,\ uniform(Xs,X)) \longrightarrow\ !,\ pure(uniform(Xs),X).$
$\boldsymbol{sample}(\_,\ sample(P,X)) \quad\longrightarrow call(Q,X).$

$\boldsymbol{run\_notab}(Goal) \leftarrow p\_reset(tab,\ Goal,\ Stat),\ cont\_notab(Stat).$
$\boldsymbol{cont\_notab}(susp(tcall(\_,Work,Work),\ Cont)) \leftarrow run\_notab(Cont).$
$\boldsymbol{cont\_notab}(done).$

$\leftarrow$ meta_predicate $run\_sampling(4,0,+,-)$.
$\boldsymbol{run\_sampling}(Sampler,Goal,S_1,S_2) \leftarrow$
$\quad run\_notab(run\_prob(sample(Sampler),Goal,S_1,S_2)).$

Threaded state includes state of pseudorandom generator. *Sampler* encapsulates the distribution parameters for each switch—there is no global mutable state. No tabling is done: worker goal merely injected into continuation.

Tabled explanation search (types)

$\leftarrow$ type $\boldsymbol{vc} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{swid(A)} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{factor} \longrightarrow @\mathit{number};\ \mathit{swid}(A){:=}A;\ \mathit{module}{:}\mathit{vc}.$
$\leftarrow$ type $\boldsymbol{sw(A)} == \mathit{pred}(-\mathit{swid}(A),\ -\mathit{list}(A),\ \mathit{list}(A)).$

A *vc* (variant class) represents all calls to a tabled goal with the same pattern of arguments and variables as a ground term (using *numbervars*/3):

Tabled explanation search (types)

$\leftarrow$ type $\boldsymbol{vc} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{swid(A)} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{factor} \longrightarrow @\mathit{number};\ \mathit{swid}(A){:=}A;\ \mathit{module}{:}\mathit{vc}.$
$\leftarrow$ type $\boldsymbol{sw(A)} == \mathit{pred}(-\mathit{swid}(A),\ -\mathit{list}(A),\ \mathit{list}(A)).$

A *vc* (variant class) represents all calls to a tabled goal with the same pattern of arguments and variables as a ground term (using *numbervars*/3):

A $\mathit{swid}(A)$ is a ground term uniqely identifiying a switch whose value are of type A. A $\mathit{sw}(A)$ is predicate which 'returns' a switch id and a difference list of the values the switch can take.

Tabled explanation search (types)

$\leftarrow$ type $\boldsymbol{vc} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{swid(A)} == \mathit{ground}.$
$\leftarrow$ type $\boldsymbol{factor} \longrightarrow @number;\ swid(A){:=}A;\ module{:}vc.$
$\leftarrow$ type $\boldsymbol{sw(A)} == pred(-swid(A),\ -list(A),\ list(A)).$

A *vc* (variant class) represents all calls to a tabled goal with the same pattern of arguments and variables as a ground term (using *numbervars*/3):

A $swid(A)$ is a ground term uniqely identifiying a switch whose value are of type A. A $sw(A)$ is predicate which 'returns' a switch id and a difference list of the values the switch can take.

A *factor* explains a probabilistic deduction step—it is either the probability of a choice from a fixed distribution, a switch with one of its values, or a module-qualified variant class representing a tabled subgoal.

Tabled explanation search (explanations)

$\boldsymbol{expl}(M{:}VC) \longrightarrow [M{:}VC].$
$\boldsymbol{expl}(SW{:=}X) \longrightarrow \{call(SW,ID,Xs,[\,]),\ member(X,Xs)\},\ [ID{:=}X].$
$\boldsymbol{expl}(dist(Ps,Xs,X)) \longrightarrow \{member2(P,X,Ps,Xs)\},\ [@P].$
$\boldsymbol{expl}(uniform(Xs,X)) \longrightarrow \{length(Xs,N),\ P \text{ is } 1/N,\ member(X,Xs)\},\ [@P].$

$\boldsymbol{term\_to\_variant\_class}(T_1,\ T_2) \leftarrow$
$\quad copy\_term\_nat(T_1,T_2),$
$\quad numbervars(T_2,0,\_).$

$\boldsymbol{member2}(X,Y,[X|\_],[Y|\_]).$
$\boldsymbol{member2}(X,Y,[\_|Xs],[\_|Ys]) \leftarrow member2(X,Y,Xs,Ys).$

Tabled explanation search (tabling types)

$\leftarrow$ type $\boldsymbol{soln} == \mathit{list}(\mathit{term})$.
$\leftarrow$ type $\boldsymbol{kont} \longrightarrow k(\mathit{list}(\mathit{var}),\mathit{term},\mathit{pred})$.
$\leftarrow$ type $\boldsymbol{table} \longrightarrow \mathit{tab}(\mathit{goal},\mathit{rbtree}(\mathit{soln},\mathit{list}(\mathit{list}(\mathit{factor}))),\mathit{list}(\mathit{cont}))$.

A *soln* (solution) is a list of values taken by variables in a tabled call.

Tabled explanation search (tabling types)

$\leftarrow$ type $\boldsymbol{soln} == \mathit{list}(\mathit{term})$.
$\leftarrow$ type $\boldsymbol{kont} \longrightarrow k(\mathit{list}(\mathit{var}),\mathit{term},\mathit{pred})$.
$\leftarrow$ type $\boldsymbol{table} \longrightarrow \mathit{tab}(\mathit{goal},\mathit{rbtree}(\mathit{soln},\mathit{list}(\mathit{list}(\mathit{factor}))),\mathit{list}(\mathit{cont}))$.

A *soln* (solution) is a list of values taken by variables in a tabled call.

A *kont* (continuation with context variables) is a continuation along with the variables to 'communicate' with it.

Tabled explanation search (tabling types)

$\leftarrow$ type $\boldsymbol{soln} == \mathit{list}(\mathit{term})$.
$\leftarrow$ type $\boldsymbol{kont} \longrightarrow k(\mathit{list}(\mathit{var}),\mathit{term},\mathit{pred})$.
$\leftarrow$ type $\boldsymbol{table} \longrightarrow \mathit{tab}(\mathit{goal},\mathit{rbtree}(\mathit{soln},\mathit{list}(\mathit{list}(\mathit{factor}))),\mathit{list}(\mathit{cont}))$.

A *soln* (solution) is a list of values taken by variables in a tabled call.

A *kont* (continuation with context variables) is a continuation along with the variables to 'communicate' with it.

A table contains the tabled goal itself (with variables), a map associating each solution with a list of explanations (each of which is a list of factors), and a list of continuations waiting for results from a tabled call.

Tabled explanation search (tabling)

```
← use_module(library(rbutils)).
← use_module(ccnbenv).

← meta_predicate run_tab(0,?)
run_tab(Goal, Ans) ← p_reset(tab, Goal, Stat), cont_tab(Stat, Ans).

cont_tab(done, _).
cont_tab(susp(tcall(M:H,Work,p_shift(prob,M:VC)), Cont), Ans) ←
    term_to_variant_class(H, VC),
    term_variables(Work, Y), K = k(Y,Ans,Cont),
    nb_app_or_new(M:VC, old_vc(R,K), new_vc(R,M:H,K)),
    (   R =solns(Ys) → rb_in(Y, _, Ys), run_tab(Cont, Ans)
    ;   R =new → run_tab(producer(M:VC, λY.Work, Ans), Ans)
    ).

old_vc(solns(Ys), K, tab(H,Ys,[K0|Ks]), tab(H,Ys,[K0,K|Ks])).
new_vc(new, H, K, tab(H,Ys,[K])) ← rb_empty(Ys).
```

Tabled explanation search (tabling)

$$
\begin{aligned}
&\textbf{\textit{producer}}(VC,\ Generate,\ Ans) \leftarrow \\
&\quad run\_prob(expl,\ call(Generate,\ \ Y),\ E,\ [\,]), \\
&\quad nb\_app(VC,\ new\_soln(Y,E,Res)), \\
&\quad Res{=}new(Ks),\ member(k(Y,Ans,C),\ Ks),\ call(C).
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{\textit{new\_soln}}(Y,\ E,\ Res,\ tab(V,Ys_1,Ks),\ tab(V,Ys_2,Ks)) \leftarrow \\
&\quad rb\_app\_or\_new(Y,\ old\_soln(Res,E),\ new\_soln(Res,Ks,E),\ Ys_1,\ Ys_2).
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{\textit{new\_soln}}(new(Ks),Ks,E,[E]). \\
&\textbf{\textit{old\_soln}}(old,E,Es,[E|Es]).
\end{aligned}
$$

This is basically the same as the tabling algorithm in (Abdallah, 2017c), slightly modified to collect explanations for each solution, instead of just collecting solutions in a set.

Dice model again

Before shallow program transformations:

$\leftarrow$ **module**(*eg*, [*die*//1, *dice*/2]).

die $\longmapsto$ [1,2,3,4].

$\leftarrow$ cctable *dice*/2.
dice(0,0).
dice(N,Z) $\leftarrow$
  die := X,
  succ(M,N), *dice*(M, Y),
  Z is X+Y.

NB. Probabilistic predicates *and* switches are module scoped.

Dice model again

After program transformations (but before DCG expansion):

$\leftarrow$ **module**(*eg*, [*die*//1, *dice*/2]).

die(*eg*:*die*) $\longrightarrow$ [1,2,3,4].

dice(N,Z) $\leftarrow$ *cctabled*(*dice*(N,Z),'*dice#*'(N,Z)).
'***dice#***'(0,0).
'***dice#***'(N,Z) $\leftarrow$
  die := X,
  succ(M,N), *dice*(M, Y),
  Z is X+Y.

NB. Probabilistic predicates *and* switches are module scoped.

Building the explanation graph

```
← use_module(library(rbutils)).
← use_module(ccprism/handlers)).
← use_module(ccprism/graph).
← use_module(ccnbenv).

← meta_predicate goal_graph(0,−).
goal_graph(Goal, G1) ←
    run_nb_env(goal_expls_tables(Goal, Es, Ts)),
    tables_graph(Ts, G0),
    prune_graph(=, '#top':top, [('#top':top)−Es | G0 ], G1).

goal_expls_tables(Goal, Es, Ts) ←
  run_tab(findall(E, run_prob(expl,Goal,E,[]), Es)),
  nb_dump(Ts).
```

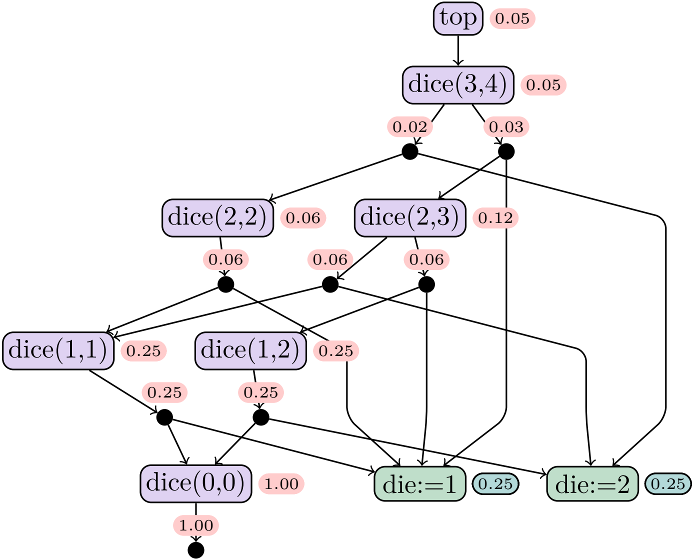
Building the explanation graph

$\textbf{\textit{tables\_graph}}(Ts,\ Graph) \leftarrow$
$\quad rb\_empty(Empty),$
$\quad rb\_fold(table\_expls,\ \ Ts,\ Empty,\ GMap),$
$\quad rb\_visit(GMap,\ Graph).$

$\textbf{\textit{table\_expls}}(\_-tab(Goal,Solns,\_)) \longrightarrow$
$\quad \{term\_variables(Goal,Vars)\},$
$\quad rb\_fold(soln\_expls(Goal,Vars),\ Solns).$

$\textbf{\textit{soln\_expls}}(G,Y,Y_1-Es) \longrightarrow$
$\quad \{copy\_term(G-Y,G_1-Y_1),\ numbervars(G_1-Y_1,\ 0,\ \_)\},$
$\quad (rb\_add(G_1,Es) \rightarrow [\,];\ [\,]).$

Explanation graph (again)



Semiring graph processing

Generalised processing over parse forests (Goodman, 1998; Goodman, 1999). Idea is to replace 'OR' and 'AND' nodes of graph with 'plus' and 'times' operators from a semiring.

A semiring is an algebra with a set of values and two binary operators $\oplus$ and $\otimes$, both monoidal (having identity elements $\mathbf{0}$ and $\mathbf{1}$ respectively), and with some additional conditions. Goodman shows how many useful parsing algorithms can be defined using the same computation with different semirings, including $(+, 0, \times, 1)$ over reals for inside algorithm, $(\max, -\infty, \times, 1)$ for Viterbi algorithm, and operations over sets of lists for parse tree extraction.

Generalised semiring

More convenient to generalise types:

$$
\begin{aligned}
\otimes &: \alpha \times \beta \to \beta \\
\oplus &: \beta \times \gamma \to \gamma \\
\mathbf{1} &: \beta, \\
\mathbf{0} &: \gamma \\
\mathit{inj} &: \mathit{factor} \times \theta \to \alpha \\
\mathit{proj} &: \gamma \to \alpha
\end{aligned}
$$

Idea is to build a dataflow graph on the explanation graph, replacing factor nodes with semiring operations, using *inj* to get initial values from switch nodes and parameters and *proj* to feed output of goal nodes back into product nodes.

Semiring graph processing

```
← use_module(library(dcg_pair)).
← use_module(library(rbutils)).

← type sr(A,B,C,T). % open union type
semiring_graph_fold(SR, Graph, Params, GoalSums) ←
    rb_empty(E),
    foldl(sr_sum(SR), Graph, GoalSums, E, FMap),
    fmap_sws(FMap, SWs),
    maplist(fmap_sw_vals(sr_param(SR),true1,FMap),SWs,Params).

sr_param(SR,F,X,P) ← sr_inj(SR,F,P,X), !.
true1(_).
```

NB. order of graph traversal is not important because we can use constraint based arithmetic predicates, e.g. $CLP(R)$, to delay numerics until variables are instantiated. Hence *Params* can contain variables for switch value probabilities.

Semiring graph processing

$\textbf{\textit{sr\_sum}}(SR,\ Goal{-}Expls,\ Goal{-}Sum1) \longrightarrow$
$\quad fmap(Goal,Proj),\ \{sr\_zero(SR,Zero)\},$
$\quad run\_right(foldr(sr\_add\_prod(SR),Expls),\ Zero,\ Sum),$
$\quad \{sr\_proj(SR,Goal,Sum,Sum1,Proj)\}.$

$\textbf{\textit{sr\_add\_prod}}(SR,\ Expl) \longrightarrow$
$\quad \{sr\_unit(SR,Unit)\},$
$\quad run\_right(foldr(sr\_factor(SR),\ Expl),\ \ Unit,\ \ Prod)\ \langle\backslash\rangle\ sr\_plus(SR,Prod).$

$\textbf{\textit{sr\_factor}}(SR,\ M{:}Head)\ \ \longrightarrow\ !,\ fmap(M{:}Head,X)\ \langle\backslash\rangle\ sr\_times(SR,X).$
$\textbf{\textit{sr\_factor}}(SR,\ SW{:=}Val) \longrightarrow\ !,\ fmap(SW{:=}Val,X)\ \langle\backslash\rangle\ sr\_times(SR,X).$
$\textbf{\textit{sr\_factor}}(SR,\ @P)\ \ \longrightarrow \{sr\_inj(SR,const,P,X)\},\ \backslash\rangle\ sr\_times(SR,X).$

By providing clauses of the *sr_* predicates, this *one* piece of code handles the inside and Viterbi algorithms with linear or log scaled probabilities, best and k-best explanation tree extraction, graph annotation, and any combination of these by semiring *composition*.

Factor-value map

Associative map from factors to α values in generalised semiring.

$\boldsymbol{fmap}(X,Y) \longrightarrow rb\_add(X,Y) \rightarrow [\,];\ rb\_get(X,Y).$
$\boldsymbol{fmap\_sws}(Map,SWs) \leftarrow$
$\quad rb\_fold(emit\_if\_sw,Map,SWs1,[\,]),$
$\quad sort(SWs1,SWs).$
$\boldsymbol{emit\_if\_sw}(F-\_) \longrightarrow \{F=(SW{:=}\_)\} \rightarrow [SW];\ [\,].$

$\leftarrow$ meta_predicate $fmap\_sw\_vals(3,1,+,+,?)$.
$\boldsymbol{fmap\_sw\_vals}(Conv,Def,Map,SW,SW-XX) \leftarrow$
$\quad call(SW,\_,Vals,[\,]),$
$\quad maplist(sw\_val\_or\_default(Conv,Def,Map,SW),Vals,XX).$

$\boldsymbol{sw\_val\_or\_default}(Conv,Def,Map,SW,Val,X) \leftarrow$
$\quad (\quad rb\_lookup(SW{:=}Val,\ P,\ Map)$
$\quad \rightarrow call(Conv,SW{:=}Val,P,X)$
$\quad ;\quad call(Def,X)$
$\quad).$

Semiring definitions

Numeric and list based semirings:
$r(pred(T,A),\ pred(C,A),\ pred(A,B,B),\ pred(B,C,C))\ :\ sr(A,B,C,T).$

$\boldsymbol{sr\_inj}(r(I,\_,\_,\_),\ \_,\ P,\ X) \quad \leftarrow call(I,P,X).$
$\boldsymbol{sr\_proj}(r(\_,P,\_,\_),\ \_,\ X,\ Y,\ Y) \leftarrow call(P,X,Y).$
$\boldsymbol{sr\_plus}(r(\_,\_,\_,O),\ X) \longrightarrow call(O,X).$
$\boldsymbol{sr\_times}(r(\_,\_,O,\_),\ X) \longrightarrow call(O,X).$
$\boldsymbol{sr\_zero}(r(\_,\_,\_,O),\ I) \leftarrow m\_zero(O,I).$
$\boldsymbol{sr\_unit}(r(\_,\_,O,\_),\ I) \leftarrow m\_zero(O,I).$

$\boldsymbol{m\_zero}(add,0.0).$
$\boldsymbol{m\_zero}(mul,1.0).$
$\boldsymbol{m\_zero}(max,-inf).$
$\boldsymbol{m\_zero}(cons,[\,]).$

Semiring definitions (Viterbi)

Much like $r(=,=,mul,max)$, but keeping the most likely exlanation subtree along.

$\boldsymbol{sr\_inj}(best(log),\ F,\ P,\ P{-}F) \leftarrow !.$
$\boldsymbol{sr\_inj}(best(lin),\ F,\ P,\ Q{-}F) \quad \leftarrow log\_e(P,Q).$
$\boldsymbol{sr\_proj}(best(\_),\ \ G,\ X{-}E,\ X{-}E,\ X{-}(G{-}E)).$
$\boldsymbol{sr\_plus}(best(\_),\ \ X) \longrightarrow max\_by\_fst(X).$
$\boldsymbol{sr\_times}(best(\_),\ \ X{-}F) \longrightarrow add(X)\ \langle\backslash\rangle\ cons(F).$
$\boldsymbol{sr\_zero}(best(\_),\ \ Z{-}\_) \quad \leftarrow m\_zero(max,Z).$
$\boldsymbol{sr\_unit}(best(\_),\ \ 0.0{-}[\,]).$

$\boldsymbol{max\_by\_fst}(LX{-}X,LY{-}Y,Z) \leftarrow$
$\quad when(ground(LX{-}LY),(LX \geq LY \rightarrow Z = LX{-}X;\ Z = LY{-}Y)).$

Semiring definitions (annotation)

Use any semiring to annotate explanation graph.

$$\boldsymbol{sr\_inj}(ann(SR),\ \ F,\ P,\ Q{-}F)\ \ \leftarrow sr\_inj(SR,F,P,Q).$$
$$\boldsymbol{sr\_proj}(ann(SR),\ \ G,\ X{-}Z,\ W{-}Z,\ Y{-}G)\quad \leftarrow sr\_proj(SR,G,X,W,Y).$$
$$\boldsymbol{sr\_plus}(ann(SR),\ \ X{-}Expl) \longrightarrow sr\_plus(SR,X)\ \langle\backslash\backslash\rangle\ cons(X{-}Expl).$$
$$\boldsymbol{sr\_times}(ann(SR),\ \ X{-}F) \longrightarrow sr\_times(SR,X)\ \langle\backslash\backslash\rangle\ cons(X{-}F).$$
$$\boldsymbol{sr\_zero}(ann(SR),\ \ Z{-}[\,])\ \leftarrow sr\_zero(SR,Z).$$
$$\boldsymbol{sr\_unit}(ann(SR),\ \ U{-}[\,])\ \leftarrow sr\_unit(SR,U).$$

Semiring definitions (pair)

Combine results from any two semirings.

$\boldsymbol{sr\_inj}(R_1{-}R_2,\ F,\ P,\ Q_1{-}Q_2) \leftarrow sr\_inj(R_1,F,P,Q_1),\ sr\_inj(R_2,F,P,Q_2).$
$\boldsymbol{sr\_proj}(R_1{-}R_2,\ G,\ X_1{-}X_2,\ Z_1{-}Z_2,\ Y_1{-}Y_2) \leftarrow$
$\quad sr\_proj(R_1,G,X_1,Z_1,Y_1),\ sr\_proj(R_2,G,X_2,Z_2,Y_2).$
$\boldsymbol{sr\_plus}(R_1{-}R_2,\ X_1{-}X_2) \longrightarrow sr\_plus(R_1,X_1)\ \langle\backslash\rangle\ sr\_plus(R_2,X_2).$
$\boldsymbol{sr\_times}(R_1{-}R_2,\ X_1{-}X_2) \longrightarrow sr\_times(R_1,X_1)\ \langle\backslash\rangle\ sr\_times(R_2,X_2).$
$\boldsymbol{sr\_zero}(R_1{-}R_2,\ Z_1{-}Z_2) \leftarrow sr\_zero(R_1,Z_1),\ sr\_zero(R_2,Z_2).$
$\boldsymbol{sr\_unit}(R_1{-}R_2,\ U_1{-}U_2) \leftarrow sr\_unit(R_1,U_1),\ sr\_unit(R_2,U_2).$

Semiring definitions (lazy best first)

Lazy, unbounded version of Huang and Chiang (2005)

```
sr_inj(kbest,   F, P, [Q-F]) <- surp(P,Q).
sr_proj(kbest,  G, X, X, Y)  <- freeze(Y,lazy_maplist(k_tag(G),X,Y)).
sr_plus(kbest,  X) --> lazy(k_min,X).
sr_times(kbest, X) --> lazy(k_mul,X).
sr_zero(kbest,  []).
sr_unit(kbest,  [0.0-[]]).

k_tag(G,L-X,L-(G-X)).
k_min([],Ys,Ys) <- !.
k_min(Xs,[],Xs) <- !.
k_min([X|Xs],[Y|Ys],[Z|Zs]) <-
   (   LX-_=X, LY-_=Y, LX =< LY
   ->  Z=X, freeze(Zs, k_min(Xs,[Y|Ys],Zs))
   ;   Z=Y, freeze(Zs, k_min([X|Xs],Ys,Zs))
   ).
```

Semiring definitions (lazy best first)

$$
\begin{array}{l}
\textbf{\textit{k\_mul}}(Xs, Ys, Zs) \leftarrow \\
\quad \textit{empty\_set}(EmptyS),\ \textit{empty\_heap}(EmptyQ), \\
\quad \textit{enqueue}(\textit{pos}(0{-}0, Xs, Ys),\ EmptyS{-}EmptyQ,\ TQ_1), \\
\quad \textit{lazy\_unfold\_finite}(\textit{k\_next},\ Zs,\ TQ_1,\ \_).
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{k\_next}}(L{-}[XF \mid YFs]) \longrightarrow \\
\quad \backslash\rangle\ \textit{pq\_get}(L, \textit{pos}(I{-}J, [X0 \mid Xs], [Y0 \mid Ys])), \\
\quad \{\_{-}XF{=}X0,\ \_{-}YFs{=}Y0,\ \textit{succ}(I, I_1),\ \textit{succ}(J, J_1)\}, \\
\quad \textit{enqueue}(\textit{pos}(I_1{-}J, Xs, [Y0 \mid Ys])), \\
\quad \textit{enqueue}(\textit{pos}(I{-}J_1, [X0 \mid Xs], Ys)).
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{enqueue}}(P) \longrightarrow \textit{new\_position\_cost}(P, L) \rightarrow \backslash\rangle\ \textit{pq\_add}(L, P);\ [\,]. \\
\textbf{\textit{new\_position\_cost}}(\textit{pos}(IJ, [X0 \mid \_], [Y0 \mid \_]), L) \longrightarrow \\
\quad \backslash\langle\ \textit{add\_to\_set}(IJ),\ \{L \text{ is } X0{+}Y0\}.
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{pq\_add}}(L, P, H_1, H_2) \leftarrow \textit{add\_to\_heap}(H_1, L, P, H_2). \\
\textbf{\textit{pq\_get}}(L, P, H_1, H_2) \leftarrow \textit{get\_from\_heap}(H_1, L, P, H_2).
\end{array}
$$

Outside algorithm in PRISM

Learning switch parameters requires expected sufficient statistics (pseudocounts representing how often each switch value is used in explanation graph).

Possibly Sato and Kameya (2001) were the first to notice that this can be done by partial differentiation of probability of top goal wrt switch parameters, then multiplying by inside probabilities:

$$\eta_{s,i} = \frac{\theta_{s,i}}{P_t} \frac{\partial P_t}{\partial \theta_{s,i}}$$

where $\theta_{s,i}$ is the probability of switch s taking value i, $\eta_{s,i}$ is the corresponding statistic, and P_t is the inside probability of the top goal.

In PRISM, this computation is expanded by hand into an explicit traversal of explanation graph annoted with inside probabilities.

ESS via automatic differentiation

Using CLP-based automatic differentiation in CHR/Prolog (Abdallah, 2017a) we can do away with all this code: simply compute the log (inside) probability of the top goal wrt to the *log* scaled switch value probabilities using a semiring composed of *differentiable* operators to get

$$\eta_{s,i} = \frac{\partial \log P_t}{\partial \log \theta_{s,i}}.$$

I suspect (not confirmed) that this will generalise to Viterbi training simply by using differentiable *max* instead of *add* in semiring.

Also expected to be useful in implementing new classes of switch distributions (e.g. exponential families) and gradient based learning (cf. deep learning).

ESS via automatic differentiation

```
← use_module(library(autodiff2), [llog/2, log/2, exp/2, add/3, mul/3,
                                  back/1, deriv/3, compile/0]).
m_zero(autodiff2:mul,1.0).
m_zero(autodiff2:add,0.0).

graph_counts(PSc, Graph, Params, Eta, LogProb) ←
    SR = r(=,=,autodiff2:mul,autodiff2:add),
    semiring_graph_fold(SR, Graph, P0, IG),
    top_value(IG, Prob), log(Prob, LogProb),
    scaling_log_params(PSc, P0, Params0, LogP0),
    map_swc(deriv(LogProb), LogP0, Eta),
    back(LogProb), compile, Params=Params0.

scaling_log_params(lin, P0, P0, LP0) ← map_swc(llog, P0, LP0).
scaling_log_params(log, P0, LP0, LP0) ← map_swc(exp, LP0, P0).
```

Learning via expectation-maximisation (EM)

We can now do EM learning (with inverse temperature for deterministic annealing) as follows: *learn*/4 returns in its fourth argument a predicate to do one step of learning.

$\boldsymbol{learn}(ml,\ ITemp,\ Graph,\ unify3(t(P_1,P_2,LP))) \leftarrow$
$\quad once(graph\_counts(lin,\ Graph,\ PP,\ Eta,\ LP)),$
$\quad map\_swc(pow(ITemp),\ P_1,\ PP),$
$\quad map\_sw(stoch,\ Eta,\ P_2).$

$\boldsymbol{unify3}(CVars,LP,P_1,P_2) \leftarrow copy\_term(CVars,\ t(P_1,P_2,LP)).$

This works because using $CLP(R)$ or similar, we can build the entire numerical dataflow graph *once* with *uninstantiated* variables. We can then use the graph multiple times by copying all the variables (including constraints), unifying the inputs with numerical values, and reading off the outputs.

Convergence of learning steps

General tool for running single step repeatedly to convergence:

$\leftarrow$ meta_predicate *converge*(+,1,−,+,−).
converge(*Test*, *Setup*, [*X0*|*History*], *S0*, *SFinal*) $\leftarrow$
  time(*call*(*Setup*, *Step*)),
  call(*Step*, *X0*, *S0*, S_1),
  converge_x(*Test*, *Step*, *X0*, *History*, S_1, *SFinal*).

converge_x(*Test*, *Step*, *X0*, [X_1|*History*], S_1, *SFinal*) $\leftarrow$
  call(*Step*, X_1, S_1, S_2),
  (*converged*(*Test*, *X0*, X_1) $\rightarrow$ *History*=[], *SFinal*=S_2
  ; *converge_x*(*Test*, *Step*, X_1, *History*, S_2, *SFinal*)
).

converged(*abs*(*Eps*), X_1, X_2) $\leftarrow$ $abs(X_1 - X_2) \leq Eps$.
converged(*rel*(*Del*), X_1, X_2) $\leftarrow$ $abs((X_1 - X_2)/(X_1 + X_2)) \leq Del$.

Declarative learning

Both maximum *a posteriori* learning (where there is a Dirichlet prior over switch probability parameters) and variational Bayes (where we learn a distribution over switch parameters, not point estimates) can be implemented in another 22 lines, reusing the same convergence tool.

Declarative learning

Both maximum *a posteriori* learning (where there is a Dirichlet prior over switch probability parameters) and variational Bayes (where we learn a distribution over switch parameters, not point estimates) can be implemented in another 22 lines, reusing the same convergence tool.

All learning is pure declarative Prolog: parameters are input and output via arguments and there are no global variables.

Declarative learning

Both maximum *a posteriori* learning (where there is a Dirichlet prior over switch probability parameters) and variational Bayes (where we learn a distribution over switch parameters, not point estimates) can be implemented in another 22 lines, reusing the same convergence tool.

All learning is pure declarative Prolog: parameters are input and output via arguments and there are no global variables.

All variations on learning (linear or log scaled variants, temperatures, priors etc.) are controlled by explicit parameters, not implicit global settings.

Declarative learning

Both maximum *a posteriori* learning (where there is a Dirichlet prior over switch probability parameters) and variational Bayes (where we learn a distribution over switch parameters, not point estimates) can be implemented in another 22 lines, reusing the same convergence tool.

All learning is pure declarative Prolog: parameters are input and output via arguments and there are no global variables.

All variations on learning (linear or log scaled variants, temperatures, priors etc.) are controlled by explicit parameters, not implicit global settings.

Gibbs and Metropolis-Hastings samplers implemented purely (using sampling effect handler) in another ~ 90 lines.

Outline

Examples: Sampling

The dice model given earlier can be sampled using *run_sampling*//2. We must provide a predicate to act as a database of switch distributions, e.g., using *uniform_sampler*//2 to assume a uniform distribution for all switches.

```
?- length(Xs,3),
   strand(run_sampling(uniform_sampler,maplist(dice(3),Xs))).
Xs = [10, 7, 6] .
```

Here *strand*/1 is a utility from an independent package *plrand* providing a random generator and various sampling distributions. *strand*(G) runs G as a DCG goal with the initial state set to a random RNG state.

Examples: Sampling

If instead we want a particular distribution for switch *die*, we can provide it using a 'lookup sampler':

```
?- make_lookup_sampler([(eg:die)-[0.5,0.1,0.3,0.1]], S),
   strand(run_sampling(S,maplist(dice(3),Xs))),
   length(Xs,3).
Xs = [8, 5, 5],
S = ccp_handlers:lookup_sampler(<rbtree>).
```

Examples: graph building

To build and pretty-print an explanation graph:

```
?- goal_graph(dice(3,4),G), print_term(G,[]).

[ ('.top' : top) - [[eg:dice(3,4)]],
  (eg : dice(0,0)) - [[]],
  (eg : dice(1,1)) - [[eg:die:=1,eg:dice(0,0)]],
  (eg : dice(1,2)) - [[eg:die:=2,eg:dice(0,0)]],
  (eg : dice(2,2)) - [[eg:die:=1,eg:dice(1,1)]],
  (eg : dice(2,3)) - [[eg:die:=2,eg:dice(1,1)],
                      [eg:die:=1,eg:dice(1,2)]],
  (eg : dice(3,4)) - [[eg:die:=2,eg:dice(2,2)],
                      [eg:die:=1,eg:dice(2,3)]]
]
```

Examples: inside probabilities

Note that parameters P get numerical values *after* running the inside agloritthm on the graph.

```
?- goal_graph(dice(3,4),G),
   semiring_graph_fold(r(=,=,mul,add),G,P,IG),
   graph_params(uniform,G,P),
   print_term(IG,[]).

[ ('.top' : top) - 0.046875,
  (eg : dice(0,0)) - 1,
  (eg : dice(1,1)) - 0.25,
  (eg : dice(1,2)) - 0.25,
  (eg : dice(2,2)) - 0.0625,
  (eg : dice(2,3)) - 0.125,
  (eg : dice(3,4)) - 0.046875
]
```

Examples: more semirings

Showing only the calls and not the output, first an explanation graph annotated with inside probabilities:

```
?- goal_graph(dice(3,4),G),
   semiring_graph_fold(ann(r(=,=,mul,add)),G,P,IG),
   graph_params(uniform,G,P),
   print_term(IG,[]).
```

Now each subgoal with log probability of most likely explanation, using log scaled probabilities:

```
?- goal_graph(dice(3,4),G),
   semiring_graph_fold(r(log_e,=,add,max),G,P,VG),
   graph_params(uniform,G,P),
   print_term(VG,[]).
```

Examples: expected switch-value counts

Compute expected sufficient statistics given log-scaled switch parameters and using log-scaled inside algorithm:

```
?- goal_graph(dice(3,4),G),
   graph_counts(log,log,G,P,Eta,LP),
   graph_params(log(uniform),G,P).

G = [...],
P = [(eg:die)-[-1.3863, -1.3863, -1.3863, -1.3863]],
Eta = [(eg:die)-[2, 1, 0, 0]],
LP = -3.0603.
```

In this case, all explanations use *die*:=1 twice and *die*:=2 once.

Examples: sampling and learning

This is a longer example combining sampling a dataset of length N and trying to learn the die distribution from it. The learned parameters are returned in P_1 and the history of likelihood values in H.

$\textbf{\textit{sample\_and\_learn\_dice}}(N,H,P_1,R_1,R_2) \leftarrow$
$\quad \textit{length}(Xs,N),$
$\quad \textit{make\_lookup\_sampler}([(eg{:}die)-[0.2,0.4,0.3,0.1]],S),$
$\quad \textit{strand}(\textit{run\_sampling}(S,\textit{maplist}(\textit{dice}(3),Xs)),R_1,R_2),$
$\quad \textit{goal\_graph}(\textit{maplist}(\textit{dice}(3),Xs),\ G),$
$\quad \textit{graph\_params}(\textit{uniform},G,P0),$
$\quad \textit{converge}(\textit{abs}(1e{-}7),\ \textit{learn}(\textit{ml},\textit{io}(\textit{log}),G),\ \ H,\ P0,\ P_1).$

This predicate has no side effects and no mutable global state is modified or referenced. The state of the random generator is passed in and out in R_1 and R_2.

Outline

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.
- Code is mostly pure, declarative Prolog, with a few metalinguistic constructs required to implement tabling.

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.
- Code is mostly pure, declarative Prolog, with a few metalinguistic constructs required to implement tabling.
- Both these aspects are intended to encourage reading, understanding, modification and extension of the code.

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.
- Code is mostly pure, declarative Prolog, with a few metalinguistic constructs required to implement tabling.
- Both these aspects are intended to encourage reading, understanding, modification and extension of the code.
- Performance: tabling is bit slow, but can be much improved using methods of Abdallah, 2017c. EM learning is slower than PRISM, but faster than equivalent computation graph implemented using Theano or TensorFlow.

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.
- Code is mostly pure, declarative Prolog, with a few metalinguistic constructs required to implement tabling.
- Both these aspects are intended to encourage reading, understanding, modification and extension of the code.
- Performance: tabling is bit slow, but can be much improved using methods of Abdallah, 2017c. EM learning is slower than PRISM, but faster than equivalent computation graph implemented using Theano or TensorFlow.
- Future work: switches distributions from exponential families. Bayesian non-parametetrics.

Conclusions

- It was possible to implement the main features of PRISM using an order of magnitude less code.
- Code is mostly pure, declarative Prolog, with a few metalinguistic constructs required to implement tabling.
- Both these aspects are intended to encourage reading, understanding, modification and extension of the code.
- Performance: tabling is bit slow, but can be much improved using methods of Abdallah, 2017c. EM learning is slower than PRISM, but faster than equivalent computation graph implemented using Theano or TensorFlow.
- Future work: switches distributions from exponential families. Bayesian non-parametetrics.
- Please check out the code!
  `https://github.com/samer--/ccprism`

Bibliography I

Abdallah, Samer (2017a). "Automatic Differentiation using Constraint Handling Rules in Prolog". In: *arXiv preprint arXiv:1706.00231.*

– (2017b). "Memoisation: Purely, Left-recursively, and with (Continuation Passing) Style". In: *arXiv preprint arXiv:1707.04724.*

– (2017c). "More declarative tabling in Prolog using multi-prompt delimited control". In: *arXiv preprint arXiv:1708.07081v2.*

Desouter, Benoit, Marko Van Dooren, and Tom Schrijvers (2015). "Tabling as a library with delimited control". In: *Theory and Practice of Logic Programming* 15.4-5, pp. 419–433.

Filinski, Andrzej (1999). "Representing layered monads". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, pp. 175–188.

Goodman, Joshua (1998). "Parsing inside-out". PhD thesis. Division of Engineering and Applied Sciences, Harvard University.

– (1999). "Semiring parsing". In: *Computational Linguistics* 25.4, pp. 573–605.

Bibliography II

Huang, Liang and David Chiang (2005). "Better k-best parsing". In: *Proceedings of the Ninth International Workshop on Parsing Technology*. Association for Computational Linguistics, pp. 53–64.

Kiselyov, Oleg and Chung chieh Shan (2008). *Embedded Probabilistic Programming*. Abstract of poster NIPS2008 workshop on Probabilistic Programming: Universal Languages, Systems and Applications.

Klein, Dan and Christopher D Manning (2004). "Parsing and hypergraphs". In: *New developments in parsing technology*. Springer, pp. 351–372.

Kurihara, Kenichi and Taisuke Sato (2006). "Variational Bayesian grammar induction for natural language". In: *Grammatical Inference: Algorithms and Applications*. Springer, pp. 84–96.

Plotkin, Gordon D and Matija Pretnar (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.

Sato, Taisuke (1995). "A statistical learning method for logic programs with distribution semantics". In: *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*. Tokyo, pp. 715–729.

Bibliography III

Sato, Taisuke (2011). "A general MCMC method for Bayesian inference in logic-based probabilistic modeling". In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two.* AAAI Press, pp. 1472–1477.

Sato, Taisuke and Yoshitaka Kameya (1997). "PRISM: a language for symbolic-statistical modeling". In: *Proc. 15th Intl. Joint Conf. on Artifical Intelligence (IJCAI).* Vol. 2, pp. 1330–1335.

– (2000). "A Viterbi-like algorithm and EM learning for statistical abduction". In: *Proceedings of UAI2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support.*

– (2001). "Parameter Learning of Logic Programs for Symbolic-statistical Modeling". In: *Journal of Artificial Intelligence Research* 15, pp. 391–454.

Schrijvers, Tom et al. (2013). "Delimited continuations for Prolog". In: *Theory and Practice of Logic Programming.*

Stuhlmüller, Andreas and Noah D Goodman (2012). "A dynamic programming algorithm for inference in recursive probabilistic programs". In: *arXiv preprint arXiv:1206.3555.*

Bibliography IV

Tolpin, David, Jan-Willem van de Meent, and Frank Wood (2015). "Probabilistic programming in Anglican". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 308–311.