A meta-probabilistic-programming language for bisimulation of probabilistic and non-well-founded type systems

Jonathan Warrell[1,2], Alexey Potapov[2], Adam Vandervorst[2], Ben Goertzel[2]

[1]*Yale University,* [2]*SingularityNET*

Abstract. We introduce a formal meta-language for probabilistic programming, capable of expressing both programs and the type systems in which they are embedded. We are motivated here by the desire to allow an AGI to learn not only relevant knowledge (programs/proofs), but also appropriate ways of reasoning (logics/type systems). We draw on the frameworks of cubical type theory and dependent typed metagraphs to formalize our approach. In doing so, we show that specific constructions within the meta-language can be related via bisimulation (implying path equivalence) to the type systems they correspond. This allows our approach to provide a convenient means of deriving synthetic denotational semantics for various type systems. Particularly, we derive bisimulations for pure type systems (PTS), and probabilistic dependent type systems (PDTS). We discuss further the relationship of PTS to non-well-founded set theory, and demonstrate the feasibility of our approach with an implementation of a bisimulation proof in a Guarded Cubical Type Theory type checker.

1 Introduction

Probabilistic programming offers a fertile ground between logic-based and machine-learning-based approaches to A(G)I. Formalization within type theory offers a rigorous approach to deriving semantics for probabilistic languages [15], and formalization of dependently typed probabilistic languages offers the promise of drawing a tight connection with probabilistic logics of various kinds (e.g. Markov Logic [19], Probabilistic Paraconsistent Logic [7]).

While the exploration of such individual systems is highly important, we might consider more abstractly how to embody general principles for the formation of diverse probabilistic type systems, logics, and programming languages within a single meta-language. Such a language can be considered a meta-theoretical language or logical framework for expressing individual type systems and logics. However, previous frameworks (such as [9]) have not been designed with probabilistic type systems and logics specifically in mind. Here, we outline a formal language, $\mathbb{M}$, designed for such a purpose. This language is intended as a formal model of the MeTTa language, currently being developed as part of the OpenCog project [14,8,16]. The language allows for (probabilistic) reasoning not only about the knowledge embedded in a system, but also about the logic employed by the system itself.

Our approach may also be seen in relation to recent methods to derive synthetic denotational semantics for logical systems using guarded cubical type theory (GCTT)

[18,11]. Such approaches are particularly promising, offering as they do a unified approach to deriving semantics for recursive datatypes as final co-algebras of appropriate functors in the context of a formulation of univalent type theory with a fully computational semantics. We draw on methods from [10] to formalize our approach in this context. This allows us to rigorously define the relationship between an object-language and its expression in our meta-language as one of bisimulation, corresponding to path equivalence in GCTT. We further show how dependently typed metagraphs can be formalized in GCTT as the basis for our framework [6,12], and how this leads to systems embedding natural type-theoretic equivalents of non-well-founded sets.

We begin by developing a general framework for representing metagraphs in GCTT, before outlining how the final co-algebra of a labeled transition system over this recursive datatype can be used to model our meta-language. We then derive bisimulations for various object-languages in our system, including simply typed (and untyped) lambda calulus, pure type systems, and probabilistic dependent type systems, hence deriving synthetic denotational semantics for these systems. Finally, we demonstrate the feasibility of our approach with an implementation of a bisimulation proof for a small-scale type system in a Guarded Cubical Type Theory type checker [4], before concluding with a discussion.

2 Labeled metagraphs as a guarded recursive datatype

We begin by defining a recursive datatype for typed metagraphs ($\mathcal{M}_{(\mathcal{T},\mathcal{L},\preceq_T)}$) using guarded cubical type theory. Here, $\mathcal{T}, \mathcal{L}$ are types of type-symbols and edge labels respectively, and $\preceq_T$: $\mathcal{T} \times \mathcal{T} \to \mathbb{B}$ is a partial order on type-symbols. The recursive datatype is defined as the final co-algebra of the functor $\mathcal{M}'_{(\mathcal{T},\mathcal{L},\preceq_T)}(A)$, which when applied to type A returns the following datatype (letting Δ stand for the assumptions $\mathcal{L}, \mathcal{T}, A : \mathcal{U}_0$; the ϵ, edge, and $\mathrm{connect}$ constructors used here follow the approach of [12] and [6]):

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \mathcal{M}'_{(\mathcal{T},\mathcal{L})}(A)}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \epsilon : \mathcal{M}'_{(\mathcal{T},\mathcal{L})}(A)}$$

$$\frac{\Gamma \vdash \Delta, n : \mathbb{N}, t_0 : \mathcal{T}, t : \mathrm{Vec}(n, \mathcal{T}), l_0 : \mathcal{L}}{\Gamma \vdash \mathrm{edge}(n, t_0, l_0, t) : \mathcal{M}'_{(\mathcal{T},\mathcal{L})}(A)}$$

$$\frac{\Gamma \vdash \Delta, a_1, a_2 : A, t_0 : \mathcal{T}, l_0 : \mathcal{L}, q : \mathbb{N} \to \mathbb{N}_{0,\infty}}{\Gamma \vdash \mathrm{connect}(a_1, a_2, t_0, l_0, q) : \mathcal{M}'_{(\mathcal{T},\mathcal{L})}(A)} \tag{1}$$

where $\mathrm{Vec}(n, A)$ is the type of vectors over A of length n, and $\mathbb{N}_{0,\infty}$ is $\mathbb{N}$ extended with 0 and ∞. We note that for notational convenience, we do not explicitly include

target labels/indices in the definition of $\mathcal{M}'_{(\mathcal{T},\mathcal{L})}(A)$ above (in contrast to [6], where $\mathcal{L}$ refers to target indices and $\mathcal{V}$ is used for edge values). If explicit indices are required to identify target 'levels', these may be included by letting $\mathcal{L} = \mathcal{L}_0 \times \sum_n \mathrm{Vec}(n, \mathbb{N})$, so that each edge label is paired with a vector of target indices. $\mathcal{M}_{(\mathcal{T},\mathcal{L},\preceq_T)}$ is then defined as a final fixed-point of $\mathcal{M}'_{(\mathcal{T},\mathcal{L})}$, such that a set of constraints are satisfied:

$$\mathcal{M}_{(\mathcal{T},\mathcal{L},\preceq_T)} = \sum M : \nu(\mathcal{M}'_{(\mathcal{T},\mathcal{L})}).C(M, \preceq_T) \tag{2}$$

where $C(M, \preceq_T)$ represents the constraints:

$$\begin{aligned} C(M, \preceq_T) = \forall n_1, n_2 : \mathbb{N}, t_1, t_2 : \mathcal{T} \\ f(M, n_1) = t_1 \wedge \\ f(M, n_2) = t_2 \wedge \\ q'_M(n_1) = n_2 \Rightarrow t_1 \preceq_T t_2 \end{aligned} \tag{3}$$

Here, $f(M, n)$ represents a function, which for metagraph M returns the type of its n'th edge or target. Specifically, when M is of the form $\mathrm{edge}(n, t_0, l_0, t)$, $f(M, 0)$ is the type of the edge, and $f(M, n > 0)$ is the type of the n'th target, and when M is of the form $\mathrm{connect}(a_1, a_2, t_0, l_0, q)$, $f(M, 0)$ is the type of the whole metagraph, while the types of the edges/targets of a_1 and a_2 are interleaved when evaluating $f(M, n > 0)$ for odd/even values of n respectively. Further, the function $q'_M : \mathbb{N} \to \mathbb{N}_{0,\infty}$ is recursively defined on $\nu(\mathcal{M}'_{(\mathcal{T},\mathcal{L})})$ (via the q function in the $\mathrm{connect}$ constructor of Eq. 1) to indicate that the n_1'th target of M is connected to the n_2'th edge/target of M, whenever $q(n_1) = n_2$, with $n_2 = \infty$ indicating that the target has no connection. $C(M, \preceq_T)$ thus provides a set of constraints that ensure the connections in a metagraph respect the $\preceq_T$ relation; further constraints are needed to ensure for instance that targets receive input from only one other target (as may be appropriate for some metagraphs). Further, $\nu = \mathrm{fix}\, X.F(\triangleright(\alpha : \mathbb{T}).X[\alpha]))$ is the guarded fixed-point operator [10]. By [10], Prop. 3.2, $\mathcal{M}_{(\mathcal{T},\mathcal{L},\preceq_T)}$ is both a subset of the initial algebra and final coalgebra of $\mathcal{M}'_{(\mathcal{T},\mathcal{L})} \circ \triangleright$. Finally, we note that our $\mathrm{connect}$ constructor corresponds to $\mathrm{Connect}_Q$ in [6], and the Union constructor is simply $\mathrm{connect}$ with $q(n) = \infty$ for all n (meaning that no new connections are added).

We briefly give some examples of typed metagraphs. For convenience, we set $\mathcal{L} = \{\mathrm{null}\}$, and $\mathcal{T} = \{A, B, C, D, \top\}$, with $\preceq_T$ the identity relation along with $t \preceq_T \top$ for all t. In our first example, we can construct metagraphs $X = \mathrm{edge}(3, A, \mathrm{null}, [D, B, C])$, and $Y = \mathrm{edge}(2, B, \mathrm{null}, [D, A])$. Then, a combined graph can be constructed as $Z' = \mathrm{connect}(X, Y, \top, \mathrm{null}, \{(1,1),(2,0)\})$, $Z'' = \mathrm{connect}(Y, X, \top, \mathrm{null}, \{(1,1),(2,0)\})$, $Z''' = \mathrm{connect}(Z', Z'', \top, \mathrm{null}, \{\})$, $Z = \mathrm{connect}(X, Z''', C, \mathrm{null}, \{(3,0)\})$. The entire metagraph is shown in Fig. 1A. We note that, in general, any metagraph with a finite number of edges and targets can be represented by a term in the initial algebra of $\mathcal{M}'_{(\mathcal{T},\mathcal{L})}$ (as is Z). Some graphs, however, may be conveniently be represented also by terms in the final coalgebra. Consider for instance Fig. 1B. Here, we may define $X'' = \mathrm{edge}(3, \top, \mathrm{null}, [B, B, A])$ and $X' = \mathrm{connect}(X'', X'', A, \mathrm{null}, \{(1,2),(2,1),(3,0)\})$, representing X' by a term in the initial algebra (suppressing visualization of the X'' subgraph). Alternatively, we may define $X'_{co} = \mathrm{connect}(\mathrm{edge}(3, A, \mathrm{null}, [B, B, A]),$

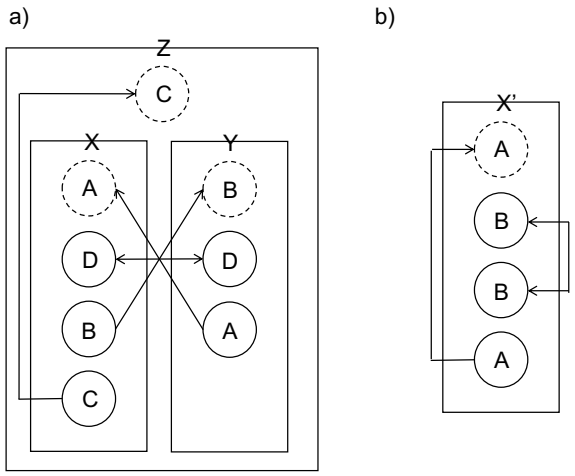

Fig. 1. Typed metagraph examples. Boxes show metagraphs, which may be single edges (containing no further boxes) or include several edges. Solid circles edge target types and dotted circles show metagraph types. Arrows show target-target or target-edge connections. Metagraph letter names are shown on the box of the metagraph to which they refer in the text.

$X'_{co}, A, \text{null}, \{(1,2),(2,1),(3,0)\})$, which implicitly determines a term in the coalgebra as a solution to the recursive equation.

3 $\mathbb{M}$ as the final coalgebra of a labeled transition system

We define the formal meta-probabilistic-programming language, $\mathbb{M}$, as a labeled transition system over typed metagraphs. Here, we are interested in typed metagraphs with a particular form. Specifically, we begin by defining $\mathcal{T}$ by the abstract syntax:

$$\begin{aligned}\mathcal{T} ::= & \; t_n \mid \mathcal{T} \to \mathcal{T} \mid \prod a : \mathcal{T}.\mathcal{M}_{\mathbb{M}} \mid \\ & \text{Eq}(\mathcal{T}, \mathcal{M}_{\mathbb{M}}, \mathcal{M}_{\mathbb{M}}) \mid \mathcal{T} \cup \mathcal{T} \mid \mathcal{T} \cap \mathcal{T} \mid \\ & \text{Type} \mid \top_{\text{Type}} \mid \top \mid \mathcal{J} \mid \mathcal{X}\end{aligned} \tag{4}$$

These syntactic constructions represent base-level types, function types, dependent types, equality types, type unions and intersections, a base universe of small types, the union of all small types, the union of all types, judgments and execution states respectively. Notice also that in Eq. 4, $\mathcal{T}$ is defined by mutual recursion with the type $\mathcal{M}_{\mathbb{M}}$, defined in Eq. 6. We then define $\mathcal{L}$ as $\mathcal{L} = \mathcal{S} \cup \mathcal{V} \cup \mathcal{K} \cup \mathcal{T} \times \mathbb{N}$. Notice that $\mathcal{L}$ includes $\mathcal{T}$, so that types may simultaneously serve as labels. Further, $\mathcal{S} = \{s_1, s_2, ...\}$ and $\mathcal{V} = \{v_1, v_2, ...\}$ denote collections of symbols and variables respectively, and $\mathcal{K}$ is a special set of $\mathbb{M}$ keywords/key-symbols:

$$\mathcal{K} = \{:, \preceq, =, \to, \text{Eq}, \text{fun-app}, \text{transform}, @, \dagger\} \tag{5}$$

Further, $\mathcal{L}$ includes an edge-specific identifier $\mathbb{N}$ to deduplicate edges which are identical in other respects.

The state of an $\mathbb{M}$ program is represented by a typed metagraph in the following space:

$$\mathcal{M}_{\mathbb{M}} = \sum \preceq_T : (\mathcal{T} \times \mathcal{T} \to \mathbb{B}). \sum M : \mathcal{M}_{(\mathcal{T},\mathcal{L},\preceq_T)}. C_{\mathbb{M}}(M, \preceq_T) \tag{6}$$

Hence, this is the space of all metagraphs over $\mathcal{L}$ and $\mathcal{T}$, with a varying $\preceq_T$ relation, where $C_{\mathbb{M}}(M, \preceq_T)$ represents a set of '$\mathbb{M}$-specific constraints' on the structure of the metagraph (to be outlined below). This state represents the *Atomspace* of the program, and the subgraphs of the Atomspace are the individual atoms (as in MeTTa, see [14,8]). We note that, since $\mathbb{M}$ serves both as a language for defining programs and type-systems within which these programs are embedded, the atoms may represent base-level propositions and programs (expressions), as well as judgments and computational state information, as reflected by their types. The $\mathbb{M}$-specific constraints, $C_{\mathbb{M}}(M, \preceq_T)$, determine the interaction of the keywords/key-symbols with the type system:

$$\begin{aligned}
&\forall m \in M. \exists n, n_1 : \mathbb{N}. \\
&\quad m = \mathrm{edge}(2, \mathcal{J}, (:, n), [\top_{\mathrm{Type}}\ \top]) \vee \\
&\quad m = \mathrm{edge}(2, \mathcal{J}, (\preceq, n), [\mathrm{Type}\ \ \mathrm{Type}]) \wedge \\
&\qquad (m_M[1] = \mathrm{edge}(0, \mathrm{Type}, (t_{n_1}, 0), []) \wedge \\
&\qquad m_M[2] = \mathrm{edge}(0, \mathrm{Type}, (t_{n_2}, 0), []) \Rightarrow (t_{n_1} \preceq t_{n_2})) \vee \\
&\quad m = \mathrm{edge}(2, \mathcal{J}, (=, n), [\top_{\mathrm{Type}}\ \top_{\mathrm{Type}}]) \vee \\
&\quad m = \mathrm{edge}(2, \mathrm{Type}, (\to, n), [\mathrm{Type}\ \ \mathrm{Type}]) \wedge \\
&\qquad ((m_1)_M[2] = m \wedge l(m_1) = (:, n_1) \wedge t(m_M[1]) = A \wedge \\
&\qquad t(m_M[2]) = B \Rightarrow t((m_1)_M[1]) \preceq A \to B) \vee \\
&\quad m = \mathrm{edge}(2, \mathrm{Type}, (\to, n), [\mathrm{Type}\ \ \top]) \wedge \\
&\qquad ((m_1)_M[2] = m \wedge l(m_1) = (:, n_1) \wedge t(m_M[1]) = A \wedge \\
&\qquad m_M[2] = m_2 \Rightarrow t((m_1)_M[1]) \preceq \prod a : A. m_2) \vee \\
&\quad m = \mathrm{edge}(3, \mathrm{Type}, (\mathrm{Eq}, n), [\mathrm{Type}\ \ t_{n_1}\ t_{n_1}]) \wedge \\
&\qquad m_M[1] = \mathrm{edge}(0, \mathrm{Type}, (t_{n_1}, 0), []) \wedge \\
&\qquad ((m_1)_M[2] = m \wedge l(m_1) = (\mathrm{Eq}, n_1) \wedge l(m_M[1]) = (T, n_1) \wedge \\
&\qquad m_M[2] = A \wedge m_M[3] = B \Rightarrow t((m_1)_M[1]) = \mathrm{Eq}(T, A, B)) \vee \\
&\quad m = \mathrm{edge}(2, \top, (\mathrm{transform}, n), [\top\ \top]) \vee \\
&\quad m = \mathrm{edge}(1, \mathcal{X}, (@, n), [\top]) \wedge \\
&\quad m = \mathrm{edge}(1, \mathcal{X}, (\dagger, 0), [\top]) \wedge \\
&\quad m = \mathrm{edge}(0, \top, (\mathcal{S} \cup \mathcal{V} \cup \mathcal{T}, n), []) \wedge \\
&\quad m = \mathrm{edge}(2, B, (\text{fun-app}, n), [A \to B'\ A]) \wedge B' \preceq B \vee \\
&\quad m = \mathrm{edge}(2, B, (\text{fun-app}, n), [\prod a : A. m_1\ A]) \wedge \\
&\qquad m_1[a = m_M[1]] \preceq B \vee
\end{aligned}$$

$$
\begin{aligned}
m &= \mathrm{connect}(\_,\_,\_,\_,\_) \wedge \\
\forall n, n_1, n_2 &: \mathbb{N}. t_n \preceq \top \wedge \\
s_n &: \top_{\mathrm{Type}} \vee s_n : \mathrm{Type} \wedge \\
v_n &: \top_{\mathrm{Type}} \vee v_n : \mathrm{Type} \wedge \\
t_n &: \mathrm{Type} \wedge \\
(t_{n_1} \preceq t_{n_2} \wedge t_{n_2} \preceq t_n &\Rightarrow t_{n_1} \preceq t_n) \wedge \\
t_n &\preceq t_n \cup t_{n_1} \wedge \\
t_n \cap t_{n_1} &\preceq t_n
\end{aligned} \tag{7}
$$

where the notation $m_M[n]$ denotes the n'th target of subgraph m in metagraph M, $t[m]$ and $l[m]$ denote the type and label of metagraph m respectively, and we write $a : A$ as shorthand for 'there exits an :-edge in M connecting a and A'. We note that, for convenience, the above formulation does not include some constructions that may be appropriate in a full implementation, but can be derived from others. For instance, tuples can be constructed by introducing a dependent function $\mathrm{tuple} : \prod A, B : \mathrm{Type}.A \to B \to \mathrm{Type}$. The left and right projection functions are then defined by $\pi_1(\mathrm{tuple}(A, B, a, b)) = a$ and $\pi_2(\mathrm{tuple}(A, B, a, b)) = b$. Dependent sums can likewise be defined as dependent tuples, $\mathrm{tuple}' : \prod A : \mathrm{Type}. \prod B : (A \to \mathrm{Type}). \prod a : A.B(a) \to \mathrm{Type}$.

3.1 Labeled transition system based on metagraph rewriting

In guarded cubical type theory, a guarded labeled transition system (GLTS) may be defined via a state-space X, a space of actions A, and a function mapping states to sets of (action,state) pairs, $f : X \to P_{\text{fin}}(A \times \triangleright X))$, where P_{fin} is the finite powerset functor. The space of all processes, or runs of the GLTS may the be defined as the final coalgebra of the following functor: $\mathrm{Proc} = \mathrm{fix}\, X.P_{\text{fin}}(A \times \triangleright(\alpha : \mathbb{T}).X[\alpha]))$ (see [10]). In order to characterize the process of evaluation in $\mathbb{M}$, we characterize the computational dynamics of $\mathbb{M}$ via a GLTS. Here, the state space is the space of all $\mathbb{M}$ metgraphs, $X = \mathcal{M}_{\mathbb{M}}$. The actions are specified by single pushout (SPO) rewriting rules, or sequences of such rules. We therefore introduce the type, $\mathcal{A}' = \mathcal{M}_{\mathbb{M}}^{(L,R)} \times \hom_p(\mathcal{M}_{\mathbb{M}})$, whose values (M', ϕ) consist of a $\mathbb{M}$ metagraph whose label set is $\mathcal{L}' = \mathcal{L} \times \{L, R, LR\} \times \{[], *, **\}$, i.e. identical to above, but with L and R labels added to each edge to indicate its membership of the left or right-hand side of the rule (notice that these may overlap), * and ** to indicate the input and output nodes of the rule (see below), and ϕ, a partial metagraph homomorphism between the L and R metagraphs of M' (defining a partial metagraph homomorphism as in [8]). Since we wish to allow sequences of rewrite rules as actions, we define the full action space to be $\mathcal{A} = \sum n : \mathbb{N}. \mathrm{Vec}(n, \mathcal{A}')$, and write the members of $\mathcal{A}$ as $a_1 \circ a_2 \circ ... \circ a_n$, where $a_{1...n} : \mathcal{A}'$. The dynamics are then defined (via f) by mapping a given metagraph state M_1 to the set of all pairs (A, M_2) such that M_2 results from an application of action A to M_1. For individual rewrite rules $a \in \mathcal{A}'$, their action is determined via a partial homomorphism between a and M_1. We note that, when there are no partial homomorphisms between a and M_1, or when the rewrite rule produces an invalid $\mathbb{M}$ graph, we set $M_2 = M_1$. Further, we note that the update may change the $\preceq$ relation, for instance by introducing an edge of the form $t_1 \preceq t_2$.

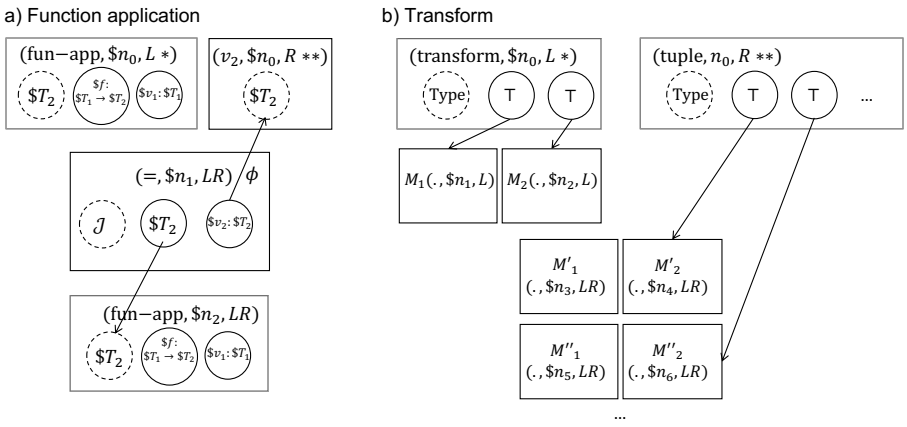

Fig. 2. Metagraph rewriting rules. Notation as in Fig. 1. Subgraphs involving only one variable are not shown explicitly, but notated directly on the targets they are connected to. See Eqs. 8 and 9 for explicit expressions for the graphs.

3.2 $\mathbb{M}$-interpretation as metagraph dynamics

We can now describe interpretation in $\mathbb{M}$ via the GLTS defined above. To do so, we map specific symbols/edges in the metagraph to actions in $\mathcal{A}$ (corresponding to the grounding domain F in [8]). Specifically, edges carrying symbols of a function type, $A \to B$, dependent product type, $\prod a : A.B$, or the $\mathrm{transform}$ symbol, are mapped to specific forms of rewrite rule, as specified below. All other edges are mapped to the null transform. Fig. 2 specifies the general forms of the rewrite rules for function application, and transform rules (we note the $\mathrm{transform}$ is equivalent to the 2-argument match keyword/function in the current version of the MeTTa language, see [16]). The dependent product rule is identical to Fig. 2a, with $A \to B$ replaced with $\prod a : A.m_1$ For explicitness, we give these below also in equational form. We note that, for convenience variable names are denoted using \$, although these should be ultimately mapped to the names $v_1, v_2, ...$

$$
\begin{aligned}
R^1_{\text{fun-app}} &= \mathrm{edge}(2, \$T_2, (\text{fun-app}, \$n_0, L), [\$T_1 \to \$T_2\ \$T_1]) \\
R^2_{\text{fun-app}} &= \mathrm{edge}(2, \mathcal{J}, (=, \$n_1, LR), [\$T_2\ \$T_2]) \\
R^3_{\text{fun-app}} &= \mathrm{edge}(2, \$T_2, (\text{fun-app}, \$n_2, LR), [\$T_1 \to \$T_2\ \$T_1]) \\
R^4_{\text{fun-app}} &= \mathrm{edge}(0, \$T_1 \to \$T_2, (\$f, \$n_3, LR), []) \\
R^5_{\text{fun-app}} &= \mathrm{edge}(0, \$T_1, (\$v_1, \$n_4, LR), []) \\
R^6_{\text{fun-app}} &= \mathrm{edge}(0, \$T_2, (\$v_2, \$n_5, LR **), []) \\
R^7_{\text{fun-app}} &= \mathrm{connect}(\mathrm{connect}(R^1_{\text{fun-app}}, R^4_{\text{fun-app}}, \top, \mathrm{null}, \{(1,0)\})), \\
&\qquad R^5_{\text{fun-app}}, \top, (\mathrm{null}, \mathrm{null}, *), \{(5,0)\}) \\
R^8_{\text{fun-app}} &= \mathrm{connect}(\mathrm{connect}(R^3_{\text{fun-app}}, R^4_{\text{fun-app}}, \top, \mathrm{null}, \{(1,0)\})),
\end{aligned}
$$

$$
\begin{aligned}
& R^5_{\text{fun-app}}, \top, \text{null}, \{(5,0)\}) \\
R^9_{\text{fun-app}} &= \text{connect}(\text{connect}(R^2_{\text{fun-app}}, R^3_{\text{fun-app}}, \top, \text{null}, \{(1,0)\})), \\
& R^5_{\text{fun-app}}, \top, \text{null}, \{(5,0)\}) \\
R_{\text{fun-app}} &= R^7_{\text{fun-app}} \cup R^8_{\text{fun-app}} \cup R^9_{\text{fun-app}}
\end{aligned} \tag{8}
$$

$$
\begin{aligned}
R^1_{\text{transform}} &= \text{edge}(2, \text{Type}, (\text{transform}, \$n_0, L), [\top\ \top]) \\
R^2_{\text{transform}} &= \text{connect}(\text{connect}(R^1_{\text{transform}}, \$M_1, \top, \text{null}, \{(1,0)\})), \\
& \quad \$M_2, \top, (\text{null}, \text{null}, L*), \{(5,0)\}) \\
R^3_{\text{transform}} &= \text{edge}(2, \text{Type}, (\text{tuple}, \$n_0, R**), [\top\ \top]) \\
R^4_{\text{transform}} &= \$M'_1 \cup \$M''_1 \cup \$M'_2 \cup \$M''_2 \cup \text{edge}(0, \top, (\text{null}, \text{null}, LR), []) \\
R^5_{\text{transform}} &= \text{connect}(R^3_{\text{transform}}, R^4_{\text{transform}}, \top, (\text{null}, \text{null}, \text{null}), \{(1,1),(2,2)\})) \\
R_{\text{transform}} &= R^2_{\text{transform}} \cup R^5_{\text{transform}}
\end{aligned} \tag{9}
$$

In Eq. 9, M'_1 and M'_2 denote metagraphs isomorphic to M_1 and M_2, using a disjoint set of variables, while M''_1 and M''_2 are defined similarly, with variables disjoint to the previous subsets. The rule in Eq. 9 is defined so as to return a 2-tuple of matches; in general, the size of the tuple returned should be large enough to allow for any number of matches (i.e the number of nodes in M), and if the number of matches is less than this, it will be padded with null values.

fun-app nodes. For a given annotated fun-app node, i.e. $\text{connect}(@, F, \text{null}, \{(1,0)\}))$, where $@ = \text{edge}(1, \mathcal{X}, (@, n), [\top])$ and F is a graph consisting of a target fun-app node and its two arguments, the full rewrite rule rewrite_F is found by forming a metagraph homomorphism between $R^7_{\text{fun-app}}$ (labeled by $*$ as the input of the rule), and F, replacing the variables in $R_{\text{fun-app}}$ by their values in F. The resulting graph is denoted $R_{\text{fun-app}}(F)$. The rule rewrite_F is then defined by the subgraphs $L = \text{connect}(\$M_0, \text{connect}(@, l_{\text{fun-app}}(F), \text{null}, \{(1,0)\})), \text{null}, \{((n_1, m_1), (n_2, m_1), ...\})$, $R = \text{connect}((\$M_0, r_{\text{fun-app}}(F), \text{null}, \{((n_1, m), (n_2, m), ...\})$, where M_0 is the graph of all nodes in M targeting F, m_1 is the index of the fun-app node in F, m_2 is the index of the $**$ output node in $r_{\text{fun-app}}(F)$, and ϕ is defined by the partial homomorphism consisting of the identity map on all nodes labeled LR.

transform nodes. For a given annotated transform node, the full rewrite rule is defined similarly. Hence, for $\text{connect}(@, F, \text{null}, \{(1,0)\}))$, where $@ = \text{edge}(1, \mathcal{X}, (@, n), [\top])$ and F is a graph consisting of a target transform node and its two arguments, the full rewrite rule rewrite_F is found by forming a metagraph homomorphism between $R^2_{\text{transform}}$ (labeled by $*$ as the input of the rule), and F, replacing the variables in $R_{\text{transform}}$ by their values in F. The resulting graph is denoted $R_{\text{transform}}(F)$. The rule rewrite_F is then defined by the subgraphs $L = \text{connect}(\$M_0, \text{connect}(@, l_{\text{transform}}(F), \text{null}, \{(1,0)\})), \text{null}, \{((n_1, m_1), (n_2, m_1), ...\})$, $R = \text{connect}((\$M_0, r_{\text{transform}}(F), \text{null}, \{((n_1, m), (n_2, m), ...\})$, where M_0 is the graph of all nodes in M targeting F, m_1 is the index of the transform node in F, m_2 is the index of the $**$ output node in $r_{\text{transform}}(F)$, and ϕ is defined by the partial homomorphism consisting of the identity map on all nodes labeled LR.

$\mathbb{M}$-evaluation. The above provides groundings for activated nodes in a metagraph; as noted, nodes not of the form above result in a null update. Evaluation in $\mathbb{M}$ involves repeatedly updating the current pointed metgraph according to the grounding of the node currently pointed to. The conditions in Eq. 7 imply there will be at most one edge labeled with † in a metagraph, whose target F specifies the rule by which the graph is updated. This is expressed via the single partial function, $\mathrm{update} : \mathcal{M}_{\mathbb{M}} \to \mathcal{M}_{\mathbb{M}}$. The action of update is determined by the form of F. If F is not an activated subgraph, i.e. it is not the target of an @-edge, the action update cannot be applied (i.e. evaluation halts). If however F is the target of an @-edge, update first checks if F itself has any activated targets. If so, then update simply applies a graph rewrite which moves the pointer † to the first such activated target (in the ordering of the edge). If not, update applies $\mathrm{rewrite}_F$, which automatically ensures that the update will finish with † pointing to the output subgraph, labeled $**$. These dynamics define a reduced GLTS, with $X = \mathcal{M}_{\mathbb{M}}$, $A = \{\mathrm{update}\}$, and $f(M) = \{(\mathrm{update}, M' | \, \mathrm{update}(M) = M')\}$. Note that there may be multiple M''s for which $\mathrm{update}(M) = M'$ if $\mathrm{rewrite}_F$ for a fun-app node is non-deterministic. Processes are defined by the fixed point $\mathrm{Proc} = \nu(P_{\mathrm{fin}}(A \times \triangleright X)))$. Normal forms of $\mathcal{M}_{\mathbb{M}}$ are metagraphs for which update cannot be applied (i.e. their grounding is null). Processes which reach a normal form are said to be terminating, and the initial expression of the process is said to evaluate to the normal form reached. Alternatively, certain expressions may not reach a normal form, resulting instead in a non-terminating computation.

4 Bisimulation of type systems in $\mathbb{M}$

As described in [10], in guarded cubical type theory, a bisimulation $R : X \to X \to U$ for the GLTS (X, A, f) may be defined via the following dependent type:

$$\begin{aligned}
\mathrm{isGLTSBisim}_f\, R = \prod x, y : X.R(x,y) \to \\
(\prod x' : \triangleright X. \prod a : A.(a,x') \in f(x) \to \exists y' : \triangleright X. \prod a : A. \\
(a,y') \in f(y) \times \triangleright(\alpha : \mathbb{T}).R(x'[\alpha])(y'[\alpha])) \times \\
(\prod y' : \triangleright X. \prod a : A.(a,y') \in f(y) \to \exists x' : \triangleright X. \prod a : A. \\
(a,x') \in f(x) \times \triangleright(\alpha : \mathbb{T}).R(x'[\alpha])(y'[\alpha])). \qquad (10)
\end{aligned}$$

As shown in [10], this type is equivalent to the path type over the recursive data type of processes defined by the GLTS, $\mathrm{Proc} = \mathrm{fix}\, X.P_{\mathrm{fin}}(A \times \triangleright(\alpha : \mathbb{T}).X[\alpha]))$. We may further define a bisimulation $R_2 : X_1 \to X_2 \to U$ between two GLTS's over a common action space, (X_1, A, f_1) and (X_2, A, f_2) via a bisimulation over their coproduct (see [1]):

$$\begin{aligned}
\mathrm{is2GLTSBisim}_{f_1,f_2}\, R_2 = \mathrm{isGLTSBisim}_{f_1+f_2}\, R_2' \times \forall x_1 : X_1.\exists x_2 : X_2.R_2(x_1,x_2) \times \\
\forall x_2 : X_2.\exists x_1 : X_2.R_1(x_1,x_2) \qquad (11)
\end{aligned}$$

where $R_2' : (X_1 + X_2) \to (X_1 + X_2) \to U$, $R_2'((a,x),(b,y)) = R(x,y)$ when $a = 1 \wedge b = 2$, $R_2'(x,y) = \bot$ otherwise, and $f_1 + f_2 : (X_1 + X_2) \to \mathcal{P}(A \times (X_1 + X_2))$

defined similarly. Since $R_2(x_1, x_2)$ contains at least one matching element for each x_1 and x_2, we may extract functions $g_1 : X_1 \to X_2$ and $g_2 : X_2 \to X_1$ as subsets of R_2, where an element in the codomain of each is chosen arbitrarily when there are multiple matches in R_2. Since bisimulation corresponds to path-equivalence for elements of each type, g_1 and g_2, we can choose π_1 and π_2 such that $g_1 \circ g_2 \circ \pi_1 = i_1$ and $g_2 \circ g_1 \circ \pi_2 = i_2$, where i_1 and i_2 are the identity on X_1 and X_2 respectively, and $\pi_1(x) = x' \Rightarrow \exists p : \mathrm{Path}_{X_1}(x, x')$, $\pi_2(x) = x' \Rightarrow \exists p : \mathrm{Path}_{X_2}(x, x')$. Hence, (g_1, g_2) is an equivalence between the recursive process types Proc_1 and Proc_2 of the two GLTS's, meaning that $\mathrm{Path}_U(\mathrm{Proc}_1, \mathrm{Proc}_2)$ is inhabited by univalence.

For a given type system, its computational content may be modeled by a GLTS by setting X to be the type of expressions in the system, A to contain an update action along with 'actions' corresponding to the judgmental and syntactic relations between expressions (e.g. is-of-type, is-of-subtype, is-a-body-of-lambda-term, and their opposite relations), and f to be the relation over expressions corresponding to the reduction relation in the system for the action update (for instance β-reduction). To show that $\mathbb{M}$ can be used as a metalanguage for a given type system, we thus show that there is a bisimulation between $\mathbb{M}$ with a specific form of Atomspace (i.e. containing specific atoms and/or additional constraints to those of Eq. 7), along with an expanded action space to incorporate the typing and syntactic relations relevant to the specific system, and the GLTS corresponding to computation in the target type system; hence the process spaces induced by the two systems are equivalent. Below, we sketch how this can be achieved for three type systems of interest, focusing on the how the computational dynamics of the update rule correspond to reduction in the target system (the typing and syntactic relations in each system straightforwardly correspond in $\mathbb{M}$ to the inbuilt typing relation and relationships definable in terms of submetagraph composition respectively).

4.1 Simply typed lambda calculus

The syntax for the simply typed lambda calculus may be defined via mutually recursive definitions of variable, type and expression datatypes:

$$\begin{aligned} \mathcal{V} &::= v_n \\ \mathcal{T} &::= t_n \mid \mathcal{T} \to \mathcal{T} \\ \mathcal{E} &::= \mathcal{V} \mid (\mathcal{E}\ \mathcal{E}) \mid \lambda v_n : \mathcal{T}.\mathcal{E} \end{aligned} \tag{12}$$

We refrain from explicitly stating the rules for type assignment as can be found in [2], which determine a typing relation $\_ : \_$ between $\mathcal{E}$ and $\mathcal{T}$ given a context Γ, which can be modeled as a partial map from $\mathcal{V}$ to $\mathcal{T}$. Together, these determine a set of valid expressions, $\mathcal{E}_{(\_:\_,\Gamma)}$, and the computational dynamics is defined by the β-reduction relation over this type:

$$((\lambda v_{n_1} : t_{n_2}.e_{n_3})\ e_{n_4}) \to_\beta e_{n_3}[v_{n_1}/e_{n_4}] \tag{13}$$

where $a[b/c]$ denotes substitution of b for c in a, where any bound variables in c are renamed so as not to clash with bound variables in a.

To simulate the simply typed lambda calculus in $\mathbb{M}$, we restrict the $\mathbb{M}$ atomspace to include only metagraphs labeled with types using the restricted type syntax of Eq. 12, and including only keywords/symbols $\{:, =, \rightarrow, \text{fun-app}, @, \dagger\}$. Then, we add the following constraint to those of Eq. 7:

$$\forall m \in M.l(m) = (:, n_1) \Rightarrow (m_M[1] \in \mathcal{S} \vee \mathcal{V}) \wedge m_M[2] \in \mathcal{T} \tag{14}$$

Hence, all typing relations are between symbols or variables (representing global and local variables respectively) and types. The context Γ is then represented by an atomspace consisting of a set of : edges between symbols and types. A given lambda expression $e = \lambda x : t_1.e'$, where $e' : t_2$ is then simulated by choosing an unused symbol, $f_e \in \mathcal{S}$, and introducing the following atoms to atomspace:

$$\begin{aligned} &(:\ f_e\ (\rightarrow\ t_1\ t_2)) \\ &(=\ (f_e\ \$x)\ m_{e'}) \end{aligned} \tag{15}$$

where $m_{e'}$ is the metagraph corresponding to expression e' (we note that Eq. 15 defines a combinator corresponding to the lambda term e). With the atomspace so specified, reduction of an expression e in context Γ in the simply typed lambda calculus corresponds to repeated application of update to the pointed atomspace containing Γ and m_e, with $@$ edges attached to all function application nodes, and the $\dagger$ pointing to m_e. The computation terminates with $\dagger$ pointing to the normal form of e. The required bisimulation thus involves pairing tuples (Γ, e) in the simply typed lambda calculus with their corresponding pointed atomspaces in $\mathbb{M}$. We note further that the untyped lambda calculus can be defined by simply removing $\mathcal{T}$ from the syntax in Eq. 12, and letting lambda expressions take the form $\lambda v_n.\mathcal{E}$. All members of $\mathcal{E}.$ are considered legal expressions, and the $\mathbb{M}$ bisimulation is achieved by converting all type symbols to $\top_{\text{Type}}$, hence treating $\top_{\text{Type}}$ as a Scott domain.

4.2 Pure Type Systems

In a pure type system (PTS, [2]), types and terms are not distinguished syntactically. PTS expressions follow the syntax:

$$\begin{aligned} &\mathcal{V} ::= v_n \\ &\mathcal{C} ::= s_{1\ldots N} \\ &\mathcal{E} ::= \mathcal{V}\ \mid\ \mathcal{C}\ \mid\ (\mathcal{E}\ \mathcal{E})\ \mid\ \lambda v_n : \mathcal{E}.\mathcal{E}\ \mid\ \prod v_n : \mathcal{E}.\mathcal{E} \end{aligned} \tag{16}$$

Here, $\mathcal{C}$ is a set of constant symbols, which in a PTS are used to represent *sorts*. The typing relation : for a PTS is defined via a set of axioms and rules. The former consist of a set of judgements $\mathcal{A} = \{s_m : s_n | (m, n) \in A \subset N \times N\}$, and the latter a set of triplets $\mathcal{R} = \{(s_l, s_m, s_n) | (l, m, n) \in R \subset N \times N \times N\}$. The typing rules for a PTS are identical to the typed lambda calculus, except for the introduction rule for dependent products, which takes the form:

$$\frac{\Gamma \vdash A : s_l \quad \Gamma, A : s_l \vdash B : s_m \quad (s_l, s_m, s_n) \in \mathcal{R}}{\Gamma \vdash (\prod x : A.B) : s_n}$$

The legal expressions then consist of the sorts, and any expression that can be typed in a context Γ, consisting of multiple typing judgments $e_1 : e_2$. The β-reduction relation is established identically to the simple lambda calculus above. Notice that there is no restriction on the form of $\mathcal{A}$ and $\mathcal{R}$; hence the typing relation : may be arbitrary between sorts (and hence may contain cycles), while the dependent product (i.e. dependent function types) may live in arbitrary sorts with respect to their inputs.

To simulate a PTS in $\mathbb{M}$, we select a collection of fixed types $t_1 ... t_N$ to represent the sorts. We then add edges of the following forms to atomspace:

$$
\begin{aligned}
&(:\ t_m\ t_n),\ \ \forall (s_m, s_n) \in \mathcal{A} \\
&(:\ (\rightarrow\ \$t_a\ \$t_b)\ (\text{transform}\ (:\ \$t_a\ t_l) \wedge (:\ \$t_b\ t_m)\ t_n)),\ \ \forall (s_l, s_m, s_n) \in \mathcal{R} \\
&(:\ (\prod \$x : \$t_a . \$m)\ (\text{transform}\ (:\ \$t_a\ t_l) \wedge (:\ \$m\ t_m)\ t_n)),\ \ \forall (s_l, s_m, s_n) \in \mathcal{R}
\end{aligned}
\tag{17}
$$

As above, lambda expressions are simulated by adding atoms of the form in Eq. 15 to the atomspace, and a context Γ is simulated by adding atoms corresponding to the typing relations it contains. Reduction of expression e in context Γ is simulated as previously by applying update to the pointed atomspace consisting of $\{\Gamma, e\}$ and the above constructions, along with † pointing to e. Further, we note that we can use PTS's can be regarded as a type-theoretic analogue of non-well-founded sets; from this viewpoint, a cyclical : relation corresponds to an accessible pointed graph (apg) underlying a non-well-founded set. For instance, including the axiom $s_1 : s_1$ in $\mathcal{A}$ defines s_1 as a type-theoretic analogue of a Quine atom. We note, however, that in the type-theoretic context, a cyclic PTS carries more structure than a non-well-founded set, since the rules ($\mathcal{R}$) carry information about how the $\rightarrow$ constructor interacts with the : relation. An interesting conjecture though would be that appropriately defined PTS's provide bisimulations of systems of non-well-founded sets definable within a recursive datatype (via a coalgebra on the powerset functor, definable in GCTT), as a general system of set equations ([3]) involving both $\in$ and $\rightarrow$ relations.

4.3 Probabilistic dependent types

Finally, we outline a version of the probabilistic dependent type system introduced in [19], and its bisimulation in $\mathbb{M}$. The syntax is a variation on the dependently typed lambda calculus:

$$
\begin{aligned}
&\mathcal{V} ::= v_n \\
&\mathcal{T} ::= t_n \ \mid\ \prod v_n : \mathcal{T} . \mathcal{E} \ \mid\ \mathcal{D}(\mathcal{T}) \ \mid\ \mathcal{T} \cup \mathcal{T} \ \mid\ \mathcal{T} \cap \mathcal{T} \ \mid\ \text{Type} \\
&\mathcal{E} ::= \mathcal{V} \ \mid\ (\mathcal{E}\ \mathcal{E}) \ \mid\ \lambda v_n : \mathcal{T} . \mathcal{E} \ \mid\ \text{random}_\rho(\mathcal{E}, \mathcal{E}) \ \mid\ \text{sample}(\mathcal{E}) \ \mid\ \text{thunk}(\mathcal{E})
\end{aligned}
\tag{18}
$$

Further, we allow the judgments $\mathcal{E} : \mathcal{T}$ (typing), $\mathcal{T} \preceq \mathcal{T}$ (subtyping), and $\mathcal{E} \rightarrow^\rho_\beta \mathcal{E}$ (weighted β-reduction), where $\rho \in \mathbb{R}$. The typing rules are as for the dependent typed lambda calculus for expressions not involving subtypes or probabilistic terms. The typing rules for subtypes include the standard $\Gamma \vdash a : A,\ A \preceq B \Rightarrow \Gamma \vdash a : B$,

$\Gamma \vdash A, B : \mathrm{Type} \Rightarrow \Gamma \vdash A \cap B \preceq A,\ A \cap B \preceq B,\ A \preceq A \cup B\ B \preceq A \cup B, \Gamma \vdash A \preceq B \Rightarrow \prod v_n : B.\mathcal{E} \preceq \prod v_n : A.\mathcal{E}, \Gamma, x : t \vdash A \preceq B \Rightarrow \prod x : t.A \preceq \prod x : t.B.$ These interact with the probabilistic terms via the following special rules:

$$\frac{\Gamma \vdash a : t_1,\ b : t_2}{\Gamma \vdash \mathrm{random}_\rho(a, b) : t_1 \cup t_2}$$

$$\frac{\Gamma \vdash A : \mathrm{Type}, p_A : \mathcal{D}(A)}{\Gamma \vdash \mathrm{sample}(p_A) : A}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{thunk}(a) : \mathcal{D}(A)}$$

where, we note that $\mathcal{D}(A)$ denotes the type of distributions over A (so, for instance, if $a : t_1,\ b : t_2$, then $\mathrm{thunk}(\mathrm{random}_\rho(a, b)) : \mathcal{D}(t_1 \cup t_2)$). For all expressions not involving probabilistic terms, $e_1 \to_\beta e_2$ in the dependent typed lambda calculus implies $e_1 \to^1_\beta e_2$ in the PDTS above. For probabilistic terms, we have the following computational rules:

$$\begin{aligned} \mathrm{random}_\rho(a, b) &\to^\rho_\beta a \\ \mathrm{random}_\rho(a, b) &\to^{1-\rho}_\beta a \\ \mathrm{sample}(\mathrm{thunk}(p_A)) &\to^1_\beta p_A \end{aligned} \tag{19}$$

Computationally, evaluation may proceed by stochastic β-reduction (i.e. sampling a reduction according to the weights ρ), or a 'full evaluation' may be made, by returning the set of all possible reduction sequences from a term, annotated with the total probability of each. We note that in any given reduction sequence, $e_1 \to^\rho_\beta e_2$ for $\rho > 0$ implies $t_2 \preceq t_1$ where $e_1 : t_1,\ e_2 : t_2$.

For the formulation in $\mathbb{M}$, we constrain the typing relation and encode lambda terms as in Eqs. 14 and 15; further, as above we encode contexts Γ by fixing atoms of the form : in atomspace. To encode the probabilistic terms, we choose fixed symbols $s_{1\ldots4}$ to correspond to $\mathrm{Distribution}, \mathrm{random}, \mathrm{sample}, \mathrm{thunk}$. Then, we fix the following atoms in atomspace:

$$\begin{aligned} &(:\ \mathrm{Distribution}\ (\to\ \mathrm{Type}\ \mathrm{Type})), \\ &(:\ \mathrm{random}\ (\to\ \$t_1\ \$t_2\ \$t_1 \cup \$t_2)), \\ &(=\ (\mathrm{random}\ \$a\ \$b)\ \$a), \\ &(=\ (\mathrm{random}\ \$a\ \$b)\ \$b), \\ &(:\ \mathrm{sample}\ (\to\ (\mathrm{Distribution}\ \$t_1)\ \$t_1)), \\ &(:\ \mathrm{thunk}\ (\to\ \$t_1\ (\mathrm{Distribution}\ \$t_1))), \\ &(=\ (\mathrm{sample}\ (\mathrm{thunk}\ \$a))\ \$a) \end{aligned} \tag{20}$$

Application of update to the pointed atomspace so defined, with † pointing to m_e (corresponding to expression e), results in a simulation of a probabilistic reduction of e

in the PDTS above. As defined, update will simulate the 'full evaluation' of all possible paths, and hence a bisimulation exists between full evaluation dynamics in the PDTS GLTS using $\beta\rho$-reduction and the GLTS defined by $\mathbb{M}$ with the restricted atomspace above. We note that, in both cases, the weights on particular paths are lost, since the ρ values are not explicitly recorded; however. it is straightforward to define a GLTS over the extended system, $(X \times \mathbb{R}, A, f)$, where $f(x) = \{((x_1, p_1), a_1), ((x_2, p_2), a_2), ...\}$ denotes that action a on x results in x_1 with probability p_1, x_2 with probability p_2, and so on.

5 Implementation of Bisimulation proof in a Guarded Cubical Type Theory type checker

We briefly give an example to show the feasibility of our approach with an implementation of a bisimulation proof for a small-scale type system in a Guarded Cubical Type Theory type checker [4]. Here, we model a minimal type system, which has one type constant $A : \mathrm{Type}$ with two constructors $v_1, v_2 : A$; one function constant $f_1 : A \to A$, where $f_1(v_1) = v_2$ and $f_1(v_2) = v_1$; and includes the sample and thunk constructs, which are combined following the syntax of Eq. 18. Our implementation models a fragment of this system where expressions are restricted to include at most three subexpressions. Hence, valid expressions of the language include: $(f_1\ (f_1\ v_1))$, $(\mathrm{thunk}\ (f_1\ v_2))$, $(\mathrm{sample}\ (\mathrm{thunk}\ v_1))$, $(f_1\ v_2)$. Our implementation in a Haskell-based Guarded Cubical Type Theory type checker [4] is given in Appendix A. Here, we implement evaluation in this system via (i) a pattern matcher over an atomspace ('update'), and (ii) direct implementation of β-reduction via case analysis over the expression space ('beta3'). We define GLTS's using both forms of evaluation ('str1' and 'str2'), and finally derive a proof that these GLTS's are bisimilar ('bisim'). The code for this example is also provided at: `https://github.com/jwarrell/metta_bisimulation`

6 Discussion

In the above, we have introduced a formal meta-probabilistic programming language, formalized in GCTT, and proposed that bisimutations link the specific object-languages (or domain specific languages) outlined above with their simulations in $\mathbb{M}$. Specifically, we have proposed that the restricted forms of $\mathbb{M}$ outlined in Secs. 4.1 and 4.2 and 4.3 form bisimulations of the simply typed lambda calculus, arbitrary PTS's, and the target PDTS, respectively.

Finally, we mention some of the areas of investigation opened up by the formal model outlined. First, we note that, while we have focused on 'full' probabilistic programming evaluation, other possibilities include investigation of sampling based evaluation which performs only one meta-graph update at each step, stochastically chosen from the possible graph rewriting locations. Second, we intend to derive further bisimulations for other kinds of probabilistic logic, particularly, probabilistic paraconsistent logic [7], and probabilistic analogues of pure type systems [2], which may be suitable for models involving infinite-order probabilities [5]. Lastly, we intend to expand our implementation

of aspects of this framework in Guarded Cubical Agda [17] to provide more complete implementations of the metalanguage and type systems explored here.

References

1. Baier, C. and Katoen, J.P., 2008. Principles of model checking. MIT press.
2. Barendregt, Henk, and Lennart Augustsson, 1992. "Lambda Calculi with Types." Handbook of Logic in Computer Science 34: 239-250.
3. Barwise, J. and Moss, L., 1996. Vicious circles: on the mathematics of non-wellfounded phenomena.
4. Birkedal, L., Bizjak, A., Clouston, R., Grathwohl, H.B., Spitters, B. and Vezzosi, A., 2016. Guarded cubical type theory: Path equality for guarded recursion. arXiv preprint arXiv:1606.05223.
5. Goertzel, B., 2008. Modeling Uncertain Self-Referential Semantics with Infinite-Order Probabilities.
6. Goertzel, B., 2020. Folding and Unfolding on Metagraphs. arXiv preprint arXiv:2012.01759.
7. Goertzel, B., 2020. Paraconsistent Foundations for Probabilistic Reasoning, Programming and Concept Formation. arXiv preprint arXiv:2012.14474.
8. Goertzel, B., 2021. Reflective Metagraph Rewriting as a Foundation for an AGI 'Language of Thought'. arXiv preprint arXiv:2112.08272.
9. Harper, R., 2012. Notes on logical frameworks. Lecture notes, Institute for Advanced Study, Nov, 29, p.34.
10. Møgelberg, R.E. and Veltri, N., 2019. Bisimulation as path type for guarded recursive types. Proceedings of the ACM on Programming Languages, 3(POPL), pp.1-29.
11. Møgelberg, R.E. and Paviotti, M., 2019. Denotational semantics of recursive types in synthetic guarded domain theory. Mathematical Structures in Computer Science, 29(3), pp.465-510.
12. Mokhov, A., 2017. Algebraic graphs with class (functional pearl). ACM SIGPLAN Notices, 52(10), pp.2-13.
13. Paviotti, M., Møgelberg, R.E. and Birkedal, L., 2015. A model of PCF in guarded type theory. Electronic Notes in Theoretical Computer Science, 319, pp.333-349.
14. Potapov, A., 2021. MeTTa language specification. `https://wiki.opencog.org/w/Hyperon`.
15. Staton, S., Wood, F., Yang, H., Heunen, C. and Kammar, O., 2016, July. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In 2016 31st annual acm/ieee symposium on logic in computer science (lics) (pp. 1-10).
16. TrueAGI, 2021. Hyperon-experimental repository. `https://github.com/trueagi-io/hyperon-experimental`.
17. Veltri, N. and Vezzosi, A., 2020, January. Formalizing π-calculus in guarded cubical Agda. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (pp. 270-283).
18. Vezzosi, A., Mörtberg, A. and Abel, A., 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. Journal of Functional Programming, 31.
19. Warrell, J. and Gerstein, M., 2018. Dependent Type Networks: A Probabilistic Logic via the Curry-Howard Correspondence in a System of Probabilistic Dependent Types. In Uncertainty in Artificial Intelligence, Workshop on Uncertainty in Deep Learning. `http://www.gatsby.ucl.ac.uk/~balaji/udl-camera-ready/UDL-19.pdf`.

Appendices

A Proof of Bisimulation for Small-scale Type System in a Guarded Cubical Type Theory type checker

Below, we provide the code for the example discussed in Sec. 5, which uses a Haskell-based GCTT type checker [4]. The code for this example is also provided at: `https://github.com/jwarrell/metta_bisimulation`

Keywords, Preliminaries and Helper functions

```
module metta_bisimulation where

---- To run this example, first install the implementation of Guarded Cubical Type Theory from:
----     https://github.com/hansbugge/cubicaltt/tree/gcubical
---- and call: cubical metta_bisimulation

import prelude
import nat

------------ Standard definitions

data list (A : U) = nil
                  | cons (a : A) (as : list A)

append (A : U) : list A -> list A -> list A = split
  nil -> idfun (list A)
  cons x xs -> \(ys : list A) -> cons x (append A xs ys)

foldr (A B : U) (f : A -> B -> B) (b0 : B) : list A -> B
    = split
      nil -> b0
      cons a as -> f a (foldr A B f b0 as)

data bool = false | true


------------ Keywords / expression data-types

-- keywords
data K
 = nl | v1 | v2 | f1 | t | s

-- expressions
data E3
 = Cons (k1 : K) (k2 : K) (k3 : K)

-- all valid expressions
data Expr = v1 | v2 | f1 | f1_v1 | f1_v2 | f1_f1_v1 | f1_f1_v2 | t_f1_v1
          | t_f1_v2 | t_v1 | t_v2 | s_t_v1 | s_t_v2 | t_t_v1 | t_t_v2
```

```
------------ Helper functions

-- keyword equality
EqK (k1 : K) : K -> bool = split
        nl -> fun1 k1
          where fun1 : K -> bool = split
            nl -> true
            v1 -> false
            v2 -> false
            f1 -> false
            t -> false
            s -> false
        v1 -> fun2 k1
          where fun2 : K -> bool = split
            nl -> false
            v1 -> true
            v2 -> false
            f1 -> false
            t ->false
            s -> false
        v2 -> fun3 k1
          where fun3 : K -> bool = split
            nl -> false
            v1 -> false
            v2 -> true
            f1 -> false
            t -> false
            s -> false
        f1 -> fun4 k1
          where fun4 : K -> bool = split
            nl -> false
            v1 -> false
            v2 -> false
            f1 -> true
            t -> false
            s -> false
        t -> fun5 k1
          where fun5 : K -> bool = split
            nl -> false
            v1 -> false
            v2 -> false
            f1 -> false
            t -> true
            s -> false
        s -> fun6 k1
          where fun6 : K -> bool = split
            nl -> false
            v1 -> false
            v2 -> false
            f1 -> false
            t -> false
            s -> true
```

```
m1 : Expr -> E3
  = split
    v1 -> Cons nl nl v1
    v2 -> Cons nl nl v2
    f1 -> Cons nl nl f1
    f1_v1 -> Cons nl f1 v1
    f1_v2 -> Cons nl f1 v2
    f1_f1_v1 -> Cons f1 f1 v1
    f1_f1_v2 -> Cons f1 f1 v2
    t_f1_v1 -> Cons t f1 v1
    t_f1_v2 -> Cons t f1 v2
    t_v1 -> Cons nl t v1
    t_v2 -> Cons nl t v2
    s_t_v1 -> Cons s t v1
    s_t_v2 -> Cons s t v2
    t_t_v1 -> Cons t t v1
    t_t_v2 -> Cons t t v2

m2 : Expr -> list K
  = split
    v1 -> cons v1 nil
    v2 -> cons v2 nil
    f1 ->  cons f1 nil
    f1_v1 ->  cons f1 (cons v1 nil)
    f1_v2 ->  cons f1 (cons v2 nil)
    f1_f1_v1 -> cons f1 (cons f1 (cons v1 nil))
    f1_f1_v2 -> cons f1 (cons f1 (cons v2 nil))
    t_f1_v1 ->  cons t (cons f1 (cons v1 nil))
    t_f1_v2 ->  cons t (cons f1 (cons v2 nil))
    t_v1 ->  cons t (cons v1 nil)
    t_v2 ->  cons t (cons v2 nil)
    s_t_v1 -> cons s (cons t (cons v1 nil))
    s_t_v2 -> cons s (cons t (cons v2 nil))
    t_t_v1 -> cons t (cons t (cons v1 nil))
    t_t_v2 -> cons t (cons t (cons v2 nil))

E3_to_list : E3 -> list K
  = split
    Cons k1 k2 k3 -> cons k1 (cons k2 (cons k3 nil))
```

Helper functions (continued)

```
list_to_Expr : list K -> Expr
    = split
      nil -> v1
      cons k0 k0s -> fun1 k0 k0s
          where fun1 (k0 : K) : list K -> Expr = split
                nil -> fun2 k0
                  where fun2 : K -> Expr = split
                        nil -> v1
                        v1 -> v1
                        v2 -> v2
                        f1 -> f1
                        t -> v1
                        s -> v1
                cons k1 k1s -> fun3 k0 k1 k1s
                      where fun3 (k0 k1 : K) : list K -> Expr = split
                          nil -> fun4 k0 k1
                              where fun4 (k0 : K) : K -> Expr = split
                                    nl -> v1
                                    v1 -> fun5 k0
                                        where fun5 : K -> Expr = split
                                              nl -> v1
                                              v1 -> v1
                                              v2 -> v1
                                              f1 -> f1_v1
                                              t -> t_v1
                                              s -> v1
                                    v2 -> fun6 k0
                                        where fun6 : K -> Expr = split
                                              nl -> v1
                                              v1 -> v1
                                              v2 -> v1
                                              f1 -> f1_v2
                                              t -> t_v2
                                              s -> v1
                                    f1 -> v1
                                    t -> v1
                                    s -> v1
```

```
cons k2 k2s -> fun7 k0 k1 k2 k2s
      where fun7 (k0 k1 k2 : K) : list K -> Expr = split
            nil -> fun8 k0 k1 k2
              where fun8 (k0 k1 : K) : K -> Expr = split
                  nl -> v1
                  v1 -> fun9 k0 k1
                    where fun9 (k0 : K) : K -> Expr = split
                        nl -> v1
                        v1 -> v1
                        v2 -> v1
                        f1 -> fun10 k0
                          where fun10 : K -> Expr = split
                              nl -> v1
                              v1 -> v1
                              v2 -> v1
                              f1 -> f1_f1_v1
                              t -> t_f1_v1
                              s -> v1
                        t -> fun11 k0
                          where fun11 : K -> Expr = split
                              nl -> v1
                              v1 -> v1
                              v2 -> v1
                              f1 -> v1
                              t -> t_t_v1
                              s -> s_t_v1
                            s -> v1
                    v2 -> fun12 k0 k1
                      where fun12 (k0 : K) : K -> Expr = split
                        nl -> v1
                        v1 -> v1
                        v2 -> v1
                        f1 -> fun13 k0
                          where fun13 : K -> Expr = split
                              nl -> v1
                              v1 -> v1
                              v2 -> v1
                              f1 -> f1_f1_v2
                              t -> t_f1_v2
                              s -> v1
                        t -> fun14 k0
                           where fun14 : K -> Expr = split
                              nl -> v1
                              v1 -> v1
                              v2 -> v1
                              f1 -> v1
                              t -> t_t_v2
                              s -> s_t_v2
                        s -> v1
                    f1 -> v1
                    t -> v1
                    s -> v2
          cons k3 k3s -> v1
```

Evaluation via MeTTa pattern matching

```
---------- Evaluation via MeTTa pattern matching

-- rules
data rule
 = Cons (r1 : K) (r2 : K) (r3 : K)

-- atomspace
A : U = list rule

as : A = cons (Cons s t nl) (cons (Cons f1 v1 v2) (cons (Cons f1 v2 v1) nil))

-- compress
comp_fun (k : K) (ks : list K) : (list K)
     = fun1 ks k
          where fun1 (ks : list K) : (k : K) -> (list K) = split
                nl -> ks
                v1 -> append K (cons v1 nil) ks
                v2 -> append K (cons v2 nil) ks
                f1 -> append K (cons f1 nil) ks
                t -> append K (cons t nil) ks
                s -> append K (cons s nil) ks

compress (ks_in : list K) : (list K)
    = foldr K (list K) comp_fun nil ks_in
```

```
-- Toy MeTTa evaluator
rewrite (r1 r2 r3 : K) : (ks : list K) -> (list K)
    = split
      nil -> nil
      cons k1 k1s -> fun1 k1s (EqK r1 k1)
          where fun1 (k1s : list K) : (b1 : bool) -> list K = split
              false -> cons k1 (rewrite r1 r2 r3 k1s)
              true -> fun1a k1s
                  where fun1a : (k1s : list K) -> (list K) = split
                      nil -> cons k1 nil
                      cons k2 k2s -> fun2 k2s (EqK r2 k2)
                          where fun2 (k2s : list K) : (b2 : bool) -> list K = split
                              false -> cons k1 (rewrite r1 r2 r3 k1s)
                              true -> cons r3 (rewrite r1 r2 r3 k2s)

upd_fun (r : rule) (ks : list K) : list K
     = fun1 ks r
         where fun1 (ks : list K) : (r : rule) -> list K = split
             Cons k1 k2 k3 -> rewrite k1 k2 k3 ks

update (rs_in : list rule) : (ks_in : list K) -> list K = split
    nil -> nil
    cons k1 k1s -> fun1 rs_in (cons k1 k1s) k1
       where fun1 (rs_in : list rule) (ks_in : list K) : K -> list K = split
            nl ->  compress (foldr rule (list K) upd_fun ks_in rs_in)
            v1 ->  compress (foldr rule (list K) upd_fun ks_in rs_in)
            v2 ->  compress (foldr rule (list K) upd_fun ks_in rs_in)
            f1 ->  compress (foldr rule (list K) upd_fun ks_in rs_in)
            t ->  ks_in
            s ->  compress (foldr rule (list K) upd_fun ks_in rs_in)

update' : list K -> list K
    = update as
```

Evaluation via direct β-reduction definition

```
-- beta reduction
beta3 : E3 -> E3
  = split
    Cons k1 k2 k3 -> fun1 k3 k2 k1
         where fun1 (k3 k2 : K) : K -> E3 = split
                    nl -> fun2 k3 k2
                        where
                           fun2 (k3 : K) : K -> E3 = split
                                nl -> Cons nl nl k3
                                v1 -> Cons nl v1 k3
                                v2 -> Cons nl v2 k3
                                f1 -> fun3 k3
                                   where fun3 : K -> E3 = split
                                           nl -> Cons nl nl nl
                                           v1 -> Cons nl nl v2
                                           v2 -> Cons nl nl v1
                                           f1 -> Cons nl f1 f1
                                           t -> Cons nl f1 t
                                           s -> Cons nl f1 s
                                t -> Cons nl t k3
                                s -> Cons nl s k3
                    v1 -> Cons v1 k2 k3
                    v2 -> Cons v2 k2 k3
```

```
f1 -> fun4 k3 k2
    where
       fun4 (k3 : K) : K -> E3 = split
            nl -> Cons f1 nl k3
            v1 -> Cons f1 v1 k3
            v2 -> Cons f1 v2 k3
            f1 -> fun5 k3
               where fun5 : K -> E3 = split
                       nl -> Cons f1 f1 nl
                       v1 -> Cons nl f1 v2
                       v2 -> Cons nl nl v2
                       f1 -> Cons f1 f1 f1
                       t -> Cons f1 f1 t
                       s -> Cons f1 f1 s
            t -> Cons nl t k3
            s -> Cons nl s k3
t -> Cons t k2 k3
s -> fun6 k3 k2
    where
       fun6 (k3 : K) : K -> E3 = split
            nl -> Cons s nl k3
            v1 -> Cons s v1 k3
            v2 -> Cons s v2 k3
            f1 -> Cons s f1 k3
            t -> fun7 k3
               where fun7 : K -> E3 = split
                       nl -> Cons s t nl
                       v1 -> Cons nl nl v1
                       v2 -> Cons nl nl v2
                       f1 -> Cons s t f1
                       t -> Cons s t t
                       s -> Cons s t s
            s -> Cons s s k3
```

Proof of Bisimulation

```
-- Guarded Streams
data gStr (A : U) k
  = Cons (x : A) (xs : |> k (gStr A $ k))

Str (A : U) : U
  = forall k, gStr A $ k

-- Head and Tail
ghd (A : U) k : (xs : gStr A $ k) -> A
  = split
      Cons x _ -> x

gtl (A : U) k : (xs : gStr A $ k) -> |> k (gStr A $ k)
  = split
      Cons _ ys -> ys

hd (A : U) (xs : Str A) : A
  = ghd A $ k0 (xs $ k0)

tl (A : U) (xs : Str A) : Str A
  = prev k (gtl A $ k (xs $ k))

-- Iteration
iterate k (A : U) (f : A -> A) : A -> gStr A $ k
  = fix k it (A -> gStr A $ k)
    (\ (a : A) ->
      Cons a (next k [it' <- it] (it' (f a))))

-- Bisimulation between streams
data Bisimilar (A : U) (xs ys : Str A) =
  consBi (h : Id A (hd A xs) (hd A ys))
         (t : Bisimilar A (tl A xs) (tl A ys))
```

```
-- update using beta reduction
update1 (e : Expr) :  Expr
    =  list_to_Expr (compress (E3_to_list (beta3 (m1 e))))

-- update using pattern matching
update2  (e : Expr) :  Expr
    =  list_to_Expr (update' (m2 e))

-- generate stream from any starting expression
str1 (e0 : Expr) : Str Expr = [ k ] iterate $k Expr update1 e0

str2 (e0 : Expr) : Str Expr = [ k ] iterate $k Expr update2 e0

-- proof that streams starting from same expression are bisimilar
bisim : (e0 : Expr) -> (Bisimilar Expr (str1 e0) (str2 e0)) = split
     v1 -> consBi (refl Expr v1) (bisim v1)
     v2 -> consBi (refl Expr v2) (bisim v2)
     f1 -> consBi (refl Expr f1) (bisim f1)
     f1_v1 -> consBi (refl Expr f1_v1) (bisim v2)
     f1_v2 ->  consBi (refl Expr f1_v2) (bisim v1)
     f1_f1_v1 ->  consBi (refl Expr f1_f1_v1) (bisim f1_v2)
     f1_f1_v2 -> consBi (refl Expr f1_f1_v2) (bisim v2)
     t_f1_v1 ->  consBi (refl Expr t_f1_v1) (bisim t_f1_v1)
     t_f1_v2 ->  consBi (refl Expr t_f1_v2) (bisim t_f1_v2)
     t_v1 -> consBi (refl Expr t_v1) (bisim t_v1)
     t_v2 -> consBi (refl Expr t_v2) (bisim t_v2)
     s_t_v1 -> consBi (refl Expr s_t_v1) (bisim v1)
     s_t_v2 ->  consBi (refl Expr s_t_v2) (bisim v2)
     t_t_v1 ->  consBi (refl Expr t_t_v1) (bisim t_t_v1)
     t_t_v2 ->  consBi (refl Expr t_t_v2) (bisim t_t_v2)
```