Declarative programming for agent applications

J. W. Lloyd · K. S. Ng

Abstract   This paper introduces the execution model of a declarative programming language intended for agent applications. Features supported by the language include functional and logic programming idioms, higher-order functions, modal computation, probabilistic computation, and some theorem-proving capabilities. The need for these features is motivated and examples are given to illustrate the central ideas.

Keywords   Declarative programming · Multi-agent systems · Multi-modal logic · Higher-order logic · Modal programming · Probabilistic programming

1 Introduction

The agent paradigm is currently attracting considerable interest, largely because of its promise of providing appropriate technology for the increasingly sophisticated applications of computer systems. Consequently, for the last two decades, there has been considerable interest in designing programming languages that directly support agent concepts. In this paper, we contribute to this effort by introducing the execution model of a declarative agent programming language, called Bach, that amongst other things provides support for agent concepts, such as beliefs, and also the probabilistic handling of uncertainty.

   We now examine some design considerations for Bach. To motivate its execution model, consider an agent situated in some environment that can receive percepts from the environment and can apply actions to the environment. The primary task of the agent is to choose appropriate actions to achieve its goals (however these are defined). A major ingredient needed to make appropriate choices is the set of beliefs of the agent; for example, the agent may need to reason about its understanding of the beliefs of other agents, temporal

J. W. Lloyd (✉) · K. S. Ng
College of Engineering and Computer Science, The Australian National University, Canberra, Australia
e-mail: john.lloyd@anu.edu.au

K. S. Ng
e-mail: keesiong.ng@gmail.com

considerations of when certain situations held, the goals of the agent itself, the possible effects of its actions, uncertainty in any of the preceding considerations, and so on. This situation suggests the choice of a highly expressive logic as the basis for the programming language.

What features should the logic have? The standard way of modelling mentalistic concepts such as beliefs, intentions, and so on, is with modal logic and since there are a number of such concepts and generally a number of agents in any application, we are led to the need for a multi-modal logic. While propositional modal logics are commonly used to analyse agents (see, for example, [1–3]), to adequately model beliefs, the logic must be much more expressive than propositional logic; we argue for the need for higher-order modal logic.

Furthermore, in many applications, it is necessary for an agent to deal with uncertainty; thus some beliefs are likely to be probabilistic. This issue leads directly to the more general problem of integrating logic and probability, a topic in artificial intelligence that is currently attracting substantial interest (see, for example, [4–9], and the references therein). One of the advantages of working in a higher-order logic is that it is expressive enough to easily encompass uncertainty without any additional logical machinery. The key idea is to represent uncertainty by probability densities; these are non-negative functions whose integral is one. Densities can conveniently be represented and manipulated by higher-order functions. It is generally straightforward to represent directly in a theory the probability that a particular assumption holds and compute the probability that a theorem proved from such assumptions holds. In summary, knowledge representation requirements suggest the need for the underlying logic of Bach to be multi-modal, higher-order logic.

As well as representing knowledge, it is necessary to reason about it. The reasoning system introduced in this paper combines a computation component and a proof component. The computation component is an equational reasoning system that significantly extends existing functional programming languages by adding facilities that provide modal computation and the idioms of logic programming. The proof component is a fairly conventional tableau theorem prover for modal higher-order logic similar to what is proposed in [10]. The computation component and the proof component are tightly integrated, in the sense that either can call the other. Furthermore, this synergy between the two is shown to make possible interesting reasoning tasks. The presentation below of the reasoning system considers first the case of (pure) computation, where no proof is involved, then (pure) proof, where no computation is involved, and finally the two are put together.

However, for this paper, we are primarily interested in the deployment of the reasoning system as the execution model of a *programming language* and, for this reason, the computation component is the one that is of most interest and relevance. For Bach, reasoning is primarily computation that occasionally needs some theorem proving support. Thus, while the reasoning system is presented theoretically with equal emphasis on computation and proof, the use of the proof component of Bach is restricted in practical applications.

Bach is thus a modal probabilistic functional logic programming language whose programs are theories in multi-modal, higher-order logic. Its core is the functional programming language Haskell [11], extended in such a way as to also provide the logic programming idioms. In addition, modalities are included so that programs are modal theories. The extension to probabilistic theories requires no extension of the logic since higher-order functions are sufficient to represent and reason about probability densities, although *efficient* probabilistic reasoning does require additional support at the programming language level. Throughout, for clarity, we use the usual mathematical notation for Bach programs; the actual Bach syntax is similar to Haskell.

The design of Bach continues one thread in the development of declarative programming languages that goes back about 15 years. The starting point was the recognition that Prolog

[12] has various flaws that reduce its credibility as a declarative programming language; these include non-declarative meta-programming facilities and the lack of a type system. This motivated the Gödel programming language [13] that was closely based on Prolog but had a polymorphic type system and declarative meta-programming facilities. The next step was Escher [14] that differed markedly from Gödel in that it was a higher-order language and was based on equational theories rather than clausal theories. In its final form, Escher was presented as an extension to Haskell, thus taking advantage of the many good design decisions of that language, by adding the idea of programming with abstractions [15] that provides the logic programming idioms. Escher also avoided the highly problematical negation as failure rule by treating negation as just another function. Bach builds on Escher mainly by providing modal and probabilistic computation that is especially useful for agent applications.

The paper is organised as follows. Section 2 contains an overview of multi-modal, higher-order logic. The computation component of Bach is described in Sect. 3. This is followed by the proof component in Sect. 4. The full reasoning system consisting of computation and proof combined is given in Sect. 5. Small instructive programming examples are sprinkled throughout these three sections to illustrate central concepts. Section 6 provides some larger programming examples. Section 7 contains a discussion of related and future work. We conclude in Sect. 8.

2 Logic

The underlying logic of Bach is a multi-modal, higher-order logic. We give a brief summary of the logic in the following, focusing to begin with on the monomorphic version. We define types and terms, and give an introduction to the modalities that we will use. Full details of the logic can be found in [16]. Other useful references on modal higher-order logic include [10,17] and on higher-order logic include [15,18–22]. For a highly readable account of the advantages of working in higher-order logic rather than first-order, we strongly recommend [23].

Definition 1 An *alphabet* consists of three sets: a set $\mathfrak{T}$ of type constructors; a set $\mathfrak{C}$ of constants; and a set $\mathfrak{V}$ of variables.

Each type constructor in $\mathfrak{T}$ has an arity. The set $\mathfrak{T}$ always includes the type constructor Ω of arity 0. Ω is the type of the booleans. Each constant in $\mathfrak{C}$ has a signature. The set $\mathfrak{V}$ is denumerable. Variables are typically denoted by $x, y, z, \ldots$.

Types are built up from the set of type constructors using the symbols $\rightarrow$ and $\times$.

Definition 2 A *type* is defined inductively as follows.

1. If T is a type constructor of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T \, \alpha_1 \ldots \alpha_k$ is a type. (Thus a type constructor of arity 0 is a type.)
2. If α and β are types, then $\alpha \rightarrow \beta$ is a type.
3. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type.

Example 1 Following are some common types we will need other than Ω. The type of the integers is denoted by *Int*, and the type of the reals by *Real*. Also (*List* σ) is the type of lists whose items have type σ. Here *Int*, *Real* and *List* are all type constructors. The first two have arity 0 and the last has arity 1. A function that maps elements of type α to elements of type β has type $\alpha \rightarrow \beta$. Since sets are identified with predicates in the logic, sets whose elements have type σ have type $\sigma \rightarrow \Omega$. We sometimes write $\{\sigma\}$ as a synonym for $\sigma \rightarrow \Omega$ when

we want to make a distinction between sets and predicates. A particular class of functions of interest is that of probability densities. We introduce the synonym *Density* $\tau \equiv \tau \rightarrow Real$, but with the understanding that functions of type *Density* τ are probability densities over elements of type τ rather than arbitrary real-valued functions over elements of type τ.

The set $\mathfrak{C}$ always includes the following constants.

1. $\top$ and $\bot$, having signature Ω.
2. $=_\alpha$, having signature $\alpha \rightarrow \alpha \rightarrow \Omega$, for each type α.
3. $\neg$, having signature $\Omega \rightarrow \Omega$.
4. $\wedge$, $\vee$, and $\longrightarrow$ having signature $\Omega \rightarrow \Omega \rightarrow \Omega$.
5. Σ_α and Π_α, having signature $(\alpha \rightarrow \Omega) \rightarrow \Omega$, for each type α.

The intended meaning of $\top$ is true, and that of $\bot$ is false. The intended meaning of $=_\alpha$ is identity, and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, and $\longrightarrow$ are as usual. The intended meanings of Σ_α and Π_α are as follows: Σ_α maps a predicate to $\top$ iff the predicate maps at least one element to $\top$; Π_α maps a predicate to $\top$ iff the predicate maps all elements to $\top$.

Other useful constants we will usually have in applications include the integers, the real numbers, and data constructors like $\sharp_\sigma : \sigma \rightarrow List\ \sigma \rightarrow List\ \sigma$ and $[]_\sigma : List\ \sigma$ for constructing lists with elements of type σ. The notation $C : \sigma$ is used to denote that the constant C has signature σ.

We assume there are necessity modality operators $\Box_i$, for $i = 1, \dots, m$.

Definition 3 A *term*, together with its type, is defined inductively as follows.

1. A variable in $\mathfrak{V}$ of type α is a term of type α.
2. A constant in $\mathfrak{C}$ having signature α is a term of type α.
3. (Abstraction) If t is a term of type β and x a variable of type α, then $\lambda x.t$ is a term of type $\alpha \rightarrow \beta$.
4. (Application) If s is a term of type $\alpha \rightarrow \beta$ and t a term of type α, then $(s\ t)$ is a term of type β.
5. (Tuple) If $t_1, \dots, t_n$ are terms of type $\alpha_1, \dots, \alpha_n$, respectively, then $(t_1, \dots, t_n)$ is a term of type $\alpha_1 \times \cdots \times \alpha_n$.
6. (Modal Term) If t is a term of type α and $i \in \{1, \dots, m\}$, then $\Box_i t$ is a term of type α.

Example 2 Constants like $\top : \Omega$, $42 : Int$, $3.11 : Real$, and $+ : Int \rightarrow Int \rightarrow Int$ are terms. Variables like x, y, z are terms. An example of a term that can be formed using abstraction is $\lambda x.((+\ x)\ x)$ of type $Int \rightarrow Int$, whose intended meaning is a function that takes a number x and returns $x + x$. To apply that function to the constant 42, for example, we use application to form the term $(\lambda x.((+\ x)\ x)\ 42)$, which has type Int.

Example 3 The term $(\sharp_{Int}\ 2\ (\sharp_{Int}\ 3\ []_{Int}))$ of type $(List\ Int)$ represents a list with the integers 2 and 3 in it, obtained via a series of applications from the constants $\sharp_{Int}$, $[]_{Int}$, 2, and 3, each of which is a term. For convenience, we sometimes write $[2, 3]$ to represent the same list.

Example 4 Sets are identified with predicates in the logic. Thus, the term

$$\lambda x.((\vee\ ((=_{Int}\ x)\ 2))\ ((=_{Int}\ x)\ 3)) \tag{1}$$

of type $Int \rightarrow \Omega$ can be used to represent the set containing the integers 2 and 3. We often use infix notation for common function symbols like equality and the connectives. We also

adopt the convention that applications are left-associative; thus $(f\ x\ y)$ means $((f\ x)\ y)$. These conventions allow us to write $\lambda x.((x =_{Int} 2) \vee (x =_{Int} 3))$ instead of (1) above. For convenience, we sometimes also write $\{2, 3\}$ to represent the same set. Since sets are predicates, set membership test is obtained using function application. Let s denote (1) above. To check whether a number y is in the set, we write $(s\ y)$.

Terms of the form $(\Sigma_\alpha\ \lambda x.t)$ are written as $\exists_\alpha x.t$ and terms of the form $(\Pi_\alpha\ \lambda x.t)$ are written as $\forall_\alpha x.t$ (in accord with the intended meaning of Σ_α and Π_α). A formula is a term of type Ω. The universal closure of a formula φ is denoted by $\forall(\varphi)$.

There is a *default term* for each type. For example, the default term of type Ω is $\bot$, that of type Int is 0, that of type $List\ \alpha$ for any α is $[]_\alpha$, and that of type $\{\alpha\}$ for any α is $\{\}$ (that is, $\lambda x.\bot$).

The polymorphic version of the logic extends what is given above by also having available parameters which are type variables (denoted by a, b, c, ...). The definition of a type as above is then extended to polymorphic types that may contain parameters and the definition of a term as above is extended to terms that may have polymorphic types. We work in the polymorphic version of the logic in the remainder of the paper. In this case, we drop the α in constants like $\exists_\alpha$, $\forall_\alpha$, $=_\alpha$, $[]_\alpha$ and $\sharp_\alpha$, since the types associated with these are now inferred from the context.

Example 5 A common polymorphic function we need is $if\_then\_else : \Omega \times a \times a \to a$. Using it, we can give the following equivalent way of writing (1) above:

$$\lambda x.(if\_then\_else\ ((x = 2), \top, (if\_then\_else\ ((x = 3), \top, \bot)))).$$

Writing *if x then y else z* as syntactic sugar for $(if\_then\_else\ (x, y, z))$, the above can be written in the following more readable form:

$$\lambda x.if\ x = 2\ then\ \top\ else\ if\ x = 3\ then\ \top\ else\ \bot.$$

Discrete probability densities can also be written down easily as terms using *if_then_else*. For instance, the term

$$\lambda x.if\ x = \top\ then\ 0.3\ else\ if\ x = \bot\ then\ 0.7\ else\ 0$$

of type *Density* Ω denotes a probability density over the booleans.

Modalities can have a variety of meanings. Some of these are indicated below; more detail can be found in, e.g., [1,3,16]. Consider an application with three agents. One meaning for the necessity operator is knowledge. So, we can use $\Box_i\varphi$, for $i = 1, 2, 3$, to denote 'agent i knows φ'. In this case, the modalities $\Box_1, \Box_2$, and $\Box_3$ can be more meaningfully written as K_1, K_2, and K_3. A weaker notion of modality is that of belief. We can use $\Box_i\varphi$, for $i = 4, 5, 6$, to denote 'agent $(i - 3)$ believes φ'. In this case, $\Box_4, \Box_5, \Box_6$ can be written as B_1, B_2, B_3. Modalities can also have a variety of temporal readings. We can introduce $\Box_7$ for 'next' (written as $\bigcirc$), $\Box_8$ for 'always in the future' (written simply as $\Box$), $\Box_9$ for 'last' (written as $\bullet$), and $\Box_{10}$ for 'always in the past' (written as $\blacksquare$). Taking the dual of $\Box$ and $\blacksquare$ we obtain $\Diamond$ ('sometime in the future') and $\blacklozenge$ ('sometime in the past').

A novel feature of the logic is that modalities can be applied to terms, not just formulas. Thus terms such as $B_i 42$ and $\bullet f$, where f is a function, are admitted. Such terms are called modal terms. The need for modal terms arises naturally in applications, as we shall see below.

The logic can be given a rather conventional semantics in the usual Kripke style for multi-modal logics, with higher-order interpretations at each world. However, since the concept of

a modal term is new in modal logic, we give some intuition for the semantics of modal terms. If t is a formula, then the meaning of $\Box_i t$ in a world is $\top$ if the meaning of t in all accessible worlds is $\top$, its meaning is $\bot$ if the meaning of t in all accessible worlds is $\bot$, and, in the other cases, the meaning of $\Box_i t$ is conventionally defined to be $\bot$. This suggests an obvious extension to terms t that have rank 0 (that is, do not have type of the form $\alpha \to \beta$): if t has the same meaning in all accessible worlds, then the meaning of $\Box_i t$ should be this common meaning; otherwise, the meaning of $\Box_i t$ should be some default value. This definition then becomes the base case of an inductive definition on the rank of the type of t of the semantics of a modal term $\Box_i t$. The details of this are given in [16, Definition 3.10].

Each application has a distinguished pointed interpretation (I, w) known as the *intended pointed interpretation*, where I is an interpretation and w is a world in I. This means that, in the application, w is the actual world and I provides the worlds accessible to w by the various accessibility relations.

In modal logics, constants generally have different meanings in different worlds. Certain constants can be declared to be rigid; they then have the same meaning in all worlds (in the semantics). Except in the most sophisticated applications, it is entirely natural for some constants to be rigid. For instance, we can declare all data constructors (e.g. $\top, \bot, 1, 2, 3, \ldots, \sharp, []$) to be rigid. Also, all constants in the Haskell Prelude, which is a library of basic function definitions, can be declared to be rigid. A term is *rigid* if every constant in it is rigid.

A theory, which is a set of formulas, can consist of two kinds of assumptions, global and local. The essential difference is that global assumptions are true in each world in the intended pointed interpretation, while local assumptions only have to be true in the actual world in the intended pointed interpretation. Each kind of assumption has a certain role to play in computations. A theory is denoted by a pair $(\mathcal{G}, \mathcal{L})$, where $\mathcal{G}$ is the set of global assumptions and $\mathcal{L}$ is the set of local assumptions.

For a particular agent in some application, the *belief base* of the agent is a theory. There are no restrictions placed on belief bases. Each assumption in a belief base is called a *belief*. Typically, for agent j, local assumptions in its belief base have the form $\boldsymbol{B}_j \varphi$, with the intuitive meaning 'agent j believes φ'. Often φ is an equation. Other typical local assumptions have the form $\boldsymbol{B}_j \boldsymbol{B}_i \varphi$, meaning 'agent j believes that agent i believes φ'. Global assumptions in a belief base typically have the form φ, with no modalities at the front since the fact that they are global implicitly implies any sequence of (necessity) modalities effectively appears at the front. Thus, in general, beliefs commonly have the form $\boldsymbol{B}_{j_1} \cdots \boldsymbol{B}_{j_r} \varphi$, where $r \geq 0$. If there is a temporal component to beliefs, this is often manifested by temporal modalities at the front of beliefs. Then, for example, there could be a belief of the form $\bullet^2 \boldsymbol{B}_j \boldsymbol{B}_i \varphi$, whose intuitive meaning is 'at the second last time, agent j believed that agent i believed φ'. (Here, $\bullet^2$ is a shorthand for $\bullet\bullet$.) For more details about how we handle the representation and acquisition of beliefs, see [24–26].

3 Computation

In this section we study the case of (pure) computation.

3.1 Computations of rank 0

Consider the problem of determining the meaning of a term t in the intended pointed interpretation. If a formal definition of the intended pointed interpretation is available, then this

problem can be solved (under some finiteness assumptions). However, we assume here that the intended pointed interpretation is not available, as is usually the case, so that the problem cannot be solved directly. However, there is still a lot that can be done if the theory $\mathcal{T}$ of the application is available and enough of it is in equational form. Intuitively, if t can be 'simplified' sufficiently using $\mathcal{T}$, its meaning may become apparent even in the absence of detailed knowledge of the intended pointed interpretation. For example, if t can be simplified to a term containing only data constructors, then the meaning of t will be known since data constructors have a fixed meaning in every interpretation.

Informally, the *computation problem* is as follows.

> Given a theory $\mathcal{T}$, a term t, and a sequence $\Box_{j_1} \cdots \Box_{j_r}$ of modalities, find a 'simpler' term t' such that the formula $\Box_{j_1} \cdots \Box_{j_r} \forall (t = t')$ is a logical consequence of $\mathcal{T}$.

Thus t and t' have the same meaning in all worlds accessible from the actual world in the intended pointed interpretation according to the modalities $\Box_{j_1} \cdots \Box_{j_r}$.

Before proceeding to a formal treatment of how the computation problem is solved, it is helpful to look at a few examples to get a feel for the kind of problems we are interested in and the kind of answers we expect to get.

Example 6 Consider the following definition of $f : \sigma \to Int$.

$$(f\ x) = \textit{if } x = A \textit{ then } 42 \textit{ else if } x = B \textit{ then } 3 \textit{ else if } x = C \textit{ then } 21 \textit{ else } 0, \tag{2}$$

where $A, B, C : \sigma$. With such a definition, the system should have no difficulty computing the values of terms like $(f\ A)$ and $(f\ B)$. Less trivially, we want our system to be able to compute the value of a term like $\lambda x.((f\ x) > 5)$, that is, the set $\{x \mid (f\ x) > 5\}$. We should expect the computation system to return the answer $\{A, C\}$ in this case.

Example 7 Consider the following definition of *append*.

$$append : \ List\ a \times List\ a \times List\ a \to \Omega$$
$$(append\ (u, v, w)) = ((u = [] \wedge v = w)\ \vee$$
$$\exists r.\exists x.\exists y.((u = r \sharp x) \wedge (w = r \sharp y) \wedge (append\ (x, v, y)))) \tag{3}$$

The intended meaning of *append* is that it is true iff its third argument is the concatenation of its first two arguments. We should expect our system to be able to simplify a term like $(append\ ([1], [2], x))$ to $x = [1, 2]$, and a term like $(append\ (x, y, [1, 2]))$ to

$$(x = [] \wedge y = [1, 2]) \vee (x = [1] \wedge y = [2]) \vee (x = [1, 2] \wedge y = []).$$

Example 8 Consider a theory that includes definitions of the function $g : Int \to Int$ at the current time and some recent times.

$$\forall x.((g\ x) = \textit{if } (even\ x)\textit{ then } (\textit{if } (x < 6)\textit{ then } 21\textit{ else } (\bullet g\ x))\textit{ else } (\bullet^2 g\ x)) \tag{4}$$
$$\bullet\,\forall x.((g\ x) = \textit{if } (x > 0)\textit{ then } (\bullet g\ x)\textit{ else } 0) \tag{5}$$
$$\bullet^2\,\forall x.((g\ x) = 42) \tag{6}$$

A typical query we want to ask is the value of, say, $(g\ 12)$. We would expect the system to return the answer 42 in this case.

Example 9 Consider the following theory that contains a record of current and past statistics on the price of a commodity.

prices : *Density Real*

$$prices = (gaussian\ 400\ 20) \tag{7}$$

$$\bullet(prices = (gaussian\ 360\ 25)) \tag{8}$$

$$\bullet^2(prices = \lambda x.if\ x = 300\ then\ 0.7\ else\ if\ x = 310\ then\ 0.3\ else\ 0) \tag{9}$$

$$\bullet^3(prices = (gaussian\ 330\ 10)) \tag{10}$$

$$\bullet^4\blacksquare(prices = \lambda x.if\ x = 280\ then\ 1\ else\ 0) \tag{11}$$

gaussian : *Real* → *Real* → *Density Real*

$$(gaussian\ u\ s) = \lambda x.\frac{1}{s\sqrt{2\pi}}e^{-\frac{(x-u)^2}{2s^2}} \tag{12}$$

mean : (*Density a*) → *Real*

$$(mean\ (gaussian\ u\ s)) = u \tag{13}$$

$$(mean\ \lambda x.0) = 0 \tag{14}$$

$$(mean\ \lambda x.if\ x = u\ then\ y\ else\ \mathbf{w}) = y \times u + (mean\ \lambda x.\mathbf{w}) \tag{15}$$

$$\blacklozenge x = (x \vee \bullet\blacklozenge x). \tag{16}$$

Here, *mean* and *gaussian* are rigid constants whereas *prices* is not. An example query we might want to ask is

$$\blacklozenge((mean\ prices) < (mean\ \bullet prices)).$$

In other words, is there a period in the past where mean prices fell? Bach should return the answer $\top$ in this case. (The $\mathbf{w}$ in the third equation for *mean* is a syntactical variable; this is explained in more detail in the Appendix.)

Here now are the details of a mechanism that addresses the computation problem by employing equational reasoning to rewrite terms to 'simpler' terms that have the same meaning. We first establish some notation and terminology. The *occurrence o* of a subterm s in a term t is a description of the path from the root of t to s. We denote the subterm of t at occurrence o by $t|_o$. The notation $t[s/r]_o$ denotes the term obtained from t by replacing s at occurrence o with r. An occurrence of a variable x in a term is *bound* if it occurs within a subterm of the form $\lambda x.t$. Otherwise it is *free*. Suppose x is a variable. The notation $t\{x/r\}$ denotes the term obtained from t by replacing every free occurrence of variable x in t with r. A *modal path* to an occurrence is the sequence of indices of modalities whose scope is entered when going down to the occurrence. A substitution is *admissible* if any term that replaces a free occurrence of a variable that is in the scope of a modality is rigid.

Definition 4 Let $\mathcal{T} \equiv (\mathcal{G}, \mathcal{L})$ be a theory. A *computation of rank 0 using* $\Box_{j_1} \cdots \Box_{j_r}$ *with respect to* $\mathcal{T}$ is a sequence $\{t_i\}_{i=1}^n$ of terms such that the following holds. For $i = 1, \ldots, n - 1$, there is

1. a subterm s_i of t_i at occurrence o_i, where the modal path to o_i in t_i is $k_1 \ldots k_{m_i}$,
2. (a) a formula $\Box_{j_1} \cdots \Box_{j_r}\Box_{k_1} \cdots \Box_{k_{m_i}} \forall(u_i = v_i)$ in $\mathcal{L}$, or
   (b) a formula $\forall(u_i = v_i)$ in $\mathcal{G}$, and
3. a substitution θ_i that is admissible with respect to $u_i = v_i$

such that $u_i\theta_i$ is α-equivalent to s_i and t_{i+1} is $t_i[s_i/v_i\theta_i]_{o_i}$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

Each subterm s_i is called a *redex*.

Each formula $\Box_{j_1}\cdots\Box_{j_r}\Box_{k_1}\cdots\Box_{k_{m_i}}\forall(u_i = v_i)$ or $\forall(u_i = v_i)$ is called an *input equation*.

The formula $\Box_{j_1}\cdots\Box_{j_r}\forall(t_1 = t_n)$ is called the *result* of the computation.

We remark that the treatment of modalities in a computation has to be carefully handled. The reason is that even such a simple concept as applying a substitution is greatly complicated in the modal setting by the fact that constants generally have different meanings in different worlds and therefore the act of applying a substitution may not result in a term with the desired meaning. This explains the restriction to admissible substitutions in the definition of computation. It also explains why, for input equations that are local assumptions, the sequence of modalities $\Box_{k_1}\cdots\Box_{k_{m_i}}$ whose scopes are entered going down to the redex must appear in the modalities at the front of the input equation. (For input equations that are global assumptions, in effect, every sequence of modalities that we might need is implicitly at the front of the input equation.)

A *selection rule* chooses the redex at each step of a computation. A common selection rule is the *leftmost* one which chooses the leftmost outermost subterm that satisfies the requirements of Definition 4. The overall redex-selection strategy in Bach is leftmost-outermost reduction, which gives lazy evaluation. This is, however, not strictly followed. Input equations can be graded, in which case leftmost-outermost reduction is performed using only level 1 input equations to begin with and, in general, the selection rule only moves from level i to level $i + 1$ when no redex can be found using level i input equations. Fine-grained control over evaluation order can be achieved using this mechanism.

Theorem 1 establishes the soundness of the basic computation system; the proof can be found in [16, Proposition 6.1].

Theorem 1 *Let $\mathcal{T}$ be a theory. Then the result of a computation of rank 0 using $\Box_{j_1}\cdots\Box_{j_r}$ with respect to $\mathcal{T}$ is a logical consequence of $\mathcal{T}$.*

We do not specify a normal form for terms. Computation continues until there is no redex left. Theorem 1 shows that all the different terms that can be obtained by choosing different redexes at each step are equal to one another.

We also do not place restrictions on pattern matching. In particular, the input equations that together form a function definition can have overlapping patterns, as long as they are mutually consistent. (Some examples of overlapping patterns are given in the Appendix.) The responsibility for writing correct theories lies ultimately with the programmer.

3.2 Pattern matching

For the computation system introduced, given terms s and t, there is a need to determine whether or not there is a substitution θ such that $s\theta$ is α-equivalent to t.

Definition 5 Let s and t be terms of the same type. Then a substitution θ is a *matcher* of s to t if $s\theta$ is α-equivalent to t. In this case, s is said to be *matchable* to t.

The algorithm in Fig. 1 determines whether one term is matchable with another. Note that the inputs to this algorithm are two terms that have no free variables in common. It is usual to standardise apart before applying a unification algorithm so doing this for matching as well is not out of the ordinary.

---

function $Match(s, t)$ **returns** matcher θ, if s is matchable to t; failure, otherwise;

inputs: s and t, terms of the same type with no free variables in common;

$\theta := \{\}$;

while $s \neq t$ **do**

    $o :=$ occurrence of innermost subterm containing symbol at leftmost point of
        disagreement between s and t;

    if $s|_o$ has form $\lambda x.v$, $t|_o$ has form $\lambda y.w$, and $x \neq y$

    then

        $s := s[\lambda x.v/\lambda z.(v\{x/z\})]_o$; -- where z is a fresh variable

        $t := t[\lambda y.w/\lambda z.(w\{y/z\})]_o$;

    else if $s|_o$ is a free occurrence of a variable x and there is no free occurrence of x in s to
        the left of o and each free occurrence of a variable in $t|_o$ is a free occurrence in t

        then

            $\theta := \theta \circ \{x/t|_o\}$;

            $s := s\{x/t|_o\}$;

        else return failure;

return θ;

Fig. 1 Algorithm for finding a matching substitution

In the algorithm, $\theta \circ \{x/t|_o\}$ denotes the composition of θ with $\{x/t|_o\}$. Since only α-equivalence is required here, given a term v, we can compute $v(\theta \circ \varphi)$ by computing $(v\theta)\varphi$. The proof of the following result is given in [16, Proposition 2.15].

Theorem 2 *Let s and t be terms of the same type with no free variables in common. If s is matchable to t, then the algorithm in Fig. 1 terminates and returns a matcher of s to t. Otherwise, the algorithm terminates and returns failure.*

Here are three examples to illustrate the matching algorithm.

Example 10 Let s be $\lambda x.(f\ x\ (g\ y\ z))$ and t be $\lambda z.(f\ z\ (g\ A\ B))$, where f, g, A, and B are constants with suitable signatures. Then the successive steps of the algorithm are as follows.

0. $\lambda x.(f\ x\ (g\ y\ z))$ $\lambda z.(f\ z\ (g\ A\ B))$

1. $\lambda w.(f\ w\ (g\ y\ z))$ $\lambda w.(f\ w\ (g\ A\ B))$ $\{y/A\}$

2. $\lambda w.(f\ w\ (g\ A\ z))$ $\lambda w.(f\ w\ (g\ A\ B))$ $\{z/B\}$

3. $\lambda w.(f\ w\ (g\ A\ B))\ \lambda w.(f\ w\ (g\ A\ B))$

(The arrows indicate the points of disagreement and the substitutions in the last column are the substitutions applied at that step in the algorithm.) Thus $\lambda x.(f\ x\ (g\ y\ z))$ is matchable to $\lambda z.(f\ z\ (g\ A\ B))$ with matcher $\{y/A\} \circ \{z/B\}$.

Example 11 Let s be $(f\ x\ (g\ x))$ and t be $(f\ y\ (g\ A))$. Then the successive steps of the algorithm are as follows.

0. $(f\ x\ (g\ x))\ (f\ y\ (g\ A))\ \{x/y\}$

1. $(f\ y\ (g\ y))\ (f\ y\ (g\ A))$

Thus $(f\ x\ (g\ x))$ is not matchable to $(f\ y\ (g\ A))$, since there is a free occurrence of y in s to the left of the point of disagreement. Note that, in contrast, s and t are unifiable.

$(f\ B)$	[2]
if $\underline{B = A}$ then 42 else if $B = B$ then 3 else if $B = C$ then 21 else 0	[D2]
$\underline{if\ \bot\ then\ 42}$ else if $B = B$ then 3 else if $B = C$ then 21 else 0	[I2]
if $\underline{B = B}$ then 3 else if $B = C$ then 21 else 0	[D1]
$\underline{if\ \top\ then\ 3}$ else if $B = C$ then 21 else 0	[I1]
3	

Fig. 2 Computation of $(f\ B)$. Redexes are underlined. The input equation used at each step is also given. Equations D2, I2, D1 and I1 come from the standard equality theory given in the Appendix

Example 12 Let s be $\lambda x.(f\ x\ y\ z)$ and t be $\lambda x.(f\ x\ A\ (g\ x))$. Then the successive steps of the algorithm are as follows.

0. $\lambda x.(f\ x\ \underset{\uparrow}{y}\ z)\ \ \lambda x.(f\ x\ \underset{\uparrow}{A}\ (g\ x))\ \ \{y/A\}$

1. $\lambda x.(f\ x\ \underset{\uparrow}{A}\ z)\ \ \lambda x.(f\ x\ A\ \underset{\uparrow}{(g\ x)})$

Thus $\lambda x.(f\ x\ y\ z)$ is not matchable to $\lambda x.(f\ x\ A\ (g\ x))$, since x has a free occurrence in $(g\ x)$ but this occurrence is not free in $\lambda x.(f\ x\ A\ (g\ x))$.

3.3 Examples of computation

Here are a few examples to illustrate rank 0 computations. They show how the computation problems described in Examples 6–9 are solved. Example 14 illustrates how computations in the style of logic programming can be supported in the functional setting. This style of programming was already supported in Escher, the predecessor of Bach. The other examples all illustrate language features that are not available in Escher. These features are discussed in more detail in Sect. 7.

Computations generally require use of definitions of $=$, the connectives and quantifiers, and some other basic functions. These definitions, which constitute what we call the standard equality theory, are given in the Appendix.

Example 13 Consider the definition of f in Example 6. Figure 2 shows the computation of $(f\ B)$. This illustrates a standard functional computation. Figure 3 shows how the term $\lambda x.((f\ x) > 5)$ is simplified by Bach. This computation makes essential use of (I3) and (I4) from the standard equality theory.

Example 14 Consider the definition of *append* in Example 7, which has been written in the relational style of logic programming. Figure 4 shows the computation of $(append\ (1\,\sharp\,[],\ 2\,\sharp\,[],\ x))$. The notable feature of the *append* definition is the presence of existential quantifiers on the right hand side, so not surprisingly the key input equation needed to make it work is concerned with the existential quantifier. At one point in the computation shown in Fig. 4, the following term is reached:

$\exists r'.\exists x'.\exists y'.((1 = r') \wedge ([] = x')\ \wedge\ (x = r'\,\sharp\,y') \wedge (append\ (x',\ 2\,\sharp\,[],\ y'))).$

An obviously desirable simplification that can be made to this term is to eliminate the variable r' since there is a 'value' (that is, 1) for it. This leads to the term

$\exists x'.\exists y'.(([] = x') \wedge (x = 1\,\sharp\,y') \wedge (append\ (x',\ 2\,\sharp\,[],\ y'))).$

$$\lambda x. (> \underline{(f\ x)}\ 5) \tag{2}$$

$$\lambda x. (> \underline{(if\ x = A\ then\ 42\ else\ if\ x = B\ then\ 3\ else\ if\ x = C\ then\ 21\ else\ 0)}\ 5) \tag{I3}$$

$$\lambda x. (\underline{(if\ x = A\ then\ (> 42)\ else\ (> (if\ x = B\ then\ 3\ else\ if\ x = C\ then\ 21\ else\ 0)))}\ 5) \tag{I4}$$

$$\lambda x. (if\ x = A\ then\ \underline{(> 42\ 5)}\ else\ (> (if\ x = B\ then\ 3\ else\ if\ x = C\ then\ 21\ else\ 0)\ 5))$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ \underline{(> (if\ x = B\ then\ 3\ else\ if\ x = C\ then\ 21\ else\ 0)}\ 5)) \tag{I3}$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ \underline{((if\ x = B\ then\ (> 3)\ else\ (> (if\ x = C\ then\ 21\ else\ 0)))\ 5)}) \tag{I4}$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \underline{(> 3\ 5)}\ else\ (> (if\ x = C\ then\ 21\ else\ 0)\ 5))$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ \underline{(> (if\ x = C\ then\ 21\ else\ 0)}\ 5)) \tag{I3}$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ \underline{((if\ x = C\ then\ (> 21)\ else\ (> 0))\ 5)}) \tag{I4}$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \underline{(> 21\ 5)}\ else\ (> 0\ 5))$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \top\ else\ \underline{(> 0\ 5)})$$

$$\lambda x. (if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \top\ else\ \bot)$$

Fig. 3 Computation of $\lambda x. ((f\ x) > 5)$. Equations I3 and I4 come from the standard equality theory given in the Appendix

$$(append\ \underline{(1 \sharp [], 2 \sharp [], x)}) \tag{3}$$

$$(\underline{(1 \sharp [] = [])} \wedge (2 \sharp [] = x)) \vee \exists r'. \exists x'. \exists y'. ((1 \sharp [] = r' \sharp x') \wedge (x = r' \sharp y') \wedge (append\ (x', 2 \sharp [], y'))) \tag{D2}$$

$$(\underline{\bot \wedge (2 \sharp [] = x)}) \vee \exists r'. \exists x'. \exists y'. ((1 \sharp [] = r' \sharp x') \wedge (x = r' \sharp y') \wedge (append\ (x', 2 \sharp [], y'))) \tag{A2}$$

$$\underline{\bot \vee \exists r'. \exists x'. \exists y'. ((1 \sharp [] = r' \sharp x')} \wedge (x = r' \sharp y') \wedge (append\ (x', 2 \sharp [], y'))) \tag{O2}$$

$$\exists r'. \exists x'. \exists y'. (\underline{(1 \sharp [] = r' \sharp x')} \wedge (x = r' \sharp y') \wedge (append\ (x', 2 \sharp [], y'))) \tag{D1}$$

$$\exists r'. \exists x'. \exists y'. (\underline{(1 = r') \wedge ([] = x')} \wedge (x = r' \sharp y') \wedge (append\ (x', 2 \sharp [], y'))) \tag{E}$$

$$\exists x'. \exists y'. (\underline{([] = x') \wedge (x = 1 \sharp y') \wedge (append\ (x', 2 \sharp [], y'))}) \tag{E}$$

$$\exists y'. ((x = 1 \sharp y') \wedge \underline{(append\ ([], 2 \sharp [], y'))}) \tag{3}$$

$$\vdots$$

$$\exists y'. ((x = 1 \sharp y') \wedge \underline{(y' = 2 \sharp [])}) \tag{E}$$

$$x = 1 \sharp 2 \sharp []$$

Fig. 4 Computation of $(append\ (1 \sharp [], 2 \sharp [], x))$

Similarly, one can eliminate x' to obtain

$$\exists y'. ((x = 1 \sharp y') \wedge (append\ ([], 2 \sharp [], y'))).$$

After some more computation, the answer $x = 1 \sharp 2 \sharp []$ results. The input equation that makes all this possible is (E), which comes from the definition of $\Sigma : (a \rightarrow \Omega) \rightarrow \Omega$ in the standard equality theory and has λ-abstractions on the left hand side of the equation.

This example illustrates how the traditional functional programming setting can be extended by means of a carefully chosen set of equations to encompass the relational style of logic programming. This general technique is called *programming with abstractions* [15].

Another feature of Bach-style logic programming is that alternative answers are returned as a disjunction. Thus the goal $(append\ (x, y, 1 \sharp 2 \sharp []))$ will be reduced, using essentially the same operations shown in Fig. 4, to the answer

$\underline{(g\ 12)}$ [4]

$(if\ \underline{(even\ 12)}\ then\ (if\ (<\ 12\ 6)\ then\ 21\ else\ (\bullet g\ 12))\ else\ (\bullet^2 g\ 12))$

$\vdots$

$\underline{(if\ \top\ then\ (if\ (<\ 12\ 6)\ then\ 21\ else\ (\bullet g\ 12))\ else\ (\bullet^2 g\ 12))}$ [I1]

$(if\ \underline{(<\ 12\ 6)}\ then\ 21\ else\ (\bullet g\ 12))$

$\underline{(if\ \bot\ then\ 21\ else\ (\bullet g\ 12))}$ [I2]

$(\bullet g\ 12)$ [M1]

$\bullet \underline{(g\ 12)}$ [5]

$\bullet (if\ \underline{(>\ 12\ 0)}\ then\ (\bullet g\ 12)\ else\ 0)$

$\bullet (if\ \top\ then\ (\bullet g\ 12)\ else\ 0)$ [I1]

$\bullet (\bullet g\ 12)$ [M1]

$\bullet^2 \underline{(g\ 12)}$ [6]

$\bullet^2 42$ [M2]

$\bullet 42$ [M2]

42

Fig. 5 Computation of $(g\ 12)$

$$(x = []\ \wedge\ y = 1\,\sharp\,2\,\sharp\,[])\ \vee\ (x = 1\,\sharp\,[]\ \wedge\ y = 2\,\sharp\,[])\ \vee\ (x = 1\,\sharp\,2\,\sharp\,[]\ \wedge\ y = []).$$

Example 15 Consider the definition of g in Example 8. Figure 5 shows the computation of $(g\ 12)$. Note how earlier definitions for g get used in the computation: at the step $\bullet \underline{(g\ 12)}$, the definition at the last time step kicks in; at the step $\bullet^2 \underline{(g\ 12)}$, the definition from two time steps ago gets chosen. Also needed in this computation are the global assumptions (M1) and (M2) from the standard equality theory. This example showcases a typical modal computation. Support for such computations is not available in existing functional programming languages.

Example 16 Figure 6 shows the computation of $\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ using the program of Example 9. Among other things, the computation shows

1. how redexes made up of non-rigid terms can only be rewritten using definitions with the correct modal context;
2. how global assumptions can be used inside any modal context;
3. how probability densities can be manipulated using higher-order functions; and
4. how syntactical variables are used to process lambda abstractions.

4 Proof

In this section we study the case of (pure) proof.

$\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [16]

$((mean\ \underline{prices}) < (mean\ \bullet prices)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [7]

$((\underline{mean\ (gaussian\ 400\ 20)}) < (mean\ \bullet prices)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [13]

$(400 < (mean\ \underline{\bullet prices})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [8]

$(400 < (mean\ \underline{\bullet(gaussian\ 360\ 25)})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [M2]

$(400 < (mean\ \underline{(gaussian\ 360\ 25)})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [13]

$(\underline{400 < 360}) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$

$\underline{\bot \lor} \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [O2]

$\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [16]

$\bullet(((mean\ \underline{prices}) < (mean\ \bullet prices)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [8]

$\bullet(((\underline{mean\ (gaussian\ 360\ 25)}) < (mean\ \bullet prices)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [13]

$\bullet((360 < (mean\ \underline{\bullet prices})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [9]

$\bullet((360 < (mean\ \underline{\bullet \lambda x.if\ x = 300\ then\ 0.7\ else\ if\ x = 310\ then\ 0.3\ else\ 0})) \lor$
$\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [M2]

$\bullet((360 < (mean\ \underline{\lambda x.if\ x = 300\ then\ 0.7\ else\ if\ x = 310\ then\ 0.3\ else\ 0})) \lor$
$\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [15]

$\bullet((360 < (\underline{300 \times 0.7} + (mean\ \lambda x.if\ x = 310\ then\ 0.3\ else\ 0)) \lor$
$\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$

$\bullet((360 < 210 + (\underline{mean\ \lambda x.if\ x = 310\ then\ 0.3\ else\ 0})) \lor$
$\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [15]

$\bullet((360 < 210 + \underline{310 \times 0.3} + (mean\ \lambda x.0)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$

$\bullet((360 < 210 + 93 + (\underline{mean\ \lambda x.0})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [14]

$\bullet((360 < \underline{210 + 93 + 0}) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$

$\bullet((\underline{360 < 303}) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$

$\bullet(\underline{\bot \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))})$ [O2]

$\bullet\bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))$ [16]

$\bullet\bullet(((mean\ \underline{prices}) < (mean\ \bullet prices)) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [9]

$\vdots$

$\bullet\bullet((303 < (mean\ \underline{\bullet prices})) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$ [10]

$\vdots$

$\bullet\bullet((\underline{303 < 330}) \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices)))$

$\bullet\bullet(\underline{\top \lor \bullet\blacklozenge((mean\ prices) < (mean\ \bullet prices))})$ [O1]

$\bullet\bullet\underline{\top}$ [M2]

$\bullet\underline{\top}$ [M2]

$\top$

Fig. 6 Computation of $\blacklozenge((mean\ prices) < (mean\ \bullet prices))$

4.1 Proofs of rank 0

The *proof problem*, which is a companion to the previously discussed computation problem, is as follows.

Given a theory $\mathcal{T}$ and formula φ, determine whether φ is a logical consequence of $\mathcal{T}$.

Here are the details of a tableau proof system that, given a theory $\mathcal{T}$ and a formula φ, can determine whether φ is a logical consequence of $\mathcal{T}$. The system employs prefixed formulas as is often the case for modal logics.

Definition 6 A *prefix* is a finite sequence of the form $1.\langle n_1, j_1 \rangle. \ \ldots .\langle n_k, j_k \rangle$, where n_i is a positive integer and $j_i \in \{1, \ldots, m\}$, for $i = 1, \ldots, k$.
A *prefixed formula* is an expression of the form $\sigma \ \varphi$, where σ is a prefix and φ is a formula.

In the following, $\langle n, j \rangle$ is abbreviated to n_j.
We concentrate on the (multi-modal) logic $\mathbf{K}_m$ (m refers to the number of modalities) which has the tableau system given by the rules in Figs. 7 and 8. Generally speaking, these rules are well known (see, for example, [27,10]), but the versions here differ in some details, in particular, in the use of the admissibility assumption in several rules. For each type α, we assume the existence of a denumerable set $\mathcal{W}_\alpha$ of witness constants. These are used in the existential rules.

Definition 7 Let $\mathcal{T}$ be a theory. A *proof of rank* 0 *with respect to* $\mathcal{T}$ is a sequence $T_1, \ldots, T_n$ of trees labelled by prefixed formulas satisfying the following conditions.

1. T_1 consists of a single node labelled by $1 \ \neg\varphi$, for some formula φ.
2. For $i = 1, \ldots, n - 1$, there is

    (a) a tableau rule R from Fig. 7 or Fig. 8 such that T_{i+1} is obtained from T_i,
        i. if R is a conjunctive rule, by extending a branch with two nodes labelled by the prefixed formulas in the denominator of R,
        ii. if R is a disjunctive rule, by splitting a branch so that the leaf node of the branch has two children each labelled by one of the prefixed formulas in the denominator of R,
        iii. otherwise, by extending a branch with a node labelled by the prefixed formula in the denominator of R,
        provided that any prefixed formulas in the numerator of R already appear in the branch and any side-conditions of R are satisfied.

3. Each branch of T_n contains nodes labelled by $\sigma \ \psi$ and $\sigma \ \neg\psi$, for some prefix σ and formula ψ.

    Each T_i is called a *tableau of rank* 0.
    A branch of a tableau of rank 0 is *closed* if it contains nodes labelled by $\sigma \ \psi$ and $\sigma \ \neg\psi$, for some prefix σ and formula ψ; otherwise, the branch is *open*.
    A tableau of rank 0 is *closed* if each branch is closed; otherwise, the tableau is *open*.
    The formula φ is called the *theorem* of the proof.

    The following soundness result is proved in [16, Proposition 6.5].

Theorem 3 *Let $\mathcal{T}$ be a theory. Then the theorem of a proof of rank* 0 *with respect to* $\mathcal{T}$ *is a logical consequence of* $\mathcal{T}$.

4.2 Tableaux expansion algorithm

The tableaux rules given in Figs. 7 and 8 are non-deterministic: they specify what *may* be done, but not what *must* be done. There is of course no general decision procedure for the logic. Here we present a sound, terminating but incomplete tableaux-expansion algorithm guided by standard heuristics.
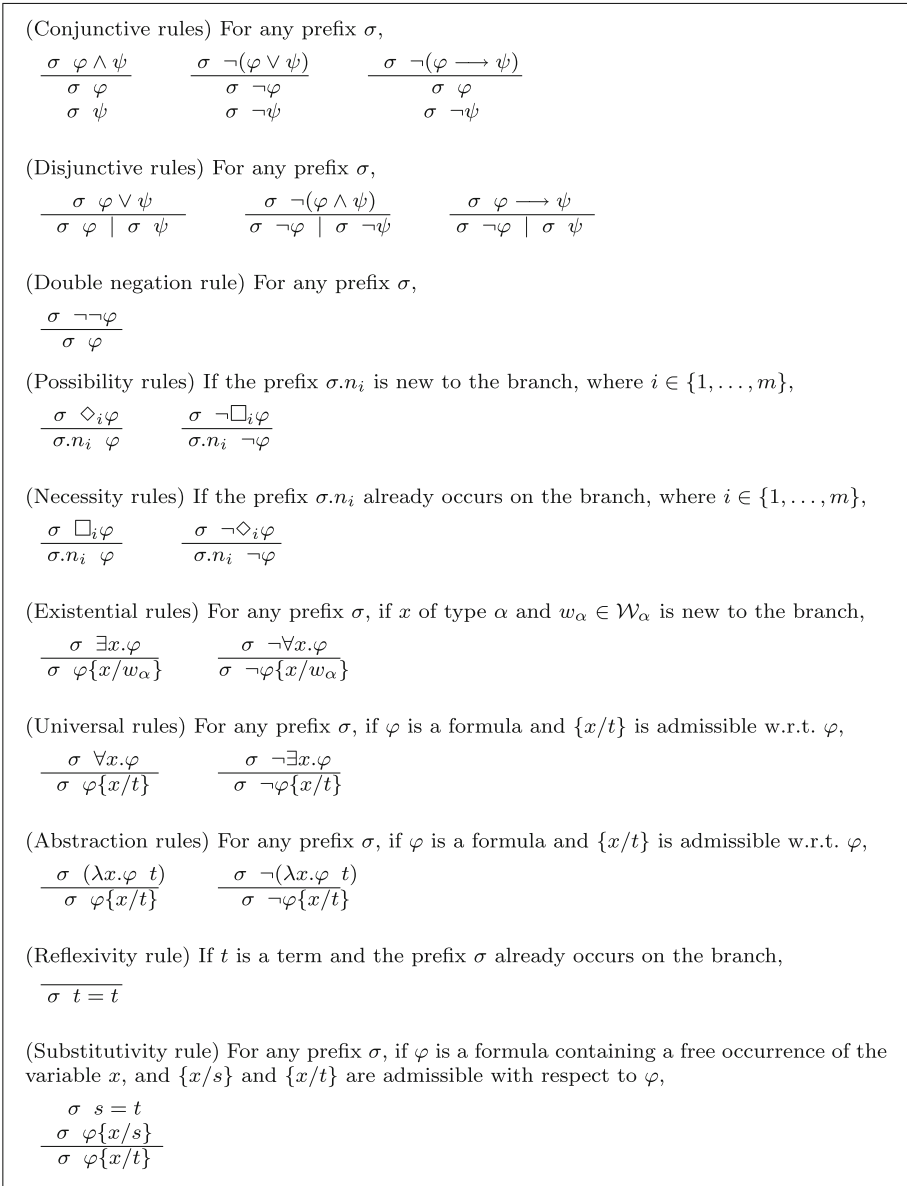
(Conjunctive rules) For any prefix σ,

$$\frac{\sigma \quad \varphi \wedge \psi}{\begin{array}{c} \sigma \quad \varphi \\ \sigma \quad \psi \end{array}} \qquad \frac{\sigma \quad \neg(\varphi \vee \psi)}{\begin{array}{c} \sigma \quad \neg\varphi \\ \sigma \quad \neg\psi \end{array}} \qquad \frac{\sigma \quad \neg(\varphi \longrightarrow \psi)}{\begin{array}{c} \sigma \quad \varphi \\ \sigma \quad \neg\psi \end{array}}$$

(Disjunctive rules) For any prefix σ,

$$\frac{\sigma \quad \varphi \vee \psi}{\sigma \quad \varphi \mid \sigma \quad \psi} \qquad \frac{\sigma \quad \neg(\varphi \wedge \psi)}{\sigma \quad \neg\varphi \mid \sigma \quad \neg\psi} \qquad \frac{\sigma \quad \varphi \longrightarrow \psi}{\sigma \quad \neg\varphi \mid \sigma \quad \psi}$$

(Double negation rule) For any prefix σ,

$$\frac{\sigma \quad \neg\neg\varphi}{\sigma \quad \varphi}$$

(Possibility rules) If the prefix $\sigma.n_i$ is new to the branch, where $i \in \{1, \ldots, m\}$,

$$\frac{\sigma \quad \Diamond_i\varphi}{\sigma.n_i \quad \varphi} \qquad \frac{\sigma \quad \neg\Box_i\varphi}{\sigma.n_i \quad \neg\varphi}$$

(Necessity rules) If the prefix $\sigma.n_i$ already occurs on the branch, where $i \in \{1, \ldots, m\}$,

$$\frac{\sigma \quad \Box_i\varphi}{\sigma.n_i \quad \varphi} \qquad \frac{\sigma \quad \neg\Diamond_i\varphi}{\sigma.n_i \quad \neg\varphi}$$

(Existential rules) For any prefix σ, if x of type α and $w_\alpha \in \mathcal{W}_\alpha$ is new to the branch,

$$\frac{\sigma \quad \exists x.\varphi}{\sigma \quad \varphi\{x/w_\alpha\}} \qquad \frac{\sigma \quad \neg\forall x.\varphi}{\sigma \quad \neg\varphi\{x/w_\alpha\}}$$

(Universal rules) For any prefix σ, if φ is a formula and $\{x/t\}$ is admissible w.r.t. φ,

$$\frac{\sigma \quad \forall x.\varphi}{\sigma \quad \varphi\{x/t\}} \qquad \frac{\sigma \quad \neg\exists x.\varphi}{\sigma \quad \neg\varphi\{x/t\}}$$

(Abstraction rules) For any prefix σ, if φ is a formula and $\{x/t\}$ is admissible w.r.t. φ,

$$\frac{\sigma \quad (\lambda x.\varphi \ t)}{\sigma \quad \varphi\{x/t\}} \qquad \frac{\sigma \quad \neg(\lambda x.\varphi \ t)}{\sigma \quad \neg\varphi\{x/t\}}$$

(Reflexivity rule) If t is a term and the prefix σ already occurs on the branch,

$$\frac{}{\sigma \quad t = t}$$

(Substitutivity rule) For any prefix σ, if φ is a formula containing a free occurrence of the variable x, and $\{x/s\}$ and $\{x/t\}$ are admissible with respect to φ,

$$\frac{\begin{array}{c} \sigma \quad s = t \\ \sigma \quad \varphi\{x/s\} \end{array}}{\sigma \quad \varphi\{x/t\}}$$

Fig. 7 Basic tableau rules

Our algorithm, which takes into consideration issues discussed in [28–30], is as follows. We start from the initial tableau T_0 consisting of only the prefixed formula 1 $\neg\varphi$, where φ is the formula to be proved. We compute the tableau T_{i+1} from T_i by applying successively the following steps:

1. Classical saturation step: apply the classical tableaux rules on all the prefixed formulas in the tableau as much as possible.

(Global assumption rule) If ψ is a global assumption and the prefix σ already occurs on the branch,

$$\overline{\sigma \ \ \psi}$$

(Local assumption rule) If ψ is a local assumption,

$$\overline{1 \ \ \psi}$$

(Derived rules for global implicational assumption) For any prefix σ, if $\varphi \longrightarrow \psi$ is a global assumption,

$$\frac{\sigma \ \ \varphi}{\sigma \ \ \psi} \qquad \frac{\sigma \ \ \neg\psi}{\sigma \ \ \neg\varphi}$$

(Derived rules for local implicational assumption) If $\varphi \longrightarrow \psi$ is a local assumption,

$$\frac{1 \ \ \varphi}{1 \ \ \psi} \qquad \frac{1 \ \ \neg\psi}{1 \ \ \neg\varphi}$$
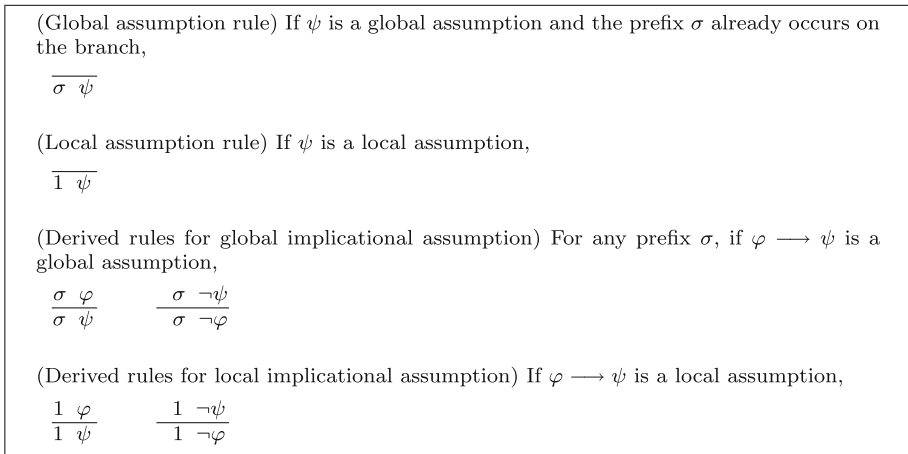
Fig. 8 More rules. A derived rule is one such that any application of it can be translated into a sequence of applications of the basic rules

2. Structural step: apply the structural rules on each prefixed formula in a non-loop world. A world is defined as a loop world iff all of its prefixed formulas are contained in some ancestral world in the accessibility relation.
3. Propagation step: apply the propagation rules as much as possible.

The above algorithm is applied until for some i, either T_i is closed or $T_{i+1} = T_i$.

The three kinds of rules mentioned in the algorithm are distinguished in [28]. The classical rules are made up of the conjunction, disjunction, double negation, existential, and universal rules. Propagation rules have the following general formulation: if there is a certain formula φ in a node having a certain pattern, then propagate a formula (either φ or some other one). Structural rules, in contrast, have the following general formulation: if there is such a pattern then add some new node(s) and edge(s). Examples of propagation rules include the necessity rules and tableaux rules for implementing modal axioms like T, 4, B and 5. Examples of structural rules include the possibility rules and tableaux rules for implementing modal axioms like D, De, and C. Tableaux rules for the different modal axioms mentioned above are omitted here because we have yet to find them useful for the kind of applications studied in this paper. We could have opted for a more specialised tableaux algorithm given that the possibility and necessity rules are the only modal rules currently needed. Our algorithm is however designed with generality in mind to accommodate potential future needs.

The main difficulty in operationalising the tableau system lies with the universal rules. These rules allow the introduction of new terms into a proof but there is potentially an infinite number of candidates and obviously some choices will be better than others. How do we decide in general? A standard technique to deal with this is to delay the choice by first introducing a free variable and use unification later to choose a value that would allow the system to close a branch. To achieve this, we replace the universal rules with those in Fig. 9 and use a more complex tableaux closure rule that not only checks for contradicting pairs $\sigma \ \psi$ and $\sigma \ \neg\psi$, for some σ and ψ, but also search for pairs $\sigma \ \psi$ and $\sigma \ \neg\varphi$ and admissible substitutions θ such that $\psi\theta$ and $\varphi\theta$ are α-equivalent. The general setting of higher-order unification only requires that $\psi\theta$ and $\varphi\theta$ are equivalent under β reductions. Higher-order unification is, however, undecidable [31]. We have opted for a simple unification (without β-reduction) here for efficiency reasons. This works fine for our target applications but there

(FV Universal rules) For any prefix σ, if φ is a formula and y is a fresh variable,

$$\frac{\sigma \quad \forall x.\varphi}{\sigma \quad \varphi\{x/y\}} \qquad \frac{\sigma \quad \neg\exists x.\varphi}{\sigma \quad \neg\varphi\{x/y\}}$$

Fig. 9 Universal rules for free variable tableaux

are clear limitations. For example, we cannot at present prove Cantor's theorem in the style of [30], which requires Huet's algorithm [32]. This part of Bach can be redesigned as suggested in [30] should the need arise.

Our algorithm for the closure rule is motivated by [33]. At the end of each tableaux expansion step, a variable-assignment problem is constructed as follows. We first compute the substitutions that can be used to close each branch of the tableaux using a version of the *Match* algorithm (Fig. 1) that performs two-way matching of terms. Each substitution so obtained is then made into a conjunction of variable assignments. E.g., a substitution like $\theta \equiv \{x_1/t_1, x_2/t_2, x_3/t_3\}$ is turned into c_θ:

$$(x_1 = t_1) \wedge (x_2 = t_2) \wedge (x_3 = t_3).$$

The collection of such variable assignments for each branch are then joined together disjunctively to form branch constraints on the variables. Finally, the branch constraints are put together conjunctively and prefixed with existential quantifiers on the relevant free variables to form the overall variable-assignment problem for the tableaux. For example, suppose a tableaux has three branches where $\{\theta_1, \theta_2\}$, $\{\theta_3\}$, and $\{\theta_4, \theta_5\}$ are the substitutions computed for the respective branches. The overall assignment problem we will obtain for this tableaux is

$$\exists x_1. \cdots \exists x_n.((c_{\theta_1} \vee c_{\theta_2}) \wedge c_{\theta_3} \wedge (c_{\theta_4} \vee c_{\theta_5})), \tag{17}$$

where $x_1, \ldots, x_n$ are the free variables that appear in the domains of the θ_i's. The tableaux is closable if (17) evaluates to $\top$. We use the computation system to solve variable-assignment problems.

The variable-assignment problem for a tableaux obtained via the above procedure corresponds directly to a collection of unification problems. If any one of these unification problems can be solved, then the resultant unifier can be used to close the tableaux.

To ensure termination, a bound is usually put on the number of times the universal rules can be applied in a tableaux. An iterative-deepening style algorithm can then be used to achieve search efficiency.

There remains one other issue. Witness constants can be introduced into the tableaux by the existential rules. If the φ in the numerator of an existential rule contains free variables *Free*(φ) introduced by the universal rules, then there is a dependency between the new witness constant w_α to be introduced and the variables in *Free*(φ). In particular, this means the witness constant w_α may not appear in any term used to instantiate any of the variables in *Free*(φ). This problem is handled in first-order tableaux systems using Skolem functions. Naive Skolemisation is, however, unsound in higher-order logic [34]. A simple solution around this is to augment the tableaux with the maintenance of a partial function R (called a variable condition in [30]) mapping witness constants to sets of variables. The existential rules now update the current definition of R with $R(w_\alpha) = Free(\varphi)$ after every application. The closure rule would then only search for admissible substitutions θ satisfying the following: for every variable x in the domain of θ, $x\theta$ does not contain a witness constant w such that $x \in R(w)$.

$1 \; \neg(\forall x.(Q \; x) \longrightarrow \exists x.\forall y.\neg((P \; y) \wedge \neg((P \; x) \wedge (Q \; x))))$ 1.
$1 \; \forall x.(Q \; x)$ 2.
$1 \; \neg\exists x.\forall y.\neg((P \; y) \wedge \neg((P \; x) \wedge (Q \; x)))$ 3.
$1 \; (Q \; x_1)$ 4.
$1 \; \neg\forall y.\neg((P \; y) \wedge \neg((P \; x_2) \wedge (Q \; x_2)))$ 5.
$1 \; \neg\neg((P \; w_{y_1}) \wedge \neg((P \; x_2) \wedge (Q \; x_2)))$ 6.
$1 \; (P \; w_{y_1}) \wedge \neg((P \; x_2) \wedge (Q \; x_2))$ 7.
$1 \; (P \; w_{y_1})$ 8.
$1 \; \neg((P \; x_2) \wedge (Q \; x_2))$ 9.

$1 \; \neg(P \; x_2)$ 10.
$1 \; \neg\forall y.\neg((P \; y) \wedge \neg((P \; x_3) \wedge (Q \; x_3)))$ 12.
$1 \; \neg\neg((P \; w_{y_2}) \wedge \neg((P \; x_3) \wedge (Q \; x_3)))$ 13.
$1 \; (P \; w_{y_2}) \wedge \neg((P \; x_3) \wedge (Q \; x_3))$ 14.
$1 \; (P \; w_{y_2})$ 15.
$1 \; \neg((P \; x_3) \wedge (Q \; x_3))$ 16.

$1 \; \neg(Q \; x_2)$ 11.
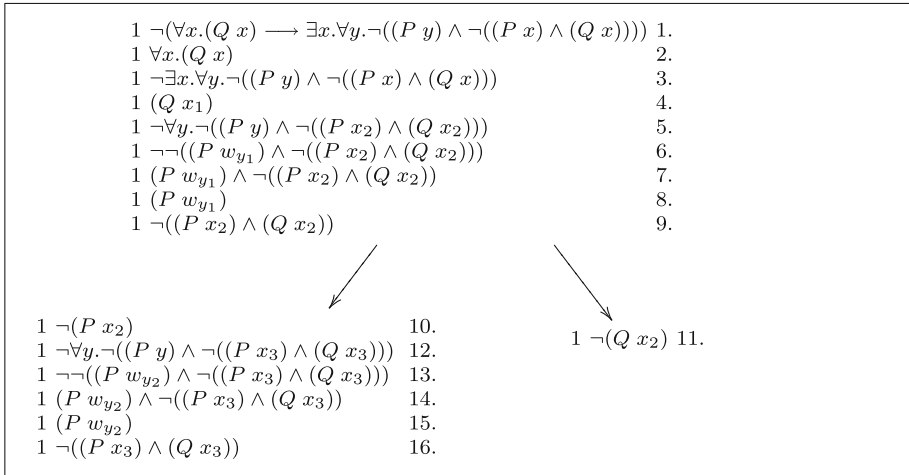
Fig. 10 Proof of $\forall x.(Q \; x) \longrightarrow \exists x.\forall y.\neg((P \; y) \wedge \neg((P \; x) \wedge (Q \; x)))$

4.3 Examples of proof

We look at some examples of proof in this section. Example 17 is taken from an exercise in [29] and serves to illustrate the basic mechanisms of the theorem prover. Example 18 shows a simple formula that can be proved using the theorem prover but not through a computation of rank 0. Example 19 shows how modal interaction axioms are handled.

Example 17 Let $P : \alpha \to \Omega$ and $Q : \alpha \to \Omega$ be two predicates. Figure 10 gives a proof of

$$\forall x.(Q \; x) \longrightarrow \exists x.\forall y.\neg((P \; y) \wedge \neg((P \; x) \wedge (Q \; x))).$$

An explanation of the proof is as follows. Item 1 is the negation of the formula to be proved; 2 and 3 are from 1 by a conjunctive rule; 4 is from 2 by a universal rule; 5 is from 3 by a universal rule; 6 is from 5 by an existential rule; 7 is from 6 by the double negation rule; 8 and 9 are from 7 by a conjunctive rule; 10 and 11 are from 9 by a disjunctive rule; 12 is from 3 by a universal rule; 13 is from 12 by an existential rule; 14 is from 13 by the double negation rule; and finally, 15 and 16 are from 14 by a conjunctive rule. At this stage, the variable condition is as follows:

$$R(w_{y_1}) = \{x_2\}, \; R(w_{y_2}) = \{x_3\}.$$

The tableaux can now be closed because the variable-assignment problem

$$\exists x_1.\exists x_2.(x_2 = w_{y_2} \wedge x_1 = x_2)$$

obtained from the tableaux ($x_2 = w_{y_2}$ from 10 and 15 and $x_1 = x_2$ from 4 and 11) can be shown to be true.

Example 18 Computations of rank 0 can be used to prove simple theorems like

$$\forall x.((x = A \vee x = B) \longrightarrow x \neq C). \tag{18}$$

Equations U1 and U2 are mainly what are needed. But the theorem-proving capability of rank 0 computations is inherently limited. For example, the following simple modification

Fig. 11 Proof of $B\bullet^2\varphi$

1	$\neg B\bullet^2\varphi$	1.
1	$\bullet^2 B\varphi$	2.
1	$\neg\bullet B\bullet\varphi$	3.
1.1$_\bullet$	$\neg B\bullet\varphi$	4.
1.1$_\bullet$	$\neg\bullet B\varphi$	5.
1.1$_\bullet$	$\bullet B\varphi$	6.

of (18) cannot be proved using rank 0 computations:

$$\forall x.(((proj_2\ x) = A \vee (proj_2\ x) = B) \longrightarrow (proj_2\ x) \neq C). \tag{19}$$

Here x is a tuple and $proj_2$ is a function that projects onto the second element of x. A proof of (19) can be easily constructed with the tableaux prover.

Example 19 Suppose we have a theory that includes the following

$$\bullet B\varphi_1, \quad \bullet^2 B\varphi_2, \quad \bullet^3 B\varphi_3, \quad \bullet^4 B\varphi_4, \quad \bullet^5 B\varphi_5$$

as local assumptions. Using the global assumption

$$\bullet B\varphi \longrightarrow B\bullet\varphi, \tag{20}$$

it can be shown that, for each $i \in \{1, \ldots, 5\}$, $B\bullet^i\varphi_i$ is a theorem of the belief base. For example, Fig. 11 shows the proof of $B\bullet^2\varphi_2$. Item 1 is the negation of the formula to be proved; 2 is a local assumption; 3 is from 1 by a derived rule from the global assumption (20); 4 is from 3 by a possibility rule; 5 is from 4 by a derived rule from (20); 6 is from 2 by a necessity rule; the tableau now closes by 5 and 6.

We often assume an agent has assumption (20) in its belief base. Informally, the meaning of this assumption is 'if, at the last time, the agent believed φ, then the agent believes (now) that φ held at the last time'.

4.4 Remarks on the theorem prover

The theorem prover plays a subsidiary role in Bach. It is used primarily to handle formulas involving universal quantifiers and implications, both of which are only weakly supported in rank 0 computations. In Sect. 5, we will see how the theorem prover can be used to augment the equational reasoning mechanism of Bach to automatically perform rather complex computation tasks. A more common use of the theorem prover is as an interactive support tool during program development. Often, in writing a Bach program, one can come up with certain non-trivial equations that, if true, can be used to either speed up computations or transform results into more convenient forms. The theorem prover can sometimes be used to verify the correctness of such formulas. A more mature system like Isabelle/HOL [35] can do this job better, but such systems do not currently deal with modalities.

5 Computation and proof

This section defines the combination of proof and computation, and shows the usefulness of this combination. Computation enhances proof with a powerful equational reasoning system; proof enhances computation by allowing some of the theory to be not in equational form.

5.1 Computations and proofs of rank k

By means of two mutually recursive definitions, the concepts of computation of rank k and proof of rank k are defined, for $k \geq 1$.

Definition 8 Let $\mathcal{T} \equiv (\mathcal{G}, \mathcal{L})$ be a theory and $k \geq 1$. A *computation of rank k using* $\Box_{j_1} \cdots \Box_{j_r}$ *with respect to* $\mathcal{T}$ is a sequence $\{t_i\}_{i=1}^n$ of terms such that, for $i = 1, \ldots, n - 1$, there is

1. a subterm s_i of t_i at occurrence o_i, where the modal path to o_i in t_i is $k_1 \ldots k_{m_i}$,
2. (a) a formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall (u_i = v_i)$ in $\mathcal{L}$, or
   (b) a formula $\forall (u_i = v_i)$ in $\mathcal{G}$, or
   (c) a formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall (u_i = v_i)$ that is the result of a computation of rank $k - 1$ using $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}}$ with respect to $\mathcal{T}$, or
   (d) a formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall (u_i = v_i)$ that is the theorem of a proof of rank $k - 1$ with respect to $\mathcal{T}$, and
3. a substitution θ_i that is admissible with respect to $u_i = v_i$

such that $u_i \theta_i$ is α-equivalent to s_i and t_{i+1} is $t_i[s_i / v_i \theta_i]_{o_i}$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

Each subterm s_i is called a *redex*.

Each formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall (u_i = v_i)$ or $\forall (u_i = v_i)$ in Part 2 of the definition is called an *input equation*.

The formula $\Box_{j_1} \cdots \Box_{j_r} \forall (t_1 = t_n)$ is called the *result* of the computation.

Definition 9 Let $\mathcal{T} \equiv (\mathcal{G}, \mathcal{L})$ be a theory and $k \geq 1$. A *proof of rank k with respect to* $\mathcal{T}$ is a sequence $T_1, \ldots, T_n$ of trees labelled by prefixed formulas satisfying the following conditions.

1. T_1 consists of a single node labelled by $1 \ \neg\varphi$, for some formula φ.
2. For $i = 1, \ldots, n - 1$, there is either

   (a) a tableau rule R from Fig. 7 or Fig. 8 such that T_{i+1} is obtained from T_i,
      i. if R is a conjunctive rule, by extending a branch with two nodes labelled by the prefixed formulas in the denominator of R,
      ii. if R is a disjunctive rule, by splitting a branch so that the leaf node of the branch has two children each labelled by one of the prefixed formulas in the denominator of R,
      iii. otherwise, by extending a branch with a node labelled by the prefixed formula in the denominator of R,
      provided that any prefixed formulas in the numerator of R already appear in the branch and any side-conditions of R are satisfied; or
   (b) there is a theorem η of a proof of rank $k - 1$ and a branch is extended with the prefixed formula $1 \ \eta$; or
   (c) there is a result η of a computation of rank $k - 1$ and a branch is extended with the prefixed formula $1 \ \eta$.

3. Each branch of T_n contains nodes labelled by $\sigma \ \psi$ and $\sigma \ \neg\psi$, for some prefix σ and formula ψ.

   Each T_i is called a *tableau of rank k*.

   A branch of a tableau of rank k is *closed* if it contains nodes labelled by $\sigma \ \psi$ and $\sigma \ \neg\psi$, for some prefix σ and formula ψ; otherwise, the branch is *open*.

A tableau of rank k is *closed* if each branch is closed; otherwise, the tableau is *open*. The formula φ is called the *theorem* of the proof.

Note that a computation of rank k is a computation of rank k' and a proof of rank k is a proof of rank k', for all $k' > k$.

Definition 10 Let $\mathcal{T}$ be a theory.

A *computation with respect to* $\mathcal{T}$ is a computation using $\Box_{j_1} \cdots \Box_{j_r}$ of rank k with respect to $\mathcal{T}$, for some $j_1 \ldots j_r$ and $k \geq 0$.

A *proof with respect to* $\mathcal{T}$ is a proof of rank k with respect to $\mathcal{T}$, for some $k \geq 0$.

The proof of the following result is given in [16, Proposition 6.9].

Theorem 4 *Let $\mathcal{T}$ be a theory. Then the following hold.*

1. *The result of a computation with respect to $\mathcal{T}$ is a logical consequence of $\mathcal{T}$.*
2. *The theorem of a proof with respect to $\mathcal{T}$ is a logical consequence of $\mathcal{T}$.*

We reiterate that, while the definitions of computation and proof of rank k are given in their most general form, in practice, for Bach, use of the proof component is restricted. Condition 2(d) in Definition 8 stipulates that a computation can rely on results obtained from the proof system in carrying out a computation step. This is difficult to realise in practice as it entails a need to search at run-time, both for conjectures that can be used as input equations and the proofs of such conjectures. We remark that the condition is stated in its most general form here to illustrate the potential of such interactions between computation and proof. In practice, rather specific search algorithms are used for problems that arise naturally in applications. We look at two such examples in the following to see how this might work.

5.2 Switching modalities

We show how interaction axioms are handled in this section. Consider the following theory containing definitions for a function $f : Int \rightarrow \Omega$.

$$\boldsymbol{B} \, \forall x.((f \ x) = (even \ x) \wedge (\bullet f \ x)) \tag{21}$$

$$\bullet \boldsymbol{B} \, \forall x.((f \ x) = (perfect \ x)) \tag{22}$$

$$\bullet \boldsymbol{B} \varphi \longrightarrow \boldsymbol{B} \bullet \varphi \tag{23}$$

Suppose we want to compute the value of the term $(f \ 28)$ in the context of $\boldsymbol{B}$. After a few computation steps, we arrive at the term $(\bullet f \ 28)$ and are apparently stuck. We can, however, make progress by calling the theorem prover to show that

$$\boldsymbol{B} \bullet \forall x.((f \ x) = (perfect \ 28))$$

is a logical consequence of the theory and use that as an input equation to continue.

The algorithm to automate this process works as follows. Suppose t is the current term we are trying to compute using $\Box_{i_1} \cdots \Box_{i_l}$, for some $l \geq 0$. For every subterm s of t at occurrence o, where the modal path to o in t is $j_1 \ldots j_m$, for some $m \geq 0$, if there exists an input equation $\Box_{k_1} \cdots \Box_{k_{l+m}} \forall (u = v)$ in the theory such that

1. there exists an admissible substitution θ with respect to $(u = v)$ such that $u\theta$ is α-equivalent to s; and
2. $\Box_{i_1} \cdots \Box_{i_l} \Box_{j_1} \cdots \Box_{j_m}$ is a permutation of $\Box_{k_1} \cdots \Box_{k_{l+m}}$; and
3. $\Box_{i_1} \cdots \Box_{i_l} \Box_{j_1} \cdots \Box_{j_m} \forall (u = v)$ is a logical consequence of the theory,

$$\frac{(f\ 28)}{(even\ 28) \wedge (\bullet f\ 28)} \qquad\qquad [21]$$

$$\vdots$$

$$\frac{\top \wedge (\bullet f\ 28)}{(\bullet f\ 28)} \qquad\qquad [M1]$$

$$\frac{}{\bullet(f\ 28)} \qquad\qquad [SM, 24]$$

	1	$\neg \boldsymbol{B}\bullet \forall x.((f\ x) = (perfect\ x))$	1.
	1	$\bullet \boldsymbol{B} \forall x.((f\ x) = (perfect\ x))$	2.
	1	$\neg\bullet \boldsymbol{B} \forall x.((f\ x) = (perfect\ x))$	3.

$$\bullet(perfect\ 28)$$

$$\vdots$$

$$\frac{\bullet\top}{\top} \qquad\qquad [M2]$$
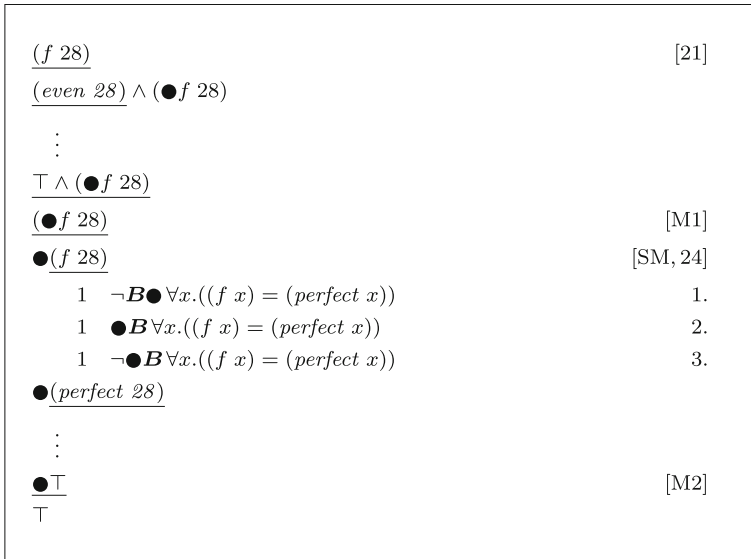
Fig. 12 Computation of rank 1 using $\boldsymbol{B}$ of $(f\ 28)$

then rewrite t to $t[s/v\theta]_o$. This algorithm is only called when no other rewrite rules can be applied.

Figure 12 shows how $(f\ 28)$ is simplified to $\top$ in the modal context $\boldsymbol{B}$. At the step marked SM, the above algorithm kicks in and the conjecture

$$\boldsymbol{B}\bullet\forall x.((f\ x) = (perfect\ x)) \tag{24}$$

is formulated and then proved. (The proof proceeds in a similar way to that shown in Example 19.) This theorem is then used as an input equation to continue the computation.

5.3 Formula simplification

Consider the problem of simplifying the term

$$((proj_1\ x) < 10) \wedge ((proj_2\ x) = 496)\ \wedge$$
$$(evenperfect\ (proj_2\ x)) \wedge ((proj_1\ x) \neq (proj_2\ x)), \tag{25}$$

where x is a tuple and $evenperfect : Int \to \Omega$ is defined by

$$(evenperfect\ x) = \exists n.(n \in \mathbb{N} \wedge (x = 2^{n-1} \times (2^n - 1)) \wedge (prime\ (2^n - 1))).$$

Other than expanding out the second conjunct, there is not much else one can do using just the mechanism available for rank 0 computations. However, we can show using the proof system that

$$\forall x.(((proj_2\ x) = 496) \longrightarrow (evenperfect\ (proj_2\ x)))\ \text{and} \tag{26}$$
$$\forall x.(((proj_1\ x) < 10) \wedge ((proj_2\ x) = 496) \longrightarrow (proj_1\ x) \neq (proj_2\ x)). \tag{27}$$

The proof of (26) is in Fig. 13. Item 1 is the negation of the formula to be proved; 2 is from 1 by an existential rule; 3 and 4 are from 2 by a conjunctive rule; 5 is from 3 and 4 by the

$$
\begin{array}{ll}
1 \ \neg\forall x.(((proj_2 \ x) = 496) \longrightarrow (evenperfect \ (proj_2 \ x))) & 1.\\
1 \ \neg((proj_2 \ y) = 496) \longrightarrow (evenperfect \ (proj_2 \ y)) & 2.\\
1 \ ((proj_2 \ y) = 496) & 3.\\
1 \ \neg(evenperfect \ (proj_2 \ y)) & 4.\\
1 \ \neg(evenperfect \ 496) & 5.\\
1 \ \neg(evenperfect \ 496) = \bot & 6.\\
1 \ \bot & 7.
\end{array}
$$

Fig. 13 Proof of (26)

substitutivity rule; 6 is from 5 by a rank 0 computation; 7 is from 5 and 6 by the substitutivity rule; the tableau now closes by 7. The proof of (27) is similar and is omitted.

Recognising that $(p \longrightarrow q) = ((p \wedge q) = p)$ is valid, we can construct two new input equations from (26) and (27) to simplify (25) to $((proj_1 \ x) < 10) \wedge ((proj_2 \ x) = 496)$.

The following is a general mechanism we can use to simplify formulas of the form $t_1 \wedge \cdots \wedge t_n$, $n \geq 2$, that cannot be dealt with using computations of rank 0 alone. Let $t_{\bar{i}}$, for $i \in \{1, \ldots, n\}$, denote $t_1 \wedge \cdots \wedge t_{i-1} \wedge t_{i+1} \wedge \cdots \wedge t_n$. We try to prove all possible formulas of the form $\forall(t_{\bar{i}} \longrightarrow t_i)$, $i \in \{1, \ldots, n\}$. If we can prove $\forall(t_{\bar{i}} \longrightarrow t_i)$, then we are entitled to remove t_i from $t_1 \wedge \cdots \wedge t_n$.

Computation problems of the kind just discussed arise naturally in several kinds of agent applications, including belief acquisition (see Sects. 6.3 and 6.4) and (first-order) decision-theoretic planning [36]. In belief acquisition, function definitions acquired from data can take the form of a decision list:

$$
\lambda x.if \ p_1 \ then \ v_1 \ else \ if \ p_2 \ then \ v_2 \ else \ if \ \ldots \ else \ if \ p_n \ then \ v_n \ else \ v_0. \tag{28}
$$

The implicit negations implied by nested *if_then_else* statements can sometimes be exploited to simplify the p_i's, leading to more comprehensible definitions. In symbolic dynamic programming algorithms [36,37], case statements having the form of (28) need to be multiplied and added together frequently in each value-iteration [38] step. Effective formula-simplification routines are needed there to avoid unnecessary blow-ups in space and time complexity. Indeed, the special-purpose reasoning engine for simplifying situation calculus formulas described in [36] bears close resemblance to Bach in several important aspects. The regression operation for situation calculus [39] is equivalent to performing Bach-style equational reasoning with successor state axioms. First-order versions of equations like (E) are also used in [36]. Further, an advanced first-order theorem prover [40] is coupled with the basic reasoning engine to achieve better formula simplication power.

Formula simplification/minimisation is a rich topic and we have only scratched the surface here. This subsection gives an indication of what can potentially be achieved using a combination of computation and proof.

6 Larger agent programming examples

We now present several more substantial examples that illustrate a number of aspects of agent construction using Bach. First, the examples in Sects. 6.1 and 6.2 show the expressive power of Bach for modelling belief bases. Sections 6.3 and 6.4 show how Bach can be used in symbolic machine learning. Section 6.5 illustrates a particular approach to agent construction that makes the state density (often called the 'belief state' [41]) of the agent a central component. This approach uses a transition model to capture the effect that actions

have on the state density and an observation model that, for each state, gives a density on the observations that could be made in that state. Section 6.6 shows how a policy selects actions on the basis of maximum expected utility.

6.1 Belief bases

Consider a TV agent that maintains a TV guide, that is, a database about television programs. There are various ways a TV guide can be represented. A standard way is to represent it as a relation. But this standard relational representation ignores a functional dependency in the data: each date, time and channel triple uniquely determines a program. Here we represent a TV guide as a function definition that correctly models this functional dependency:

$$tv\_guide : Occurrence \rightarrow Program,$$
where $Occurrence = Date \times Time \times Channel$
$$Program = Title \times Duration \times Genre \times Classification \times Synopsis.$$

Here is a typical definition for *tv_guide*.

$\boldsymbol{B}_t(tv\_guide =$
   $\lambda x.if \ (x = ((1, 1, 2008), (21, 30), ABC)) \ then \ (\text{``The Bill''}, 50, Drama, M, \text{``}\dots\text{''})$
      $else \ if \ (x = ((1, 1, 2008), (19, 00), ABC)) \ then \ (\text{``ABC News''}, 30, News, G, \text{``}\dots\text{''})$
      $else \ if \ (x = ((1, 1, 2008), (20, 30), TEN)) \ then \ (\text{``Numb3rs''}, 60, Crime, M, \text{``}\dots\text{''})$
      $else \ if \ (x = ((1, 1, 2008), (19, 30), WIN)) \ then \ (\text{``Seinfeld''}, 30, Sitcom, PG, \text{``}\dots\text{''})$
         $\vdots$
      $else \ (\text{``''}, 0, NA, NA, \text{``''})),$

where $\boldsymbol{B}_t$ is the belief modality of the TV agent, and the last entry ("", 0, *NA*, *NA*, "") is the default term of type *Program*.

   Listed below are some typical queries we can answer using the definition. All computations are done using $\boldsymbol{B}_t$. Answers to some of the more complex queries are computed using the technique explained earlier in Example 13.

1.   Find the program at occurrence $((1, 1, 2008), (20, 30), TEN)$.

   Query: $(tv\_guide \ ((1, 1, 2008), (20, 30), TEN))$.
   Answer: $(\text{``Numb3rs''}, 60, Crime, M, \text{``When} \dots\text{''})$.

2.   Find the time and channel "The Bill" is screened on 1 Jan 2008.

   Query: $\exists y.((y = (tv\_guide \ ((1, 1, 2008), t, c))) \ \wedge \ ((projTitle \ y) = \text{``The Bill''}))$.
   Answer: $(t = (21, 30)) \ \wedge \ (c = ABC)$.

3.   Find all *M*-rated programs in the database.

   Query: $\{ x \mid \exists y.((x = (tv\_guide \ y)) \ \wedge \ ((projClassification \ x) = M)) \}$.
   Answer: $\{ \ (\text{``The Bill''}, 50, Drama, M, \text{``}\dots\text{''}),$
            $(\text{``Numb3rs''}, 60, Crime, M, \text{``}\dots\text{''}), \ \dots \}$.

4. Find all current-affairs programs in the database.

    Query: { x | $\exists y.((x = (tv\_guide\ y)) \land (currentAffairs\ (projGenre\ x)))$ }.
    Answer: { ("*ABC News*", 30, *News*, *G*, "..."), ... }.

    Here *currentAffairs* : *Genre* → Ω is a predicate on genres defined elsewhere. This example shows how *tv_guide* can be used in conjunction with other functions in the same way relations can be joined to answer complex queries.

    The definition for *tv_guide* given above has a linear structure. This is clearly not the best way to represent a database. We note here that the same data can be captured in a better data structure such as a red-black tree and the same set of queries can still be answered using essentially the same basic underlying mechanism, albeit more efficiently.

6.2 Probabilistic belief bases

We look at probabilistic belief bases in this section. In particular, we will examine how Bayesian networks [42–44], a powerful class of probabilistic models in increasing use in agents [45], can be represented and reasoned with in Bach.

    It is well known [43,44] that the joint distribution given by a Bayesian network with K nodes is

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k \mid pa_k),\tag{29}$$

where $\mathbf{x} \equiv \{x_1, \ldots, x_K\}$ is the set of random variables associated with the nodes of the graph, and pa_k denotes the set of parents of x_k. Each factor $p(x_k \mid pa_k)$ in (29) is a conditional probability function that takes n arguments, where n is the cardinality of pa_k, and returns a density over the domain of x_k. Clearly, an expression like (29) can be written down directly in Bach. Probabilistic inference algorithms [46,47] can also be implemented in Bach using higher-order functions. To make the above concrete, we now look at a detailed example taken from [48,49].

    It is a genetic model of the inheritance of a single gene that determines a person's blood type. Each person has two copies of the chromosome containing this gene, one, the m-chromosome, inherited from the mother, and one, the p-chromosome, inherited from the father. Figure 14 [48] shows a Bayesian network modelling the inheritance of blood types within a particular family.

    We first show a straightforward way of encoding this Bayesian network in Bach, starting with the declaration of the following constants.

    A, *B*, *O* : *Chromosome*
    A, *B*, *AB*, *O* : *BloodType*.

The four unconditional nodes (nodes without parents) in Fig. 14 are all governed by a uniform distribution on constants of type *Chromosome*. Each conditional probability density (CPD) in the network (there is one CPD per node) is represented by a function. In Fig. 14, there are three separate CPDs which are reused in a number of places:
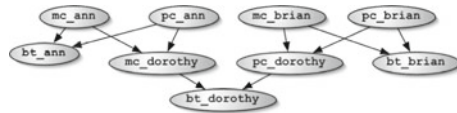
Fig. 14 A Bayesian network modelling the inheritance of blood types within a family. The variables mc_X and pc_X represent respectively the m and p-chromosomes of person X. The variable bt_X represents the blood type of person X

$mcd : Chromosome \rightarrow Chromosome \rightarrow (Density\ Chromosome)$

$(mcd\ A\ A\ z) = if\ (z = A)\ then\ 0.98\ else\ 0.01$

$(mcd\ B\ A\ z) = if\ (z = B)\ then\ 0.98\ else\ 0.01$

$\vdots$

$pcd : Chromosome \rightarrow Chromosome \rightarrow (Density\ Chromosome)$

$(pcd\ A\ A\ z) = if\ (z = A)\ then\ 0.98\ else\ 0.01$

$(pcd\ B\ A\ z) = if\ (z = A)\ then\ 0.98\ else\ 0.01$

$\vdots$

$btd : Chromosome \rightarrow Chromosome \rightarrow (Density\ BloodType)$

$(btd\ A\ A\ z) = if\ (z = A)\ then\ 0.97\ else\ 0.01$

$(btd\ B\ A\ z) = if\ (z = AB)\ then\ 0.97\ else\ 0.01$

$\vdots$

The three functions above are reproductions of the CPDs given in [48]. For example, the values of the three variables bt_ann, bt_brian, and $bt\_dorothy$ are governed by the function btd conditioned on the values of their parent variables.

Let mc_x, pc_x and bt_x denote the relevant variables in Fig. 14. Using a, b, d as shorthands for *Ann*, *Brian* and *Dorothy*, we can define the joint distribution as follows:

$joint : Density\ (Chromosome \times Chromosome \times Chromosome \times Chromosome \times$

$\qquad Chromosome \times Chromosome \times BloodType \times BloodType \times BloodType)$

$(joint\ (mc_a, pc_a, mc_b, pc_b, mc_d, pc_d, bt_a, bt_b, bt_d)) =$

$\qquad (1/3)^4 \times (mcd\ mc_a\ pc_a\ mc_d) \times (pcd\ mc_b\ pc_b\ pc_d) \times$

$\qquad (btd\ mc_a\ pc_a\ bt_a) \times (btd\ mc_b\ pc_b\ bt_b) \times (btd\ mc_d\ pc_d\ bt_d). \qquad (30)$

The simple encoding (30) does not take full advantage of the expressiveness of Bach. It is fine for small families but is tedious to write down for large multi-generation families. We now show how the network can be encoded more efficiently by exploiting two observations. The first is that although different families are associated with different Bayesian networks, each instance of these networks shares essentially the same basic structure. By separating the descriptions of family-dependent and -independent parts, we can compactly represent and reason with a large class of similar Bayesian networks. This is the strategy employed by systems like BLP [48] and one we adapt here. The second observation is that the random variables in Fig. 14 can be grouped under three classes: one parameterised by the function $mc : Person \rightarrow Chromosome$, one by the function $pc : Person \rightarrow Chromosome$, and one by the function $bt : Person \rightarrow BloodType$. Since there is a one-to-one correspondence between

an assignment of values to all the variables of the form mc_X and a definition for the function mc (and similarly for pc and bt), the joint given in (30) has an equivalent representation in the form of a joint density over the possible definitions for mc, pc, and bt. We are thus led to the following.

> *family* : *List Person*
>
> *family* = [*Ann, Brian, Dorothy*]
>
> *mother* : *Person* → *Person*
>
> (*mother x*) = if (*x* = *Dorothy*) then *Ann* else *Unknown*
>
> *father* : *Person* → *Person*
>
> (*father x*) = if (*x* = *Dorothy*) then *Brian* else *Unknown*
>
> *joint* : *Density* (*Person* → *Chromosome*) × (*Person* → *Chromosome*) ×
>
>   (*Person* → *BloodType*)
>
> (*joint* (f_{mc}, f_{pc}, f_{bt})) = (*expand family* (f_{mc}, f_{pc}, f_{bt}))
>
> *expand* : (*List Person*) → (*Person* → *Chromosome*) ×
>
>   (*Person* → *Chromosome*) × (*Person* → *BloodType*) → *Real*
>
> (*expand* [] (f_{mc}, f_{pc}, f_{bt})) = 1
>
> (*expand* ($\sharp$ *p t*) (f_{mc}, f_{pc}, f_{bt})) =
>
>   (*expand t* (f_{mc}, f_{pc}, f_{bt})) × (*btd* (f_{mc} *p*) (f_{pc} *p*) (f_{bt} *p*))
>
>   × if *mp* = *Unknown* then 1/3 else (*mcd* (f_{mc} *mp*) (f_{pc} *mp*) (f_{mc} *p*))
>
>   × if *fp* = *Unknown* then 1/3 else (*pcd* (f_{mc} *fp*) (f_{pc} *fp*) (f_{pc} *p*))
>
>   where *mp* = (*mother p*) ∧ *fp* = (*father p*)

Note that f_{mc}, f_{pc}, and f_{bt} above are function variables. The main work of constructing the complete Bayesian network from the arguments to *joint* is performed by the function *expand*. Also, by changing the definitions of *family* and *mother* and *father*, we get different Bayesian networks for different families.

Given the above, one can answer questions like "What is the probability that Dorothy has blood type A given that the m-chromosome of *Brian* is A?" by computing the expression

$$\frac{1}{K} \sum_{f_m \in C^P} \sum_{f_p \in C^P} \sum_{f_b \in B^P} (joint\ (f_m, f_p, f_b))(\mathbb{I}\ (f_b\ Dorothy) = A \wedge (f_m\ Brian) = A),$$

where K is a normalisation constant:

$$K = \sum_{f_m \in C^P} \sum_{f_p \in C^P} \sum_{f_b \in B^P} (joint\ (f_m, f_p, f_b))(\mathbb{I}\ (f_m\ Brian) = A),$$

$P = \{Ann, Brian, Dorothy\}$, $C = \{A, B, O\}$, $B = \{A, B, AB, O\}$, Y^X denotes the set of all functions from X to Y (the set Y^X can be incrementally enumerated using Bach's logic-programming facilities given only X and Y), and $\mathbb{I}$ is the indicator function that maps $\top$ to 1 and $\bot$ to 0. To compute such expressions efficiently, we need to exploit symmetries in function definitions and the factorisation of the joint to move products outside sums whenever possible. The following equations, which are easily represented in Bach, are typical examples of what we need.

$$\sum_{y \in s} t_1 \times \cdots \times t_n \;=\; t_i \times \sum_{y \in s} t_1 \times \cdots \times t_{i-1} \times t_{i+1} \times \cdots \times t_n$$

$$- - \text{ if } y \text{ is not free in } t_i$$

$$\sum_{f \in Y^X} \prod_{x \in X} (g \ (f \ x)) = \prod_{x \in X} \sum_{f \in Y^{\{x\}}} (g \ (f \ x)),$$

where $g : Y \rightarrow \mathbb{R}$ is an arbitrary function. The style of inference just described, where we deal with a group of random variables all at the same time, is called lifted probabilistic inference [5,50,51]. Such inference techniques present a compelling case for the need of higher-order functions in agent languages. (Similar arguments are made in [52] for the need for a second-order logic for robot planning.) For further details on probabilistic modelling and inference in Bach, we refer the reader to [8,9], where we also show how different probabilistic logic formalisms [4,53–59] can be supported in Bach.

6.3 Multi-agent systems in temporal domains

We next look at an application concerning belief acquisition in multi-agent systems. In particular, we discuss the TV recommender agent described in [60]. Suppose the current occupants of a household are Alice, Bob, and Cathy, and that the TV agent has acquired from training examples their television preferences in the form of current and past definitions for the function *likes* : *Program* $\rightarrow$ Ω for each of them, where *likes* is true for a program iff the person likes the program.

Let $\boldsymbol{B}_t$ be the belief modality for the TV agent, $\boldsymbol{B}_a$ the belief modality for Alice, $\boldsymbol{B}_b$ the belief modality for Bob, and $\boldsymbol{B}_c$ the belief modality for Cathy. Thus part of the TV agent's belief base has the following form:

$\boldsymbol{B}_t \boldsymbol{B}_a \, \forall x.((likes \ x) = \varphi_0)$

$\blacklozenge \boldsymbol{B}_t \boldsymbol{B}_a \, \forall x.((likes \ x) = \varphi_1)$

$\qquad \vdots$

$\blacklozenge^{n-1} \boldsymbol{B}_t \boldsymbol{B}_a \, \forall x.((likes \ x) = \varphi_{n-1})$

$\blacklozenge^{n} \boldsymbol{B}_t \forall x.(\blacklozenge \boldsymbol{B}_a (likes \ x) = \bot)$

$\boldsymbol{B}_t \boldsymbol{B}_b \, \forall x.((likes \ x) = \psi_0)$

$\blacklozenge \boldsymbol{B}_t \boldsymbol{B}_b \, \forall x.((likes \ x) = \psi_1)$

$\qquad \vdots$

$\blacklozenge^{k-1} \boldsymbol{B}_t \boldsymbol{B}_b \, \forall x.((likes \ x) = \psi_{k-1})$

$\blacklozenge^{k} \boldsymbol{B}_t \forall x.(\blacklozenge \boldsymbol{B}_b (likes \ x) = \bot)$

$\boldsymbol{B}_t \boldsymbol{B}_c \, \forall x.((likes \ x) = \xi_0)$

$\blacklozenge \boldsymbol{B}_t \boldsymbol{B}_c \, \forall x.((likes \ x) = \xi_1)$

$\qquad \vdots$

$\blacklozenge^{l-1} \boldsymbol{B}_t \boldsymbol{B}_c \, \forall x.((likes \ x) = \xi_{l-1})$

$\blacklozenge^{l} \boldsymbol{B}_t \forall x.(\blacklozenge \boldsymbol{B}_c (likes \ x) = \bot),$

for suitable φ_i, ψ_i, and ξ_i. The form these can take is explained in [60].

In the beginning, the belief base contains the formula

$$\boldsymbol{B}_t \forall x.(\blacklozenge \boldsymbol{B}_a(likes\ x) = \bot),$$

whose purpose is to prevent runaway computations into the infinite past for certain formulas of the form $\blacklozenge \varphi$. After n time steps, this formula has been transformed into

$$\bullet^n \boldsymbol{B}_t \forall x.(\blacklozenge \boldsymbol{B}_a(likes\ x) = \bot).$$

In general, at each time step, the beliefs about *likes* at the previous time steps each have another $\bullet$ placed at their front to push them one step further back into the past, and a new current belief about *likes* is acquired.

Based on these beliefs about the occupant preferences for TV programs, the task for the agent is to recommend programs that all three occupants would be interested in watching together. To achieve this, a (current) definition for the function

$$group\_likes : Program \rightarrow \Omega$$

needs to be acquired. The informal meaning of *group_likes* is that it is true for a program iff the occupants collectively like the program. (This may involve a degree of compromise by some of the occupants.) Training examples for this function can come in the form of individual examples and/or rules. Here are two examples:

$$\boldsymbol{B}_t \forall x.((x = (\text{``ABC news''}, 30, News, G, \text{``}\ldots\text{''})) \longrightarrow (group\_likes\ x))$$
$$\boldsymbol{B}_t \forall x.(((projGenre\ x) = Sports) \longrightarrow (group\_likes\ x)).$$

The following is a typical definition for *group_likes* acquired from training examples.

$$\boldsymbol{B}_t \forall x.\ ((group\_likes\ x) =$$
$$\quad if\ (\boldsymbol{B}_a likes\ x) \wedge (\boldsymbol{B}_b likes\ x) \wedge (\boldsymbol{B}_c likes\ x)\ then\ \top$$
$$\quad else\ if\ (\blacklozenge \boldsymbol{B}_a likes\ x) \wedge (\boldsymbol{B}_b likes\ x) \wedge (\blacklozenge \boldsymbol{B}_c likes\ x)\ then\ \top$$
$$\quad \vdots$$
$$\quad else\ \bot).$$

The algorithm used to acquire the definition is a generalisation of Rivest's decision-list learning algorithm [61]. We shall not be concerned with the actual algorithm here. Instead we will look at the kind of computational tasks that must be solved in support of the algorithm. The most important of these involve simplifying terms of the form

$$x = (\text{``ABC news''}, 30, News, G, \text{``}\ldots\text{''}) \wedge (\boldsymbol{B}_a likes\ x) \wedge (\boldsymbol{B}_b likes\ x) \wedge (\boldsymbol{B}_c likes\ x)$$

and

$$((projGenre\ x) = Sports) \wedge (\blacklozenge \boldsymbol{B}_a likes\ x) \wedge (\boldsymbol{B}_b likes\ x) \wedge (\blacklozenge \boldsymbol{B}_c likes\ x)$$

in the context of $\boldsymbol{B}_t$, using the previously acquired definitions of *likes*, the standard equality theory, and global assumptions like

$$\blacklozenge \varphi = \varphi \vee \bullet \blacklozenge \varphi$$
$$\bullet \boldsymbol{B}_i \varphi \longrightarrow \boldsymbol{B}_i \bullet \varphi.$$

Many of the computation facilities illustrated throughout this paper are needed in tight integration to solve the above computation problems.

6.4 Incremental belief revision

A TV recommender agent like that described in Sect. 6.3 needs to track the changing preferences of its users over a lifetime. Since preferences tend only to change slowly over time, we want an incremental belief revision scheme that can reuse past beliefs when appropriate. This section outlines how such a scheme can be realised in Bach.

We will start with an abstract formulation. Consider the problem of tracking a function $f : \sigma \to \tau$ that changes slowly over time. We have access to the previous acquired definitions

$$\bullet \, \boldsymbol{B} \, (f = \lambda x.\varphi_1)$$

$$\vdots$$

$$\bullet^{n-1} \, \boldsymbol{B} \, (f = \lambda x.\varphi_{n-1})$$
$$\bullet^{n} \, \blacksquare \, \boldsymbol{B} \, (f = \lambda x.\varphi_n).$$

in the belief base. ($\boldsymbol{B}$ is the belief modality of the relevant agent.) A new training set arrives and now a new definition for f needs to be acquired. How do we proceed?

A symbolic machine learning algorithm will search in a hypothesis space to find a new definition that fits the training set. Assume the new definition for f and each $\lambda x.\varphi_i$ already in the belief base take the form of a decision list [61]:

$$\lambda x.if \ (p_1 \ x) \ then \ v_1 \ else \ if \ (p_2 \ x) \ then \ v_2 \ \cdots \ else \ if \ (p_n \ x) \ then \ v_n \ else \ v_0, \qquad (31)$$

where each p_i belongs to a predicate space H and each v_i belongs to a label space L. The standard decision-list learning algorithm [61] can be used to learn such function definitions. Obviously, in computing the current definition for f, we would like to reuse those parts of the previous definitions that are still valid in the light of new evidence. One way to achieve that is to add to H predicates that capture different ways the old definitions can be changed, or perturbed, in small ways. We now show how the higher-order and modal facilities of Bach make this a relatively easy operation.

We start by defining the following function. (The function is defined informally here; a formal definition can be given using the technique of programming with abstractions described earlier.)

$$covered : Int \to Int \to (a \to b) \to (a \to \Omega)$$
$$(covered \ i \ j \ \lambda x.if \ (p_1 \ x) \ then \ v_1 \ else$$
$$\qquad\qquad if \ (p_2 \ x) \ then \ v_2 \ \cdots \ else \ if \ (p_n \ x) \ then \ v_n \ else \ v_0)$$
$$= \ if \ (i = 1) \ then \ \lambda x.((p_1 \ x) \vee (p_2 \ x) \vee \cdots \vee (p_j \ x))$$
$$else \ \lambda x.(\neg(p_1 \ x) \wedge \neg(p_2 \ x) \wedge \cdots \wedge \neg(p_{i-1} \ x) \wedge ((p_i \ x) \vee (p_{i+1} \ x) \vee \cdots \vee (p_j \ x))).$$

Thus, given a decision list f having the form of (31) and an individual x, the term $((covered \ i \ j \ f) \ x)$ evaluates to true iff x is covered by one of the predicates between p_i and p_j in f inclusive. Our desired new predicate space is obtained by adding to H the following set of predicates:

$$\{(covered \ j \ k \ \bullet^i f) \ : \ \text{for suitable values of } i, j \text{ and } k\}.$$

We also need to add $\{(\bullet^i f \ x)\}$ to the label space L. We now show that the space of functions so defined contains most of the ways we might want to modify an existing decision list. This new hypothesis space, in combination with Rivest's decision-list learning algorithm, gives us our desired incremental belief revision algorithm.

1. Recalling an old definition from m steps ago can be realised using

$$B(f = \lambda x.(\bullet^m f\ x).$$

2. The operation of adding a condition *if $(r\ x)$ then v* in between the k-th and $(k + 1)$-th nodes in a previous definition $\bullet^m f$ can be realised by the definition

$$\boldsymbol{B}\ (f = \lambda x.if\ ((covered\ 1\ k\ \bullet^m f)\ x)\ then\ (\bullet^m f\ x)\ else\ if\ (r\ x)\ then\ v\ else\ (\bullet^m f\ x).$$

3. We can piece together parts from definitions obtained at different times to form the current definition. For instance, we can have

$$\boldsymbol{B}\ (f = \quad \lambda x.if\ ((covered\ 2\ 8\ \bullet^2 f)\ x)\ then\ (\bullet^2 f\ x)$$
$$else\ if\ ((covered\ 6\ 9\ \bullet^4 f)\ x)\ then\ (\bullet^4 f\ x)\ else\ v_0),$$

where v_0 is some default value.

We end with a remark on comprehensibility. After several revision steps, the current definition may contain sub-decision-lists of older definitions that themselves are defined in terms of yet older definitions. Thus the exact meaning of the current definition may not be readily interpretable. If this is a problem, then one can simply unfold every definition before inserting it into the belief base.

More details about the incremental belief revision algorithm can be found in [26].

6.5 Bayesian tracking

Consider an agent situated in some environment that can receive percepts from the environment and can apply actions that generally have a non-deterministic effect on the environment. The primary task of the agent is to do the 'right thing', that is, choose the appropriate action for each state it finds itself in, where 'appropriate' usually means maximising its expected performance measure.

Suppose that *State* is the type of states of the world, *Action* is the type of actions, and *Observation* is the type of observations that the agent can make with its sensors. Then *Density State* is the type of densities of states and *Density Observation* is the type of densities of observations. There are five functions on these types that an agent must have available in order to choose actions. These are

    transition : *Action* → *State* → (*Density State*)

    observe : *State* → (*Density Observation*)

    observationUpdate : *Observation* → (*Density State*) → (*Density State*)

    actionUpdate : *Action* → (*Density State*) → (*Density State*)

    policy : (*Density State*) → *Action*

Given an action and a state, the function *transition* returns a state density which gives the distribution on the states the agent could end up in as a result of applying the action to the current state. The function *observe* provides the observation model, which gives the distribution on the observations the agent can perceive in any given state.

The next function *observationUpdate* provides the update of the state density as a result of the agent perceiving a particular observation. Essentially, this update is an application of Bayes rule. The function *actionUpdate* provides the update of the state density as a result of the agent applying some action. This update is a simple computation using the transition

function and the (current) state density. The definitions of the two functions can be easily derived from the rules of probability theory:

$$(observationUpdate\ o\ d) = (normalise\ \lambda s.((d\ s) \times (observe\ s\ o))). \tag{32}$$

$$(actionUpdate\ a\ d) = \lambda z. \sum_{y} (d\ y) \times (transition\ a\ y\ z). \tag{33}$$

The function *policy*, the most important of the functions that the agent needs, gives the action that is appropriate for any particular state density. We will look at that in Sect. 6.6. In this section, we will first look at how an agent can track its belief state using (32) and (33). We previously examined some simple techniques like Kalman filters [62] in [9]. In this paper, we will consider the following simplified version of Texas Hold'em poker. We have a deck of cards consisting of the following:

cards : {*Card*}
cards = {(♠, 1), (♠, 2), (♠, 3), (♠, 4), (♠, 5),
        (♣, 1), (♣, 2), (♣, 3), (♣, 4), (♣, 5)}.

Each game involves two players. At the beginning, each player is dealt a private card. Each player must then make a decision whether to *Play* or *Fold*. Subsequently four community cards are revealed one at a time, each followed by another round of betting. Each *Play* action incurs a cost of $10. A *Fold* action ends the current game with the folding player losing all the money bet so far. If both players play on till the end of the game, the winner is the one with the best combination of two cards from the private and community cards.

Let us now model the situation from the perspective of an agent playing the game. Each state of the game is captured by the private cards of the two players, the list of already revealed community cards, a flag indicating whether the opponent has folded, and a flag indicating whether we have reached the end of the game.

$$State = Card \times Card \times (List\ Card) \times \Omega \times \Omega.$$

Here, the first card belongs to the agent and the second to the opponent. The opponent's card is for the most part not observable to the agent.

The agent can perform two actions:

Play, *Fold* : *Action*.

Further, it can observe the latest new community card to be revealed, whether the opponent has folded, and the opponent's card at the end of a game. We are thus led to declare the following data constructors:

OpFold : *Observation*
NewCard : *Card* → *Observation*
OpCard : *Card* → *Observation*.

The way the state changes after each of the agent's actions is captured by the following state-transition model.

$\boldsymbol{B}_a((\text{transition Fold } (c, o, l, \bot, \bot)) = \lambda s.\text{if } s = (c, o, l, \bot, \top) \text{ then } 1 \text{ else } 0)$

$\boldsymbol{B}_a((\text{transition Play } (c, o, l, \bot, \bot)) =$

 if (length l) < 4 then

   $\lambda s.\text{if } \exists x.((\text{cards } x) \wedge (x \neq c) \wedge (x \neq o) \wedge \neg(\text{in } x \ l) \wedge (s = (c, o, (x \, \sharp \, l), \bot, \bot)))$

    then $1/(10 - (2 + (\text{length } l))) \times (\text{opAction } o \ l \ Play)$

    else if $s = (c, o, l, \top, \top)$ *then* $(\text{opAction } o \ l \ Fold)$

    else 0

 else $\lambda s.\text{if } s = (c, o, l, \bot, \top)$ *then* $(\text{opAction } o \ l \ Play)$

    else if $s = (c, o, l, \top, \top)$ *then* $(\text{opAction } o \ l \ Fold)$

    else $0)$

Here $\boldsymbol{B}_a$ is the belief modality of the agent. We assume the agent makes the first move in each betting round. The *Fold* case is straightforward. Two scenarios can follow in the *Play* case. The opponent can choose to play on. This happens with probability (*opAction o l Play*) defined below. From there, another community card is drawn randomly from the deck, unless we have reached the end of the current game. Alternatively, the opponent can fold, which leads to the end of the current game.

The definition of *opAction* is not usually just a straightforward combinatorial calculation because of the phenomenon of bluffing in poker. (Predictability can be brutally exploited.) Instead, the definition is usually acquired from data using machine learning techniques. This is where opponent modelling comes in. The following is a particularly simple example of an acquired definition.

$\text{opAction} : Card \rightarrow (\text{List } Card) \rightarrow (\text{Density Action})$

$\boldsymbol{B}_a((\text{opAction } o \ l \ Play) = \text{if } \exists x.((\text{in } x \ l) \wedge (\text{pair } o \ x)) \text{ then } 0.9$

         else if $(\text{proj}_2 \ o) \geq 3$ *then* 0.6

         else $0.5)$

$\boldsymbol{B}_a((\text{opAction } o \ l \ Fold) = 1 - (\text{opAction } o \ l \ Play))$

The opponent model says that the opponent will play on with 0.9 probability if he can form a pair from his private card and one of the community cards. He will also play with 0.6 probability if his private card is higher than 3. In general, the opponent will base its action on a combination of an estimate of what the agent's private card is, its winning probability, and its belief of a bluffing success among other things.

The agent is equipped with the following observation model to find out what happens after each of its actions. The observations are non-probabilistic but they can be made so to accommodate potential sensor errors.

$\boldsymbol{B}_a((\text{observe } (c, o, l, \top, \top)) = \lambda x.\text{if } x = OpFold \text{ then } 1 \text{ else } 0)$

$\boldsymbol{B}_a((\text{observe } (c, o, (y \, \sharp \, l), \bot, \bot)) = \lambda x.\text{if } x = (NewCard \ y) \text{ then } 1 \text{ else } 0)$

$\boldsymbol{B}_a((\text{observe } (c, o, l, \bot, \top)) = \lambda x.\text{if } x = (OpCard \ o) \text{ then } 1 \text{ else } 0)$

To see how tracking works, suppose the agent is dealt the private card (♠, 1) and starts with the prior that every possible initial state is equally probable. The agent would end up with the

following posterior after making three *Play* actions and observing first (*NewCard* ($\clubsuit$, 3)), then (*NewCard* ($\spadesuit$, 4)) and (*NewCard* ($\clubsuit$, 1)):

$\lambda s. if\ s = ((\spadesuit, 1), (\spadesuit, 2), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.0838$

$\quad else\ if\ s = ((\spadesuit, 1), (\spadesuit, 3), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.3257$

$\quad else\ if\ s = ((\spadesuit, 1), (\spadesuit, 5), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.1448$

$\quad else\ if\ s = ((\spadesuit, 1), (\clubsuit, 2), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.0838$

$\quad else\ if\ s = ((\spadesuit, 1), (\clubsuit, 5), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.1448$

$\quad else\ if\ s = ((\spadesuit, 1), (\clubsuit, 4), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp)\ then\ 0.2172$

$\quad else\ 0.$ \hfill (34)

It is worth noting that only computations of rank 0 are needed for this example to work. Equations like (E) are what we need to turn intensional descriptions like that given for *transition* into their extensional forms for processing by the summation function.

The state space in the simplified Texas Hold'em poker is small enough for exact Bayesian tracking to work. In most practical real-time applications, the size of the state space is such that an exact calculation of (32)–(33) is impossible. Such problems call for the need of approximate probabilistic inference techniques like particle filtering [63,64]. We will come back to this issue in Sect. 7.

6.6 Search and control

Consider again the simplified Texas Hold'em poker of Sect. 6.5. We have dealt with the problem of tracking. We now look at control, that is, action selection. To differentiate between good and bad actions, we need some measure of how valuable it is to be in a certain state. This is provided by a reward function. The following is a suitable one for our simplified Texas Hold'em.

reward : *State* → *Real*

$(reward\ (c, o, l, \top, x)) = ((length\ l) - 1) \times 10$

$(reward\ (c, o, l, \perp, x)) = if\ x = \perp\ then\ 0$

$\qquad\qquad\qquad\qquad else\ if\ (bestComb\ (c, l)) > (bestComb\ (o, l))\ then\ 30$

$\qquad\qquad\qquad\qquad else\ -30$

The function *bestComb* returns the best combination of cards that can be formed from a private card and the list of community cards. Here, the agent is rewarded if the opponent folded. The agent is also rewarded/penalised at the end of the game depending on the outcome. All other states have zero reward.

The utility of a state density *d* can be defined by adding together the expected reward with respect to *d* and the average utility of future state densities as follows:

utility : (*Density State*) → *Real*

$(utility\ d) =$

$(\mathbb{E}\ d\ reward)\ +$
  $if\ (endOfGame\ d)\ then\ 0$
  $else\ (max\ (costOfFolding\ d)$
      $\sum_{o}(obsProb\ (actionUpdate\ Play\ d)\ o)\ \times$
          $(utility\ (observationUpdate\ o\ (actionUpdate\ Play\ d)))),$

where $\mathbb{E}$ calculates the expected value of a function with respect to a density:

$\mathbb{E} : (Density\ a) \rightarrow (a \rightarrow Real) \rightarrow Real$
$(\mathbb{E}\ \lambda x.0\ f) = 0$
$(\mathbb{E}\ \lambda x.if\ x = u\ then\ v\ else\ \mathbf{w}\ f) = v \times (f\ u) + (\mathbb{E}\ \lambda x.\mathbf{w}\ f)$

and the probability of making a certain observation given a state density is given by

$obsProb : (Density\ State) \rightarrow Observation \rightarrow Real$
$(obsProb\ d\ o) = (\mathbb{E}\ d\ \lambda s.(observe\ s\ o)).$

Given the above, the action an agent that maximises expected reward should take in any given state density is given by

$policy : (Density\ State) \rightarrow Action$
$(policy\ d) = if\ (costOfFolding\ d)$
      $< \sum_{o}(obsProb\ (actionUpdate\ Play\ d)\ o)\ \times$
          $(utility\ (observationUpdate\ o\ (actionUpdate\ Play\ d)))$
      $then\ Play\ else\ Fold.$

For example, the function *policy* applied to (34) evaluates to *Play*. This is because the value of folding now is $-\$20$, whereas that of playing on is $-\$9.51$. Playing on has a higher expected value because there is a ~ 0.25 probability that the opponent may fold in response.

As in the previous section, only computations of rank 0 are needed for this example.

7 Discussion

This section contains a discussion of the advantages of the higher-order logic setting, a comparison of Bach with similar programming languages, some comments on our approach to agent architectures, and some remarks about current and future work on Bach.

7.1 Higher-order logic

Bach is set in the context of higher-order logic. Since most other declarative agent programming languages employ first-order logic, we now make some comments on the two settings. An outline of the technical details supporting these remarks is given in [23]. In summary, our view is that for many purposes, including programming agents, the higher-order setting is superior to the first-order setting. To explain our view, we now examine the two settings from several points of view.

The first aspect is that of expressive power. One way to think about higher-order logic, also known as simple type theory [22], is that it is a formalisation of everyday informal mathematics. Mathematical concepts are easy to express directly in higher-order logic because, amongst other things, the logic allows quantification over predicates and functions. This is illustrated in the agent programming context by the heavy and often essential use of higher-order functions in examples given throughout this paper; other good examples in more general settings are given in [23]. In contrast, first-order logic only allows one to model many mathematical concepts indirectly and requires the introduction of (semantically complicated) set theory to give a satisfactory foundation for mathematics. The great expressive power of higher-order logic partly explains its widespread use in several subfields of computer science; in functional programming, where a program can be understood as a higher-order equational theory; in formal methods, where the logic is used to give specifications of programs and prove properties about them; in theoretical computer science, where various kinds of semantics are typically higher order; and elsewhere.

However, even accepting the superior expressive power of higher-order logic, a common criticism is that it is computationally less attractive than first-order logic. This criticism is usually fuelled by observations such as the fact that higher-order unification is undecidable [31] and the logic does not have a sound and complete proof system. Carefully formulated, these criticisms are correct, but they do not present a balanced view of the situation. For that, we need to say something about the semantics of higher-order logic.

In the semantics, each (closed) type α is interpreted by a (non-empty) set D_α. The crucial aspect of the semantics in this discussion is the meaning given to function types. In the standard semantics, for a function type $\alpha \rightarrow \beta$, the set $D_{\alpha \rightarrow \beta}$ is *all* functions from D_α to D_β. The models given by this semantics are called standard models. Gödel showed in 1931 that, with the standard semantics, higher-order logic does not have a sound and complete proof system. Also (higher-order versions of) the compactness theorem and the Löwenhein-Skolem theorem do not hold.

The characteristic of the standard semantics that leads to these undesirable properties is that there are comparatively few models. A way to get a completeness result is to expand the class of models. This was famously done by Henkin in [65]. The key idea is to expand the class of models by allowing $\alpha \rightarrow \beta$ to denote a *subset* of the set of all functions from D_α to D_β, not necessarily all functions. The class of models given by this definition are called general models. Each standard model is a general model, but the converse is not true. With general models, Henkin was able to prove that there is a sound and complete proof procedure for higher-order logic. Also, using the Henkin semantics, the compactness theorem and the Löwenheim-Skolem theorem hold. At this point, Lindström's (first) theorem [66] can be applied to show that higher-order logic with the Henkin semantics is essentially just a variant of first-order logic. In spite of this result, something important has been gained: instead of being forced to express certain concepts awkwardly in first-order logic, the greater expressive power of higher-order logic can be exploited.

With regard to the undecidability of higher-order unification, note first that we cannot avoid dealing with undecidability even in the first-order case, since the validity problem of first-order logic is undecidable. In any case, one can do a lot in higher-order logic without ever having to resort to (higher-order) unification. This should be evident from the execution models of widely used functional languages like Haskell and ML, all of which are highly efficient and effective. The main equational reasoning component of Bach, which can be viewed as an extension of Haskell, is also a useful subset of higher-order logic that is both expressive and tractable. It uses linear-time (one-way) matching of terms instead of the difficult (two-way) unification of terms for pattern matching. The general strategy adopted here

is of course no different from the common approach of restricting first-order logic in various ways to achieve tractability in inference.

7.2 Comparison with similar programming languages

7.2.1 Multi-agent programming languages

Multi-agent programming systems are described extensively in [67]. A useful survey paper of current agent programming languages is [68]. Most of the logic-based systems in this literature are based on Prolog, extended in different ways to capture important aspects of agents. AgentSpeak(L) [69], Jason [70], and 3APL [71] are all based on extensions of Prolog that capture BDI concepts. In these systems, agent beliefs are represented as Prolog programs and plans, which are context-sensitive clauses with trigger events as heads and actions/subgoals as bodies, are used to update beliefs. MINERVA [72], also based on BDI agent concepts, uses multi-dimensional dynamic logic programming [73] and a knowledge and behaviour update language to specify agents and their behaviour. The distinguishing feature of MINERVA is that agent knowledge is represented by non-monotonic theories and updates are non-monotonic. IMPACT [74] is based on a language that extends Prolog with deontic modalities and also temporal and probabilistic reasoning [54,75]. It provides a framework for building agents on top of heterogeneous sources of knowledge. Another agent programming language that employs explicit modalities in programs is METATEM [76]. It is based on the direct execution of (first-order) modal logic statements, a process that involves the construction of models using a forward-chaining algorithm. Go! [77] is a programming language for multi-agent systems that, like Bach, supports both functional and logic programming idioms.

7.2.2 Prolog, Haskell, and Escher

This paper can be seen as a proposal for an alternative foundation for multi-agent programming systems based on modal probabilistic functional logic programming; thus the most relevant comparison with existing work is between Bach and Prolog itself, rather than the various agent languages based on Prolog. We also discuss why we think Bach is a better candidate for agent applications than its precursors Haskell and Escher [14].

It will be helpful to first understand the relationship between Escher and Haskell, and that between Escher and Prolog. Escher is an extension of Haskell. The difference between Escher and Haskell comes down to the following two points.

1. Haskell allows pattern matching only on data constructors. Escher extends this by also allowing pattern matching on function symbols and lambda abstractions. Examples of equations that Haskell won't accept include (E) and several others in the standard equality theory. This means Haskell cannot perform the kind of logic-programming style computations typified by Example 14 and used extensively elsewhere in the paper (Sects. 4.2, 5.3, 6.1, and 6.5).
2. Escher allows reduction of terms inside lambda abstractions, an operation not permitted in Haskell. This mechanism allows Escher to handle sets (and similar data types) in a natural and intensional way. Thus Haskell cannot perform (in a direct way) the kind of set-processing computations illustrated in Example 13 and Sect. 6.1.

The extra expressiveness afforded by Escher comes with a price, however. Some common optimisation techniques developed for efficient compilation of Haskell code (see [78] for a survey) cannot be used in the implementation of Escher.

We next explore the relationship between Escher and Prolog. The general relationship between Prolog and standard functional programming languages is well understood and will not be explored further here. Instead, we will concentrate on logic-programming facilities in Escher. Perhaps surprisingly, there is actually a significant overlap between Escher and Prolog. In fact, any pure Prolog program can be mechanically translated into Escher via Clark's completion [79]. For example, the definition of *append* given in Example 14 is essentially the completion of the following Prolog definition:

> *append* ([], *L*, *L*).
> *append* ([*X*|*L1*], *L2*, [*X*|*L3*]) $\longleftarrow$ *append* (*L1*, *L2*, *L3*).

Procedurally, there is also a difference between Escher and Prolog in that Prolog computes alternative answers one at a time via backtracking whereas Escher returns all alternative answers in a disjunction (a set). This is also illustrated in Example 14.

We can now compare Bach and Escher. Computations of rank 0 (Sect. 3.1) extend the execution model of Escher in several ways which we now examine. We start by looking at the definition of an Escher statement. An Escher program is a theory in which each statement is a term of the form $h = b$, where h has the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function symbol f. In contrast, an input equation in Bach is a term of the form

$$\Box_{j_1} \cdots \Box_{j_r} \forall (u = v),$$

where $\Box_{j_1} \cdots \Box_{j_r}$ is a sequence of modalities which may be empty, and u and v are arbitrary terms in the logic, possibly with modalities in them. There are thus two main differences between Escher and the rank 0 computation component of Bach:

1. The restriction on the form of the left hand side of an Escher statement is dropped in Bach. Equation I3, which we have seen serves an important role in supporting 'reverse'-direction computations in Example 13 and Sect. 6.1, is an example of an equation available in Bach but not in Escher. This extra flexibility in Bach comes at a small price in the form of a slightly more computationally expensive pattern-matching algorithm.
2. Modalities are only supported in Bach; Escher cannot perform the kind of computations illustrated in Examples 15 and 16, and others using modalities in this paper.

We have concentrated on the basic equational-reasoning component of Bach so far. There is also a significant difference in theorem-proving capabilities between Escher and Bach. Theorem-proving support in Escher is provided through the Σ and Π rules in the standard equality theory. Although sufficient for a range of common tasks, this is fundamentally a limited set. In contrast, Bach has a general-purpose theorem prover as a subsystem and the interaction between computation and proof makes possible some interesting computational tasks as shown in Sect. 5. We have argued in this paper that all the extensions of Escher available in Bach are needed in various aspects of agent programming.

7.2.3 Modal programming languages

Modal computation has been studied for 20 years or so, mostly in the logic programming community in the context of epistemic or temporal logic programming languages. Useful surveys of this work are in [80,81]. A recent paper showing the current state of the art of

modal logic programming languages is [82]. What is common between these works and this paper is the emphasis on epistemic and temporal modalities. What is different is that almost all are based on Prolog and are, therefore, first order, and it seems they usually either provide epistemic modalities or temporal modalities, but seldom both. Bach extends typical modal higher-order theorem proving systems, such as those in [10,27], largely in that it also has an equational reasoning component.

In the past, the motivations for employing modal higher-order logic have mostly been of a philosophical or linguistic nature, and outside these areas there have been very few works. For a brief historical account of these motivations, the reader is referred to the excellent handbook chapter [17]. A recent account of modal higher-order logic motivated by philosophical considerations is given in [10]. An earlier work motivated by mainly linguistic considerations is [83].

Modal logic has also been studied in the functional programming community in the context of type systems. In particular, modal (propositional) logics have been used to formulate sophisticated type systems that capture complex properties of environments in which programs are executed. Useful introductions to this line of work include [84,85]. Bach differs from these works in that modalities appear directly inside the language, not in a (meta-level) type system.

7.2.4 Higher-order/functional programming languages

The traditional foundation for functional programming languages has been the λ-calculus, rather than a higher-order *logic*. However, it is possible to regard functional programs as equational theories in a higher-order logic and this also provides a useful semantics. Bach extends the core execution mechanisms of existing functional languages in that it is modal, it admits logic programming idioms through programming with abstractions, and it also contains a theorem-proving system.

In the 1980s, higher-order programming in the logic programming community was introduced through the language λProlog [86]. The logical foundations of λProlog are provided by almost exactly the same logic as that underlying Bach (without modalities). However, a different sublogic is used for λProlog programs than the equational theories proposed here. In λProlog, program statements are higher-order hereditary Harrop formulas, a generalisation of the definite clauses used by Prolog. The language provides an elegant use of λ-terms as data structures, meta-programming facilities, universal quantification and implications in goals, amongst other features.

A long-term interest amongst researchers in declarative programming has been the goal of building integrated functional logic programming languages. Probably the best developed of these functional logic languages is the Curry language [87], which is the result of an international collaboration over the last 15 years or so. A survey of functional logic programming up to 1994 is in [88] and a more recent survey is given in [89].

7.3 Agent architectures

In Sects. 6.5 and 6.6, we have taken a decision-theoretic approach to agents instead of the more common BDI approach in the multi-agent programming literature. Generally, while adopting the BDI concept of belief in our agent architectures, which we show in [25] to also be able to capture notions of probabilistic beliefs in decision-theoretic settings, we prefer using rewards and/or utility to select actions rather than BDI concepts such as desires,

intentions or goals. For example, instead of plans, we employ policies (in the sense of Markov decision processes [90]) that maximise expected future reward, as in Sect. 6.6. This is because we have a preference towards designing agents that *learn* good policies, and there are many well-established learning algorithms in the decision-theoretic setting. Having said that, Bach is largely agnostic on this choice. Indeed, a learning system based on Escher/Bach, called Alkemy [15,91], has been used successfully in both decision-theoretic [92–94] and BDI architectures [95–97].

Note also that standard techniques for representing and reasoning about the effects of actions and changes in the world like the situation calculus [39] and variants [98] are well supported in Bach. Indeed, successor state axioms (SSAs) can be written down directly in Bach theories and the operation of regression can be straightforwardly understood as performing computation (equational reasoning) with SSAs. More sophisticated techniques that deal with incomplete states and other issues like the FLUX system [99] can also be implemented in Bach. Further, as we saw in Sect. 6.5, support for probabilistic computation also allows Bayesian tracking concepts from probabilistic artificial intelligence to be implemented directly in Bach.

7.4 Current and future work

This paper has proposed using modal probabilistic functional logic programming as the execution model of the Bach programming language. The formal definition of the execution model was given (and its theoretical underpinnings are available in [16]). Many examples were given to illustrate programming idioms provided by Bach that naturally arise in agent applications. Particular emphasis was placed on modal probabilistic computations since these are pervasive in agent applications. The language has been used in several projects including that of a tracking system for vehicles at intersections which was part of a National ICT Australia project aimed at improving the efficiency of road traffic in the Sydney metropolitan area. The tracking system was similar in structure to the example in Sect. 6.5 with state, a transition model, and an observation model, but considerably more complicated.

Generally speaking, in its current form, Bach is a useful prototyping tool for developing agent applications. However, much work still remains to be done on its implementation to make it a truly practical programming language. New algorithms and design approaches need to be developed to speed up performance-critical aspects of the language. The basic execution model also needs to be extended with constructs like modules, I/O, and concurrency; these constructs could be adopted directly from (concurrent) Haskell, for example. Also needed are facilities specific to agent applications, such as an agent communication language. In addition, further research is needed on control aspects of the interface between the computation and proof components.

Current work is partly concerned with the probabilistic computations. In particular, we are exploring the Bayesian tracking ideas for 'cognitive' agent applications, such as the one in Sect. 6.5, rather than the 'probabilistic robotics' domain for which Bayesian tracking was first introduced into artificial intelligence. Thus, in our case, the emphasis is on structured states and discrete distributions rather than simple feature vectors, continuous distributions, and Kalman filters typical of robotics applications. We are confident that this approach to state estimation will be successful in applications for which the agent maintains a sophisticated belief base (unlike typical robotics applications). The computational complexity of Bayesian tracking in non-trivial applications often necessitates the use of a sampling-based

approach, such as particle filters [63,64]. We are in the process of extending Bach with such facilities (along the line suggested in [100–102]).

8 Conclusion

This paper has introduced the execution model of Bach, a modal probabilistic functional logic programming language designed to facilitate the development of agent applications. We conclude by summarising the main contributions of the paper.

1. Multi-modal, higher-order logic is shown to be an expressive and practical formalism in which to model and reason about agent concepts.
2. An account of how modal and probabilistic computations can be supported in a functional logic programming language is presented.
3. Bach computations are shown to support a range of agent programming tasks, which include various forms of reasoning with modal and uncertain beliefs.
4. An execution model that tightly integrates equational reasoning and theorem proving is presented. This execution model, with suitable restrictions and control, is shown to make possible complex computational tasks that arise naturally in agent applications.

Appendix: Standard equality theory

Given the intended meanings of equality, the connectives and the quantifiers, it is natural that their definitions would normally be taken to be *global* assumptions in the theories of applications. All substitutions appearing in the following are assumed to be admissible.

Some of the equations listed below are schemas. A schema is intended to stand for the collection of formulas that can be obtained from the schema by replacing its syntactical variables with terms that satisfy the side conditions, if there are any. (Syntactical variables are typeset in bold in the following.) Thus a schema is a compact way of specifying a (possibly infinite) collection of formulas. When using a schema in a computation, a choice of terms to replace its syntactical variables is first made. The resultant formula is then handled as usual.

The first definition is that for =.

$$= : \; a \rightarrow a \rightarrow \Omega$$
$$(\boldsymbol{C}\; x_1 \ldots x_n = \boldsymbol{C}\; y_1 \ldots y_n) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \tag{D1}$$
$$\qquad -- \text{where } \boldsymbol{C} \text{ is a data constructor of arity } n.$$
$$(\boldsymbol{C}\; x_1 \ldots x_n = \boldsymbol{D}\; y_1 \ldots y_m) = \bot \tag{D2}$$
$$\qquad -- \text{where } \boldsymbol{C} \text{ is a data constructor of arity } n, \boldsymbol{D} \text{ is a data constructor of}$$
$$\qquad -- \text{arity } m \text{ and } \boldsymbol{C} \neq \boldsymbol{D}.$$

$$((x_1, \ldots, x_n) = (y_1, \ldots, y_n)) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n) \quad -- \text{ where } n = 2, 3, \ldots$$

$$(\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}) = (\text{less } \lambda x.\boldsymbol{u} \ \lambda y.\boldsymbol{v}) \wedge (\text{less } \lambda y.\boldsymbol{v} \ \lambda x.\boldsymbol{u})$$

$$\text{less } : \ (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow \Omega$$

$$\text{less } \lambda x.\boldsymbol{d} \ z = \top \quad -- \text{ where } \boldsymbol{d} \text{ is a default term.}$$

$$\text{less } \lambda x.(\text{if } \boldsymbol{u} \text{ then } v \text{ else } \boldsymbol{w}) \ z =$$
$$(\forall x.(\boldsymbol{u} \longrightarrow v = (z \ x))) \wedge (\text{less } (\text{remove } \lambda x.\boldsymbol{u} \ \lambda x.\boldsymbol{w}) \ z)$$

$$\text{remove } : \ (a \rightarrow \Omega) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b)$$

$$\text{remove } s \ \lambda x.\boldsymbol{d} = \lambda x.\boldsymbol{d} \quad -- \text{ where } \boldsymbol{d} \text{ is a default term.}$$

$$\text{remove } s \ \lambda x.(\text{if } \boldsymbol{u} \text{ then } v \text{ else } \boldsymbol{w}) =$$
$$\lambda x.(\text{if } \boldsymbol{u} \wedge \neg(s \ x) \text{ then } v \text{ else } ((\text{remove } s \ \lambda x.\boldsymbol{w}) \ x))$$

The first three equations above simply capture the intended meanings of data constructors and tuples. (Note that for (D1), the right hand side is $\top$ if $n = 0$.) The fourth equation is more subtle.[1] In formulations of higher-order logics, it is common for the axioms for equality to include the axiom of extensionality:

$$(f = g) = \forall x.((f \ x) = (g \ x)).$$

This axiom is not used in Bach because it is not computationally useful: there can be infinitely many values of x to consider in general. Instead, a special case of the axiom of extensionality is used. Its purpose is to provide a method of checking whether certain abstractions representing finite sets, finite multisets and similar data types are equal. The equation relies on the definitions of *less* and *remove*. The intended meaning of *less* is best given by an illustration. Consider the multisets m and n. Then (*less m n*) is true iff each item in the support of m is also in the support of n and has the same multiplicity there. For sets, *less* is simply the subset relation. If s is a set and m a multiset, then *remove s m* returns the multiset obtained from m by removing all the items from its support that are in s.

The following definitions are for the connectives $\wedge$, $\vee$, and $\neg$. Symmetric versions of some of the equations have been omitted for brevity.

$$\wedge \ : \ \Omega \rightarrow \Omega \rightarrow \Omega$$

$$\top \wedge x = x \tag{A1}$$

$$\bot \wedge x = \bot \tag{A2}$$

$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$$

$$(\text{if } u \text{ then } v \text{ else } w) \wedge t = (\text{if } u \wedge t \text{ then } v \text{ else } (w \wedge t))$$

$$\boldsymbol{u} \wedge (\exists x_1. \cdots \exists x_n.\boldsymbol{v}) = \exists x_1. \cdots \exists x_n.(\boldsymbol{u} \wedge \boldsymbol{v}) \tag{C1}$$
$$-- \text{ where } \boldsymbol{u} \text{ does not contain a free occurrence of any of the } x_i$$

---

[1] A note on scoping: In this paper, the body of an abstraction (or a quantifier) extends as little to the right as possible. For example, $\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}$ should be read as $(\lambda x.\boldsymbol{u}) = (\lambda y.\boldsymbol{v})$, not $\lambda x.(\boldsymbol{u} = \lambda y.\boldsymbol{v})$. The same rule applies for modalities.

$$u \wedge (x = t) \wedge v = u\{x/t\} \wedge (x = t) \wedge v\{x/t\} \qquad \text{(C2)}$$

$\quad$ – – where x is a variable free in u or v but not free in t, and t is not a variable.

$\vee \; : \; \Omega \to \Omega \to \Omega$

$$\top \vee x = \top \qquad \text{(O1)}$$

$$\bot \vee x = x \qquad \text{(O2)}$$

$(if\ u\ then\ \top\ else\ w) \vee t = (if\ u\ then\ \top\ else\ (w \vee t))$

$(if\ u\ then\ \bot\ else\ w) \vee t = (\neg\, u \wedge w) \vee t$

$\neg \; : \; \Omega \to \Omega$

$\neg\, \bot = \top$

$\neg\, \top = \bot$

$\neg\, (\neg\, x) = x$

$\neg\, (x \wedge y) = (\neg\, x) \vee (\neg\, y)$

$\neg\, (x \vee y) = (\neg\, x) \wedge (\neg\, y)$

$\neg\, (if\ u\ then\ v\ else\ w) = (if\ u\ then\ \neg\, v\ else\ \neg\, w)$

These definitions are straightforward, except perhaps for (C1) and (C2). The first of these allows the scope of existential quantifiers to be extended provided it does not result in free variable capture. The second allows the elimination of some occurrences of a free variable (x, in this case), thus simplifying an expression. A few words about the expression $u \wedge (x = t) \wedge v$ are necessary. The intended meaning of this expression is that it is a term such that $(x = t)$ is 'embedded conjunctively' inside it. More formally, a term t is embedded conjunctively in t and, if t is embedded conjunctively in r (or s), then t is embedded conjunctively in $r \wedge s$. So, for example, $(x = s)$ is embedded conjunctively in the term $((p \wedge q) \vee r) \wedge ((x = s) \wedge (t \vee u))$. The same remark applies to (E) and (U1) below.

Next come the definitions of Σ and Π. Recall that $\exists x.t$ stands for $(\Sigma \; \lambda x.t)$ and $\forall x.t$ stands for $(\Pi \; \lambda x.t)$.

$\Sigma \; : \; (a \to \Omega) \to \Omega$

$\exists x.\top = \top$

$\exists x.\bot = \bot$

$$\exists x_1. \cdots \exists x_n.(x \wedge (x_1 = u) \wedge y) = \exists x_2. \cdots \exists x_n.((x \wedge y)\{x_1/u\}) \qquad \text{(E)}$$

$\qquad$ – – where x_1 is not free in u.

$\exists x_1. \cdots \exists x_n.(u \vee v) = ((\exists x_1. \cdots \exists x_n.u) \vee (\exists x_1. \cdots \exists x_n.v))$

$\exists x_1. \cdots \exists x_n.(if\ u\ then\ \top\ else\ v) = (if\ \exists x_1. \cdots \exists x_n.u\ then\ \top\ else\ \exists x_1. \cdots \exists x_n.v)$

$\exists x_1. \cdots \exists x_n.(if\ u\ then\ \bot\ else\ v) = \exists x_1. \cdots \exists x_n.(\neg\, u \wedge v)$

$\Pi \; : \; (a \to \Omega) \to \Omega$

$\forall x_1. \cdots \forall x_n.(\bot \longrightarrow u) = \top$

$\forall x_1. \cdots \forall x_n.((x \wedge (x_1 = u) \wedge y) \longrightarrow v) =$

$\quad \forall x_2. \cdots \forall x_n.(((x \wedge y) \longrightarrow v)\{x_1/u\}) \qquad \text{(U1)}$

$\qquad$ – – where x_1 is not free in u.

$$\forall x_1. \cdots \forall x_n.((\boldsymbol{u} \vee \boldsymbol{v}) \longrightarrow \boldsymbol{t}) = \forall x_1. \cdots \forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t}) \wedge \forall x_1. \cdots \forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}) \quad \text{(U2)}$$

$$\forall x_1. \cdots \forall x_n.((\textit{if } \boldsymbol{u} \textit{ then } \top \textit{ else } \boldsymbol{v}) \longrightarrow \boldsymbol{t}) =$$
$$\forall x_1. \cdots \forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t}) \wedge \forall x_1. \cdots \forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t})$$

$$\forall x_1. \cdots \forall x_n.((\textit{if } \boldsymbol{u} \textit{ then } \bot \textit{ else } \boldsymbol{v}) \longrightarrow \boldsymbol{t}) = \forall x_1. \cdots \forall x_n.((\neg \boldsymbol{u} \wedge \boldsymbol{v}) \longrightarrow \boldsymbol{t})$$

These equations are essentially what are needed to support logic programming idioms in the functional setting.

The next four equations involve the *if _then_else* function.

$$(\textit{if } \top \textit{ then } u \textit{ else } v) = u \tag{I1}$$

$$(\textit{if } \bot \textit{ then } u \textit{ else } v) = v \tag{I2}$$

$$(\boldsymbol{w} \ (\textit{if } \boldsymbol{x} = \boldsymbol{t} \textit{ then } u \textit{ else } v)) = (\textit{if } \boldsymbol{x} = \boldsymbol{t} \textit{ then } (\boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{t}\} \ u) \textit{ else } (\boldsymbol{w} \ v)) \tag{I3}$$
$$- - \text{ where } \boldsymbol{x} \text{ is a variable.}$$

$$((\textit{if } \boldsymbol{x} = \boldsymbol{t} \textit{ then } u \textit{ else } v) \ \boldsymbol{w}) = (\textit{if } \boldsymbol{x} = \boldsymbol{t} \textit{ then } (u \ \boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{t}\}) \textit{ else } (v \ \boldsymbol{w})) \tag{I4}$$
$$- - \text{ where } \boldsymbol{x} \text{ is a variable.}$$

There is also the definition corresponding to β-reduction.

$$(\lambda x.\boldsymbol{u} \ t) = \boldsymbol{u}\{x/t\} \tag{B}$$

Also included in the standard equality theory is the schema

$$(\Box_i \mathbf{s} \ \mathbf{t}) = \Box_i (\mathbf{s} \ \mathbf{t}), \tag{M1}$$

where $\mathbf{s}$ is a syntactical variable ranging over terms of type $\alpha \rightarrow \beta$ and $\mathbf{t}$ is a syntactical variable ranging over *rigid* terms of type α. Another useful schema in the standard equality theory is

$$\Box_i \boldsymbol{t} = \boldsymbol{t}, \tag{M2}$$

where $\boldsymbol{t}$ is a syntactical variable ranging over *rigid* terms.

References

1. Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. (1995). *Reasoning about knowledge*. Cambridge: MIT Press.
2. Wooldridge, M. (2000). *Reasoning about rational agents*. Cambridge: MIT Press.
3. Gabbay, D. M., Kurucz, A., Wolter, F., & Zakharyaschev, M. (2003). *Many-dimensional modal logics: Theory and applications, Studies in logic and the foundations of mathematics* (Vol. 148). Amsterdam: Elsevier.
4. Halpern, J. Y. (1990). An analysis of first-order logics of probability. *Artificial Intelligence, 46*(3), 311–350.
5. Poole, D. (2003). First-order probabilistic inference. In *Proceedings of the 18th international joint conference on artificial intelligence* (pp. 985–991).
6. De Raedt, L., & Kersting, K. (2003). Probabilistic logic learning. *SIGKDD Explorations, 5*(1), 31–48.
7. Getoor, L. & Taskar, B. (Eds.). (2007). *Introduction to statistical relational learning*. Cambridge: MIT Press.
8. Ng, K. S., & Lloyd, J. W. (2009). Probabilistic reasoning in a classical logic. *Journal of Applied Logic, 7*(2), 218–238. doi:10.1016/j.jal.2007.11.008.
9. Ng, K. S., Lloyd, J. W., & Uther, W. T. B. (2008). Probabilistic modelling, inference and learning using logical theories. *Annals of Mathematics and Artificial Intelligence, 54*, 159–205. doi:10.1007/s10472-009-9136-7.
10. Fitting, M. (2002). *Types, tableaus, and Gödel's God*. Dordrecht: Kluwer Academic Publishers.

11. Peyton Jones, S. (2003). *Haskell 98 language and libraries: The revised report*. Cambridge: Cambridge University Press.
12. Lloyd, J. W. (1987). *Foundations of logic programming* (2nd ed.). New York: Springer.
13. Hill, P. M., & Lloyd, J. W. (1994). *The Gödel programming language*. Cambridge, MA: MIT Press.
14. Lloyd, J. W. (1999). Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming, 3*, 1–49.
15. Lloyd, J. W. (2003). *Logic for learning: Learning comprehensible theories from structured data*. New York: Springer.
16. Lloyd, J. W. (2007). Knowledge representation and reasoning in modal higher-order logic. Available at http://rsise.anu.edu.au/~jwl.
17. Muskens, R. (2006). Higher order modal logic. In P. Blackburn, J. van Benthem, & F. Wolter (Eds.), *Handbook of modal logic*. Amsterdam: Elsevier.
18. Andrews, P. B. (1986). *An introduction to mathematical logic and type theory: To truth through proof*. San Diego: Academic Press.
19. van Benthem, J., & Doets, K. (1983). Higher-order logic. In D. Gabbay & F. Guenther (Eds.), *Handbook of philosophical logic* (Vol. 1, pp. 275–330). Reidel.
20. Leivant, D. (1994). Higher-order logic. In D. Gabbay, C. Hogger, J. Robinson, & J. Siekmann (Eds.), *Handbook of logic in artificial intelligence and logic programming* (Vol. 2, pp. 230–321). Oxford: Oxford University Press.
21. Shapiro, S. (2001). Classical logic II—higher-order logic. In L. Goble (Ed.), *The Blackwell guide to philosophical logic* (pp. 33–54). Oxford: Blackwell.
22. Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic, 5*, 56–68.
23. Farmer, W. (2008). The seven virtues of simple type theory. *Journal of Applied Logic, 6*(3), 267–286.
24. Lloyd, J. W., & Ng, K. S. (2008). Reflections on agent beliefs. In M. Baldoni, T. C. Son, M. B. van Riemsdijk, & M. Winikoff (Eds.), *Declarative agent languages and technologies V, fifth international workshop, DALT 2007, LNAI 4897* (pp. 122–139). Springer, New York.
25. Lloyd, J. W., & Ng, K. S. (2008) Probabilistic and logical beliefs. In M. Dastani, J. Leite, A. El Fallah Seghrouchni, & P. Torroni (Eds.), *Languages, methodologies and development tools for multi-agent systems, international workshop, LADS 2007, LNAI 5118* (pp. 19–36). Springer, New York.
26. Lloyd, J. W., & Ng, K. S. (2007). Learning modal theories. In S. Muggleton, R. Otero, & A. Tamaddoni-Nezhad (Eds.), *Proceedings of the 16th international conference on inductive logic programming, LNAI 4455* (pp. 320–334).
27. Fitting, M., & Mendelsohn, R. L. (1998). *First-order modal logic*. Dordrecht: Kluwer Academic Publishers.
28. del Cerro, L. F., & Gasquet, O. (2002). A general framework for pattern-driven modal tableaux. *Logic Journal of the IGPL, 10*(1), 51–83.
29. Fitting, M. (1990). *First-order logic and automated theorem proving*. New York: Springer-Verlag.
30. Kohlhase, M. (1998). Higher-order automated theorem proving. In W. Bibel & P. H. Schmidt, *Automated deduction: A basis for applications. Volume I. Foundations: Calculi and methods*. Dordrecht: Kluwer Academic Publishers.
31. Dowek, G. (2001). Higher-order unification and matching. In *Handbook of automated reasoning* (pp. 1009–1062). Elsevier Science Publishers B.V.
32. Huet, G. P. (1975). A unification algorithm for typed λ-calculus. *Theoretical Computer Science, 1*, 27–57.
33. Giese, M. (2001). Incremental closure of free variable tableaux. In R. Goré, A. Leitsch, & T. Nipkow (Eds.), *Proceedings of international joint conference on automated reasoning, Siena, Italy, no. 2083 in LNCS* (pp. 545–560). Springer-Verlag, New York.
34. Miller, D. A. (1983). Proofs in higher-order logic. Ph.D. thesis, Mathematics Department, Carnegie-Mellon University.
35. Nipkow, T., Paulson, L. C., & Wenzel, M. (2005). *Isabelle/HOL: A proof assistant for higher-order logic*. No. 2283 in LNCS. Springer-Verlag, New York.
36. Sanner, S. (2008). First-order decision-theoretic planning in structured relational environments. Ph.D. thesis, University of Toronto.
37. Sanner, S., & Boutilier, C. (2009). Practical solution techniques for first-order MDPs. *Artificial Intelligence, 173*, 748–788. doi:10.1016/j.artint.2008.11.003.
38. Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Belmont: Athena Scientific.
39. Reiter, R. (2001). *Knowledge in action: Logical foundations for specifying and implementing dynamical systems*. Cambridge: MIT Press.
40. Riazanov, A., & Voronkov, A. (2002). The design and implementation of Vampire. *AI Communications, 15*(2), 91–110.

41. Russell, S. J., & Norvig, P. (2002). *Artificial intelligence: A modern approach* (2nd ed.). Upper Saddle River: Prentice-Hall.
42. Murphy, K. P. (2001). An introduction to graphical models. Technical report.
43. Jordan, M. I. (2004). Graphical models. *Statistical Science, 19*, 140–155.
44. Bishop, C. M. (2006). *Pattern recognition and machine learning*. New York: Springer.
45. Xiang, Y. (2002). *Probabilistic reasoning in multiagent systems: A graphical models approach*. Cambridge: Cambridge University Press.
46. Zhang, N. L., & Poole, D. (1994). A simple approach to Bayesian network computations. In *Proceedings of the 10th biennial Canadian artificial intelligence conference* (pp. 171–178).
47. Kschischang, F. R., Frey, B. J., & Loeliger, H. A. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory, 47*(2), 498–519.
48. Kersting, K., & De Raedt, L. (2007). Bayesian logic programming: Theory and tool. In L. Getoor, & B. Taskar (Eds.), *Introduction to statistical relational learning* (Chap. 10). MIT Press, Cambridge.
49. Friedman, N., Getoor, L., Koller, D., & Pfeffer, A. (1999). Learning probabilistic relational models. In *Proceedings of the 16th international joint conference on artificial intelligence* (pp. 1300–1307).
50. de Salvo Braz, R., Amir, E., & Roth, D. (2005). Lifted first-order probabilistic inference. In *Proceedings of the 19th international joint conference on artificial intelligence* (pp. 1319–1325).
51. de Salvo Braz, R., Amir, E., & Roth, D. (2007). Lifted first-order probabilistic inference. In L. Getoor & B. Taskar (Eds.), *Introduction to statistical relational learning* (Chap. 15). MIT Press, Cambridge.
52. Grosskreutz, H. (1999). Probabilistic, temporal projections in ConGolog. In *Proceedings of IJCAI'99 workshop on robot action planning*.
53. Fagin, R., & Halpern, J. Y. (1994). Reasoning about knowledge and probability. *Journal of the ACM, 41*(2), 340–367.
54. Ng, R. T., & Subrahmanian, V. S. (1992). Probabilistic logic programming. *Information and Computation, 101*(2), 150–201.
55. Lakshmanan, L. V. S., & Sadri, F. (1994). Modeling uncertainty in deductive databases. In D. Karagiannis (Ed.), *Proceedings of the international conference on database and expert systems applications, DEXA'94* (pp. 724–733).
56. Dekhtyar, A., & Subrahmanian, V. S. (2000). Hybrid probabilistic programs. *Journal of Logic Programming, 43*(3), 187–250.
57. Muggleton, S. (1996). Stochastic logic programs. In L. De Raedt (Ed.), *Advances in inductive logic programming* (pp. 254–264). Amsterdam: IOS Press.
58. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., & Kolobov, A. (2005). BLOG: Probabilistic models with unknown objects. In L. Kaelbling & A. Saffiotti (Eds.), *Proceedings of the 19th international joint conference on artificial intelligence* (pp. 1352–1359).
59. Ramsey, N., & Pfeffer, A. (2002). Stochastic lambda calculus and monads of probability distributions. In *Proceedings of 29th annual ACM symposium on principles of programming languages*. *SIGPLAN Notices, 37*(1), 154–165.
60. Cole, J. J., Gray, M., Lloyd, J. W., & Ng, K. S. (2005). Personalisation for user agents. In F. Dignum, et al. (Eds.), *Fourth international joint conference on autonomous agents and multi agent systems (AAMAS 05)* (pp. 603–610).
61. Rivest, R. L. (1987). Learning decision lists. *Machine Learning, 2*(3), 229–246.
62. Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME: Journal of Basic Engineering, 82(D)*, 35–45.
63. Doucet, A. (1998). On sequential simulation-based methods for Bayesian filtering. Technical report TR310. Department of Engineering, University of Cambridge.
64. Arulampalam, S., Maskell, S., Gordon, N., & Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing, 50*, 174–188.
65. Henkin, L. (1950). Completeness in the theory of types. *Journal of Symbolic Logic, 15*(2), 81–91.
66. Ebbinghaus, H., Flum, J., & Thomas, W. (1984). *Mathematical logic*. New York: Springer-Verlag.
67. Bordini, R. H., Dastani, M., Dix, J., & El Fallah Seghrouchni, A. (Eds.) (2005). *Multi-agent programming: languages, platforms and applications*. Springer, New York.
68. Bordini, R. H., Braubach, L., Dastani, M., El Fallah Seghrouchni, A., Gomez-Sanz, J. J., Leite, J., O'Hare, G., Pokahr, A., & Ricci, A. (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica, 30*, 33–44.
69. Rao, A. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde & J. Perram (Eds.), *Proceedings of the 7th workshop on modelling autonomous agents in a multi-agent world, LNAI 1038* (pp. 42–55). Springer-Verlag, New York.

70. Bordini, R., Hübner, J., & Vieira, R. (2005). Jason and the golden fleece of agent-oriented programing. In R. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Multi-agent programming: languages, platforms and applications* (Chap. 1, pp. 3–37). Springer, New York.

71. Hindriks, K., Boer, F., de Hock, W., & van der Meyer, J. J. C. (1999). Agent programming in 3APL. *Journal of Autonomous Agents and Multi-Agent Systems, 2*(4), 357–401.

72. Leite, J., Alferes, J., & Pereira, L. (2002). MINERVA—a dynamic logic programming agent architecture. In J. J. Meyer & M. Tambe (Eds.), *Intelligent agents VIII—agent theories, architectures, and languages, LNAI 2333* (pp. 141–157). Springer, New York.

73. Leite, J. A., Alferes, J. J., & Pereira, L. M. (2001). Multi-dimensional dynamic knowledge representation. In T. Eiter, W. Faber, & M. Truszczynski (Eds.), *Proceedings of the 6th international conference on logic programming and nonmonotonic reasoning, LNAI 2173* (pp. 365–378). Springer, New York.

74. Dix, J., & Zhang, Y. (2005). IMPACT: A multi-agent framework with declarative semantics. In R. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Multi-agent programming: languages, platforms and applications* (Chap. 3, pp. 69–94). Springer, New York.

75. Dix, J., Kraus, S., & Subrahmanian, V. (2002). Agents dealing with time and complexity. In M. Gini, T. Ishida, C. Castelfranchi, & W. Johson (Eds.), *Proceedings of the international joint conference on autonomous agents and multiagent systems* (pp. 912–919).

76. Fisher, M. (2005). METATEM: The story so far. In R. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni (Eds.), *Proceedings of the third international workshop on programming multiagent systems (ProMAS-03), LNAI 3862* (pp. 3–22). Springer, New York.

77. Clark, K. L., & McCabe, F. G. (2004). Go!—a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence, 41*(2–4), 171–206.

78. Peyton Jones, S. L. (1987). *The implementation of functional programming languages*. New York: Prentice-Hall.

79. Clark, K. (1978). Negation as failure. In H. Gallaire & J. Minker (Eds.), *Logic and databases* (pp. 293–322). New York: Plenum Press.

80. Orgun, M. A., & Ma, W. (1994). An overview of temporal and modal logic programming. In D. Gabbay & H. Ohlbach (Eds.), *Proceedings of the first international conference on temporal logics (ICTL'94), LNAI* (Vol. 827, pp. 445–479). Springer, New York.

81. Gergatsoulis, M. (2001). Temporal and modal logic programming languages. In A. Kent & J. Williams (Eds.), *Encyclopedia of microcomputers* (Vol. 27, pp. 393–408). Marcel Dekker, New York.

82. Nguyen, L. A. (2006). Multimodal logic programming. *Theoretical Computer Science, 360*, 247–288.

83. Gallin, D. (1998). *Intensional and higher-order modal logic*. Dordrecht: Kluwer Academic Publishers.

84. Nanevski, A. (2004). Functional programming with names and necessity. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

85. Fairtlough, M., Mendler, M., & Moggi, E. (2001). Special issue: Modalities in type theory. *Mathematical Structures in Computer Science, 11*, 507–509.

86. Nadathur, G., & Miller, D. (1998). Higher-order logic programming. In D. Gabbay, C. Hogger, & A. Robinson, *Handbook of logic in AI and logic programming, Vol. 5: Logic programming*. Oxford: Oxford University Press.

87. Hanus, M. (Ed.) Curry: An integrated functional logic language. http://www.informatik.uni-kiel.de/~curry.

88. Hanus, M. (1994). The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming, 19&20*, 583–628.

89. Hanus, M. (2007). Multi-paradigm declarative languages. In *Proceedings of the international conference on logic programming (ICLP 2007), LNCS 4670* (pp. 45–75). Springer, New York.

90. Puterman, M. (1994). *Markovian decision problems*. New York: Wiley.

91. Ng, K. S. (2005). Learning comprehensible theories from structured data. Ph.D. thesis, Computer Sciences Laboratory, The Australian National University.

92. Gretton, C., & Thiébaux, S. (2004) Exploiting first-order regression in inductive policy selection. In *Proceedings of the 20th conference on uncertainty in artificial intelligence*. Morgan Kaufmann, San Mateo.

93. Bridle, R., & McCreath, E. (2004). Improving the learning rate by inducing a transition model. In *Proceedings of the 3rd international joint conference on autonomous agents and multi agent systems* (pp. 1330–1331).

94. Cole, J. J., Lloyd, J. W., & Ng, K. S. (2003). Symbolic learning for adaptive agents. In *Annual partner conference*, Smart Internet Technology Cooperative Research Centre (pp. 139–148).

95. Phung, T., Winikoff, M., & Padgham, L. (2005). Learning within the BDI framework: An empirical analysis. In *Proceedings of the international conference on knowledge-based and intelligent information and engineering systems*.
96. Nguyen, A., & Wobcke, W. (2005). An agent-based approach to dialogue management in personal assistants. In *Proceedings of the international conference on intelligent user interfaces* (pp. 137–144).
97. Nguyen, A., & Wobcke, W. (2006). An adaptive plan-based dialogue agent: Integrating learning into a BDI architecture. In *Proceedings of the international joint conference on autonomous agents and multiagent systems* (pp. 785–788).
98. Lespérance, Y., Levesque, H., Lin, F., Marcu, D., Reiter, R., & Scherl, R. (1996). Foundations of a logical approach to agent programming. In M. Wooldridge, J. Müller, & M. Tambe (Eds.), *Intelligent agents II: Agent theories, architectures, and languages, LNAI 1037* (pp. 331–346). Springer, New York.
99. Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming, 5*(4–5), 533–565.
100. Goodman, N., Mansinghka, V. K., Roy, D., Bonawitz, K., & Tenenbaum, J. B. (2008). Church: A language for generative models. In D. A. McAllester & P. Myllymäki (Eds.), *Proceedings of the 24th conference in uncertainty in artificial intelligence* (pp. 220–229). AUAI Press, Corvallis.
101. Zettlemoyer, L. S., Pasula, H. M., & Kaelbling, L. P. (2007). Logical particle filtering. In *Proceedings of the Dagstuhl seminar on probabilistic, logical, and relational learning*.
102. Hajishirzi, H., & Amir, E. (2008). Sampling first order logical particles. In *Proceedings of UAI-08* (pp. 248–255).