

Declarative Programming for Agent Applications

J.W. Lloyd · K.S. Ng

Received: date / Accepted: date

Abstract This paper introduces the computational 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.

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 computational 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 computational 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 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.

J.W. Lloyd
College of Engineering and Computer Science, The Australian National University
E-mail: john.lloyd@anu.edu.au

K.S. Ng
National ICT Australia, The Australian National University
E-mail: keesiong.ng@nicta.com.au

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 for computing with modalities. 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 computational 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 equational 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.

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 program-

ming 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 Section 3. This is followed by the proof component in Section 4. The full reasoning system consisting of computation and proof combined is given in Section 5. Small instructive programming examples are sprinkled throughout these three sections to illustrate central concepts. Section 6 provides some larger applications. Section 7 contains a discussion of related and future work. We conclude in Section 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, \dots .

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 $\alpha_1, \dots, \alpha_k$ are types, then $T \alpha_1 \dots \alpha_k$ 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 $\alpha_1, \dots, \alpha_n$ are types, then $\alpha_1 \times \dots \times \alpha_n$ 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 \perp , 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 \perp 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 $\#_\sigma : \sigma \rightarrow List\ \sigma \rightarrow List\ \sigma$ and $\llbracket \sigma : List\ \sigma$ 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 \square_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 $\alpha_1 \times \dots \times \alpha_n$.
6. (Modal Term) If t is a term of type α and $i \in \{1, \dots, m\}$, then $\square_i t$ 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 $(\#_{Int}\ 2\ (\#_{Int}\ 3\ \llbracket_{Int}))$ of type $(List\ Int)$ represents a list with the integers 2 and 3 in it, obtained via a series of applications from the constants $\#_{Int}$, \llbracket_{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 \perp , 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 $\{\}$ (i.e., $\lambda x.\perp$).

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, \dots). 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 $\#\alpha$, since the types associated with these are now inferred from the context.

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

$$\lambda x.(if_then_else ((x = 2), \top, (if_then_else ((x = 3), \top, \perp)))).$$

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:

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

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

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

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 $\mathbf{K}_1, \mathbf{K}_2$, and \mathbf{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 $\mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3$. Modalities can also have a variety of temporal readings. We can introduce \Box_7 for ‘next’ (written as \circ), \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 $\mathbf{B}_i 42$ 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 \perp if the meaning of t in all accessible worlds is \perp , and, in the other cases, the meaning of $\Box_i t$ is conventionally defined to be \perp . This suggests an obvious extension to terms t that have rank 0 (that is, do not have type of the form $\alpha \rightarrow \beta$): 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. $\top, \perp, 1, 2, 3, \dots, \#$, \square) to be rigid. Also, all constants in the Haskell Prelude 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 $B_j \varphi$, with the intuitive meaning ‘agent j believes φ ’. Often φ is an equation. Other typical local assumptions have the form $B_j B_i \varphi$, 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 $B_{j_1} \dots B_{j_r} \varphi$, 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 $\bullet^2 B_j B_i \varphi$, 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 $f : \sigma \rightarrow Int$.

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

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 terms like $\lambda x.((f x) > 5)$, i.e. 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*.

$$\begin{aligned} \text{append} & : List\ a \times List\ a \times List\ a \rightarrow \Omega \\ (\text{append}\ (u, v, w)) & = ((u = [] \wedge v = w) \vee \\ & \quad \exists r. \exists x. \exists y. ((u = r \# x) \wedge (w = r \# y) \wedge (\text{append}\ (x, v, y)))) \end{aligned} \quad (3)$$

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 $(\text{append}\ ([1], [2], x))$ to $x = [1, 2]$, and a term like $(\text{append}\ (x, y, [1, 2]))$ 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 $g : Int \rightarrow Int$ at the current time and some recent times.

$$\forall x. ((g x) = \text{if } (\text{even } x) \text{ then } (\text{if } (x < 6) \text{ then } 21 \text{ else } (\bullet g x)) \text{ else } (\bullet^2 g x)) \quad (4)$$

$$\bullet \forall x. ((g x) = \text{if } (x > 0) \text{ then } (\bullet g x) \text{ else } 0) \quad (5)$$

$$\bullet^2 \forall x. ((g x) = 42) \quad (6)$$

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

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

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

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

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

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

gaussian : Real \rightarrow Real \rightarrow Density Real

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

mean : (Density *a*) \rightarrow Real

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

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

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

$$\blacklozenge x = (x \vee \bullet \blacklozenge x). \quad (16)$$

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

$$\blacklozenge((\text{mean } \text{prices}) < (\text{mean } \bullet \text{prices})).$$

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 Section 3.3.)

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. 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 a subterm is the sequence of indices of modalities whose scope is entered when going down to the subterm. 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* $\square_{j_1} \cdots \square_{j_r}$ *with respect to* \mathcal{T} *is a sequence* $\{t_i\}_{i=1}^n$ *of terms such that the following holds. For* $i = 1, \dots, n-1$, *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 $k_1 \dots k_{m_i}$,
2. (a) a formula $\square_{j_1} \cdots \square_{j_r} \square_{k_1} \cdots \square_{k_{m_i}} \forall (u_i = v_i)$ 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 have a normal form for all terms. We compute until there is no redex left. Theorem 1 shows that all the different terms we can obtain by choosing different redexes at each step are equal to one another. In practice, we can always get the answer in the form we want, either through carefully chosen input equations or a post-processing step. For example, in applications where computations result in normal terms [15] that represent individuals, we can always transform each answer into its equivalent and unique basic-term form.

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. (We will see some examples of overlapping patterns in Section 3.3.) 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 Figure 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  $\theta$ , 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  $\theta$ ;

```

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 Figure 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. $\lambda x.(f x (g y z)) \quad \lambda z.(f z (g A B))$
1. $\lambda w.(f w (g y z)) \quad \lambda w.(f w (g A B)) \{y/A\}$
2. $\lambda w.(f w (g A z)) \quad \lambda w.(f w (g A B)) \{z/B\}$
3. $\lambda w.(f w (g A B)) \quad \lambda w.(f w (g A B))$

(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.

$$\begin{array}{l} 0. (f \underset{\uparrow}{x} (g x)) (f \underset{\uparrow}{y} (g A)) \{x/y\} \\ 1. (f y (g \underset{\uparrow}{y})) (f y (g \underset{\uparrow}{A})) \end{array}$$

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.

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.

$$\begin{array}{l} 0. \lambda x.(f x \underset{\uparrow}{y} z) \lambda x.(f x \underset{\uparrow}{A} (g x)) \{y/A\} \\ 1. \lambda x.(f x A \underset{\uparrow}{z}) \lambda x.(f x A (g \underset{\uparrow}{x})) \end{array}$$

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 Standard Equality Theory

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 discussed next. 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 $=$.

$$\begin{array}{l} = : a \rightarrow a \rightarrow \Omega \\ (\mathbf{C} x_1 \dots x_n = \mathbf{C} y_1 \dots y_n) = (x_1 = y_1) \wedge \dots \wedge (x_n = y_n) \end{array} \quad (\text{D1})$$

-- where \mathbf{C} is a data constructor of arity n .

$$(\mathbf{C} x_1 \dots x_n = \mathbf{D} y_1 \dots y_m) = \perp \quad (\text{D2})$$

-- where \mathbf{C} is a data constructor of arity n , \mathbf{D} is a data constructor of

-- arity m and $\mathbf{C} \neq \mathbf{D}$.

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

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

$$\begin{aligned}
\text{less} & : (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow \Omega \\
\text{less } \lambda x. \mathbf{d} z & = \top \quad \text{-- where } \mathbf{d} \text{ is a default term.} \\
\text{less } \lambda x. (\text{if } \mathbf{u} \text{ then } v \text{ else } \mathbf{w}) z & = \\
& (\forall x. (\mathbf{u} \rightarrow v = (z x))) \wedge (\text{less } (\text{remove } \lambda x. \mathbf{u} \lambda x. \mathbf{w}) z) \\
\text{remove} & : (a \rightarrow \Omega) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \\
\text{remove } s \lambda x. \mathbf{d} & = \lambda x. \mathbf{d} \quad \text{-- where } \mathbf{d} \text{ is a default term.} \\
\text{remove } s \lambda x. (\text{if } \mathbf{u} \text{ then } v \text{ else } \mathbf{w}) & = \\
& \lambda x. (\text{if } \mathbf{u} \wedge \neg(s x) \text{ then } v \text{ else } ((\text{remove } s \lambda x. \mathbf{w}) x))
\end{aligned}$$

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 $(\text{less } m n)$ 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 $\text{remove } s m$ 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.

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

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

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

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

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

$$\mathbf{u} \wedge (\exists x_1. \dots \exists x_n. \mathbf{v}) = \exists x_1. \dots \exists x_n. (\mathbf{u} \wedge \mathbf{v}) \tag{C1}$$

-- where \mathbf{u} does not contain a free occurrence of any of the x_i

$$\mathbf{u} \wedge (\mathbf{x} = \mathbf{t}) \wedge \mathbf{v} = \mathbf{u}\{\mathbf{x}/\mathbf{t}\} \wedge (\mathbf{x} = \mathbf{t}) \wedge \mathbf{v}\{\mathbf{x}/\mathbf{t}\} \tag{C2}$$

-- where \mathbf{x} is a variable free in \mathbf{u} or \mathbf{v} but not free in \mathbf{t} , and \mathbf{t} is not a variable.

$$\vee : \Omega \rightarrow \Omega \rightarrow \Omega$$

$$\top \vee x = \top \tag{O1}$$

$$\perp \vee x = x \tag{O2}$$

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, $\lambda x. \mathbf{u} = \lambda y. \mathbf{v}$ should be read as $(\lambda x. \mathbf{u}) = (\lambda y. \mathbf{v})$, not $\lambda x. (\mathbf{u} = \lambda y. \mathbf{v})$. The same rule applies for modalities.

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

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

$$\neg : \Omega \rightarrow \Omega$$

$$\neg \perp = \top$$

$$\neg \top = \perp$$

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

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

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

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

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 $\mathbf{u} \wedge (\mathbf{x} = \mathbf{t}) \wedge \mathbf{v}$ are necessary. The intended meaning of this expression is that it is a term such that $(\mathbf{x} = \mathbf{t})$ 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 $(\Pi \lambda x.t)$.

$$\Sigma : (a \rightarrow \Omega) \rightarrow \Omega$$

$$\exists x.\top = \top$$

$$\exists x.\perp = \perp$$

$$\exists x_1. \dots \exists x_n. (\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y}) = \exists x_2. \dots \exists x_n. ((\mathbf{x} \wedge \mathbf{y})\{x_1/\mathbf{u}\}) \quad (\text{E})$$

-- where x_1 is not free in \mathbf{u} .

$$\exists x_1. \dots \exists x_n. (\mathbf{u} \vee \mathbf{v}) = ((\exists x_1. \dots \exists x_n. \mathbf{u}) \vee (\exists x_1. \dots \exists x_n. \mathbf{v}))$$

$$\exists x_1. \dots \exists x_n. (if\ \mathbf{u}\ then\ \top\ else\ \mathbf{v}) = (if\ \exists x_1. \dots \exists x_n. \mathbf{u}\ then\ \top\ else\ \exists x_1. \dots \exists x_n. \mathbf{v})$$

$$\exists x_1. \dots \exists x_n. (if\ \mathbf{u}\ then\ \perp\ else\ \mathbf{v}) = \exists x_1. \dots \exists x_n. (\neg \mathbf{u} \wedge \mathbf{v})$$

$$\Pi : (a \rightarrow \Omega) \rightarrow \Omega$$

$$\forall x_1. \dots \forall x_n. (\perp \longrightarrow \mathbf{u}) = \top$$

$$\forall x_1. \dots \forall x_n. ((\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y}) \longrightarrow \mathbf{v}) =$$

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

-- where x_1 is not free in \mathbf{u} .

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

$$\forall x_1. \dots \forall x_n. ((if\ \mathbf{u}\ then\ \top\ else\ \mathbf{v}) \longrightarrow \mathbf{t}) =$$

$$\quad \forall x_1. \dots \forall x_n. (\mathbf{u} \longrightarrow \mathbf{t}) \wedge \forall x_1. \dots \forall x_n. (\mathbf{v} \longrightarrow \mathbf{t})$$

$$\forall x_1. \dots \forall x_n. ((if\ \mathbf{u}\ then\ \perp\ else\ \mathbf{v}) \longrightarrow \mathbf{t}) = \forall x_1. \dots \forall x_n. ((\neg \mathbf{u} \wedge \mathbf{v}) \longrightarrow \mathbf{t})$$

As we shall see later, these equations are essentially what we need to support logic programming idioms in the functional setting.

The next four equations involve the *if_then_else* function.

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

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

$$(w (if \mathbf{x} = \mathbf{t} then u else v)) = (if \mathbf{x} = \mathbf{t} then (w\{x/t\} u) else (w v)) \tag{I3}$$

-- where \mathbf{x} is a variable.

$$((if \mathbf{x} = \mathbf{t} then u else v) w) = (if \mathbf{x} = \mathbf{t} then (u w\{x/t\}) else (v w)) \tag{I4}$$

-- where \mathbf{x} is a variable.

There is also the definition corresponding to β -reduction.

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

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

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

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

3.4 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 logic-programming style computations can be supported in the functional setting. This style of programming is 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 details in Section 3.5.

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 have been written in the relational style of logic programming. Figure 4 shows the computation of $(append (1 \# [], 2 \# [], x))$. 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 Figure 4, the following term is reached:

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

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

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

<u>(f B)</u>	[2]
<u>if B = A then 42 else if B = B then 3 else if B = C then 21 else 0</u>	[D2]
<u>if ⊥ then 42 else if B = B then 3 else if B = C then 21 else 0</u>	[I2]
<u>if B = B then 3 else if B = C then 21 else 0</u>	[D1]
<u>if ⊤ then 3 else if B = C then 21 else 0</u>	[I1]
3	

Fig. 2 Computation of $(f B)$. Redexes are underlined. The input equation used at each step is also given

$\lambda x. (> \underline{(f x)} 5)$	[2]
$\lambda x. (> \underline{(if x = A then 42 else if x = B then 3 else if x = C then 21 else 0)} 5)$	[I3]
$\lambda x. (\underline{(if x = A then (> 42) else (> (if x = B then 3 else if x = C then 21 else 0)))} 5)$	[I4]
$\lambda x. (if x = A then (> 42 5) else (> (if x = B then 3 else if x = C then 21 else 0) 5))$	
$\lambda x. (if x = A then ⊤ else (> \underline{(if x = B then 3 else if x = C then 21 else 0)} 5))$	[I3]
$\lambda x. (if x = A then ⊤ else \underline{((if x = B then (> 3) else (> (if x = C then 21 else 0)))} 5))$	[I4]
$\lambda x. (if x = A then ⊤ else if x = B then (> 3 5) else (> (if x = C then 21 else 0) 5))$	
$\lambda x. (if x = A then ⊤ else if x = B then ⊥ else (> \underline{(if x = C then 21 else 0)} 5))$	[I3]
$\lambda x. (if x = A then ⊤ else if x = B then ⊥ else \underline{((if x = C then (> 21) else (> 0))} 5))$	[I4]
$\lambda x. (if x = A then ⊤ else if x = B then ⊥ else if x = C then (> 21 5) else (> 0 5))$	
$\lambda x. (if x = A then ⊤ else if x = B then ⊥ else if x = C then ⊤ else \underline{(> 0 5)})$	
$\lambda x. (if x = A then ⊤ else if x = B then ⊥ else if x = C then ⊤ else ⊥)$	

Fig. 3 Computation of $\lambda x. ((f x) > 5)$

Similarly, one can eliminate x' to obtain

$$\exists y'. ((x = 1 \# y') \wedge (\text{append} (\[], 2 \# \[], y'))).$$

After some more computation, the answer $x = 1 \# 2 \# \[]$ 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, with a carefully chosen set of equations, can be extended 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 $(\text{append} (x, y, 1 \# 2 \# \[]))$ will be reduced, using essentially the same operations shown in Figure 4, to the answer

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

$\underline{\text{append}(1\#[], 2\#[], x)}$	[3]
$((1\#[] = []) \wedge (2\#[] = x)) \vee \exists r'. \exists x'. \exists y'. ((1\#[] = r'\#x') \wedge (x = r'\#y') \wedge \text{append}(x', 2\#[], y'))$	[D2]
$(\perp \wedge (2\#[] = x)) \vee \exists r'. \exists x'. \exists y'. ((1\#[] = r'\#x') \wedge (x = r'\#y') \wedge \text{append}(x', 2\#[], y'))$	[A2]
$\perp \vee \exists r'. \exists x'. \exists y'. ((1\#[] = r'\#x') \wedge (x = r'\#y') \wedge \text{append}(x', 2\#[], y'))$	[O2]
$\exists r'. \exists x'. \exists y'. ((1\#[] = r'\#x') \wedge (x = r'\#y') \wedge \text{append}(x', 2\#[], y'))$	[D1]
$\exists r'. \exists x'. \exists y'. ((1 = r') \wedge ([] = x') \wedge (x = r'\#y') \wedge \text{append}(x', 2\#[], y'))$	[E]
$\exists x'. \exists y'. (([] = x') \wedge (x = 1\#y') \wedge \text{append}(x', 2\#[], y'))$	[E]
$\exists y'. ((x = 1\#y') \wedge \text{append}([], 2\#[], y'))$	[3]
\vdots	
$\exists y'. ((x = 1\#y') \wedge (y' = 2\#[]))$	[E]
$x = 1\#2\#[]$	

Fig. 4 Computation of $(\text{append}(1\#[], 2\#[], x))$

$(g\ 12)$	[4]
$(\text{if } \underline{\text{even } 12} \text{ then } (\text{if } (< 12\ 6) \text{ then } 21 \text{ else } (\bullet g\ 12)) \text{ else } (\bullet^2 g\ 12))$	
\vdots	
$(\text{if } \top \text{ then } (\text{if } (< 12\ 6) \text{ then } 21 \text{ else } (\bullet g\ 12)) \text{ else } (\bullet^2 g\ 12))$	[I1]
$(\text{if } (< 12\ 6) \text{ then } 21 \text{ else } (\bullet g\ 12))$	
$(\text{if } \perp \text{ then } 21 \text{ else } (\bullet g\ 12))$	[I2]
$(\bullet g\ 12)$	[M1]
$\bullet(g\ 12)$	[5]
$\bullet(\text{if } (> 12\ 0) \text{ then } (\bullet g\ 12) \text{ else } 0)$	
$\bullet(\text{if } \top \text{ then } (\bullet g\ 12) \text{ else } 0)$	[I1]
$\bullet(\bullet g\ 12)$	[M1]
$\bullet^2(g\ 12)$	[6]
$\bullet^2 42$	[M2]
$\bullet 42$	[M2]
42	

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

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 $\bullet(g\ 12)$, the definition at the last time step kicks in; at the step $\bullet^2(g\ 12)$, 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 $\blacklozenge((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$. 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.

3.5 A Comparison with Escher, Haskell, and Prolog

Computations of rank 0 extend the computational model of Escher in several ways which we now examine. It will be helpful to first understand the relationship between Escher and Haskell, and that between Escher and Prolog.

Escher vs Haskell 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 shown in Example 14.
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 the kind of set-processing computations illustrated in Example 13.

The extra expressiveness afforded by Escher comes with a price, however. Some common optimisation techniques developed for efficient compilation of Haskell code (see [27] for a survey) cannot be used in the implementation of Escher.

Escher vs Prolog 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 [28]. For example, the definition of *append* given in Example 14 is essentially the completion of the following Prolog definition:

$$\begin{aligned} \text{append} ([], L, L). \\ \text{append} ([X|L1], L2, [X|L3]) \leftarrow \text{append} (L1, L2, L3). \end{aligned}$$

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 point is illustrated in Example 14.

$\diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[16]
$((\text{mean prices}) < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[7]
$((\text{mean } (\text{gaussian } 400 \ 20)) < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[13]
$(400 < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[8]
$(400 < (\text{mean } \bullet (\text{gaussian } 360 \ 25))) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[M2]
$(400 < (\text{mean } (\text{gaussian } 360 \ 25))) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[13]
$(400 < 360) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	
$\perp \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[O2]
$\bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[16]
$\bullet(((\text{mean prices}) < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[8]
$\bullet(((\text{mean } (\text{gaussian } 360 \ 25)) < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[13]
$\bullet((360 < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[9]
$\bullet((360 < (\text{mean } \bullet \lambda x. \text{if } x = 300 \text{ then } 0.7 \text{ else if } x = 310 \text{ then } 0.3 \text{ else } 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[M2]
$\bullet((360 < (\text{mean } \lambda x. \text{if } x = 300 \text{ then } 0.7 \text{ else if } x = 310 \text{ then } 0.3 \text{ else } 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[15]
$\bullet((360 < 300 \times 0.7 + (\text{mean } \lambda x. \text{if } x = 310 \text{ then } 0.3 \text{ else } 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	
$\bullet((360 < 210 + (\text{mean } \lambda x. \text{if } x = 310 \text{ then } 0.3 \text{ else } 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[15]
$\bullet((360 < 210 + 310 \times 0.3 + (\text{mean } \lambda x. 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	
$\bullet((360 < 210 + 93 + (\text{mean } \lambda x. 0)) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[14]
$\bullet((360 < 210 + 93 + 0) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	
$\bullet((360 < 303) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	
$\bullet(\perp \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[O2]
$\bullet \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$	[16]
$\bullet \bullet (((\text{mean prices}) < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[9]
\vdots	
$\bullet \bullet ((303 < (\text{mean } \bullet \text{prices})) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[10]
\vdots	
$\bullet \bullet ((303 < 330) \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	
$\bullet \bullet (\top \vee \bullet \diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices})))$	[O1]
$\bullet \bullet \top$	[M2]
$\bullet \top$	[M2]
\top	

Fig. 6 Computation of $\diamond((\text{mean prices}) < (\text{mean } \bullet \text{prices}))$

Bach vs Escher We can understand how rank 0 computations extend Escher by first 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 $f t_1 \dots t_n$, $n \geq 0$, for some function symbol f . In contrast, an input equation in Bach is a term of the form

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

where $\Box_{j_1} \dots \Box_{j_r}$ 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, 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.

We have concentrated on the basic equational-reasoning component of Bach so far. There is a significant difference in theorem-proving capabilities between Escher and Bach as well. 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 interesting computational tasks. The proof component is described next.

4 Proof

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

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 $1.\langle n_1, j_1 \rangle. \dots .\langle n_k, j_k \rangle$, where n_i is a positive integer and $j_i \in \{1, \dots, m\}$, for $i = 1, \dots, k$.

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 Figure 7 and Figure 8. Generally speaking, these rules are well known (see, for example, [29] and [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}_α 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, \dots, 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 $i = 1, \dots, n - 1$, there is
 - (a) a tableau rule R from Figure 7 or Figure 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 collection of tableaux rules given in Figure 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.

Our algorithm, which takes into consideration issues discussed in [30], [31] and [32], 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.

(Conjunctive rules) For any prefix σ ,		
$\frac{\sigma \ \varphi \wedge \psi}{\sigma \ \varphi}$	$\frac{\sigma \ \neg(\varphi \vee \psi)}{\sigma \ \neg\varphi}$	$\frac{\sigma \ \neg(\varphi \longrightarrow \psi)}{\sigma \ \neg\psi}$
$\sigma \ \psi$	$\sigma \ \neg\psi$	$\sigma \ \neg\psi$
(Disjunctive rules) For any prefix σ ,		
$\frac{\sigma \ \varphi \vee \psi}{\sigma \ \varphi \mid \sigma \ \psi}$	$\frac{\sigma \ \neg(\varphi \wedge \psi)}{\sigma \ \neg\varphi \mid \sigma \ \neg\psi}$	$\frac{\sigma \ \varphi \longrightarrow \psi}{\sigma \ \neg\varphi \mid \sigma \ \psi}$
(Double negation rule) For any prefix σ ,		
$\frac{\sigma \ \neg\neg\varphi}{\sigma \ \varphi}$		
(Possibility rules) If the prefix $\sigma.n_i$ is new to the branch, where $i \in \{1, \dots, m\}$,		
$\frac{\sigma \ \diamond_i \varphi}{\sigma.n_i \ \varphi}$	$\frac{\sigma \ \neg\Box_i \varphi}{\sigma.n_i \ \neg\varphi}$	
(Necessity rules) If the prefix $\sigma.n_i$ already occurs on the branch, where $i \in \{1, \dots, m\}$,		
$\frac{\sigma \ \Box_i \varphi}{\sigma.n_i \ \varphi}$	$\frac{\sigma \ \neg\diamond_i \varphi}{\sigma.n_i \ \neg\varphi}$	
(Existential rules) For any prefix σ , if x of type α and $w_\alpha \in \mathcal{W}_\alpha$ is new to the branch,		
$\frac{\sigma \ \exists x.\varphi}{\sigma \ \varphi\{x/w_\alpha\}}$	$\frac{\sigma \ \neg\forall x.\varphi}{\sigma \ \neg\varphi\{x/w_\alpha\}}$	
(Universal rules) For any prefix σ , if φ is a formula and $\{x/t\}$ is admissible w.r.t. φ ,		
$\frac{\sigma \ \forall x.\varphi}{\sigma \ \varphi\{x/t\}}$	$\frac{\sigma \ \neg\exists x.\varphi}{\sigma \ \neg\varphi\{x/t\}}$	
(Abstraction rules) For any prefix σ , if φ is a formula and $\{x/t\}$ is admissible w.r.t. φ ,		
$\frac{\sigma \ (\lambda x.\varphi \ t)}{\sigma \ \varphi\{x/t\}}$	$\frac{\sigma \ \neg(\lambda x.\varphi \ t)}{\sigma \ \neg\varphi\{x/t\}}$	
(Reflexivity rule) If t is a term and the prefix σ already occurs on the branch,		
$\frac{}{\sigma \ t = t}$		
(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 ,		
$\frac{\sigma \ s = t}{\sigma \ \varphi\{x/s\}}$		
$\sigma \ \varphi\{x/t\}$		

Fig. 7 Basic tableau 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 $T_{i+1} = T_i$.

<p>(Global assumption rule) If ψ is a global assumption and the prefix σ already occurs on the branch,</p> $\frac{}{\sigma \psi}$ <p>(Local assumption rule) If ψ is a local assumption,</p> $\frac{}{1 \psi}$ <p>(Derived rules for global implicational assumption) For any prefix σ, if $\varphi \longrightarrow \psi$ is a global assumption,</p> $\frac{\sigma \varphi}{\sigma \psi} \quad \frac{\sigma \neg\psi}{\sigma \neg\varphi}$ <p>(Derived rules for local implicational assumption) If $\varphi \longrightarrow \psi$ is a local assumption,</p> $\frac{1 \varphi}{1 \psi} \quad \frac{1 \neg\psi}{1 \neg\varphi}$
--

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

The three kinds of rules mentioned in the algorithm are distinguished in [30]. 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 Figure 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 [33]. We have opted for a simple syntactic match here for efficiency reasons. This works fine for our target applications but there are clear limitations. For example, we cannot at present prove Cantor's theorem in the style of [32], which requires Huet's algorithm [34]. This part of Bach can be redesigned as suggested in [32] should the need arise.

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

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

Fig. 9 Universal rules for free variable tableaux

Our algorithm for the closure rule is motivated by [35]. 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 (Figure 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

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

where x_1, \dots, x_n are the free variables that appear in the domains of the θ_i 's. The tableaux is closable if (17) is true. 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 $Free(\varphi)$ introduced by the universal rules, then there is a dependency between the new witness constant w_α to be introduced and the variables in $Free(\varphi)$. In particular, this means the witness constant w_α may not appear in any term used to instantiate any of the variables in $Free(\varphi)$. This problem is handled in first-order tableaux systems using Skolem functions. Naive Skolemisation is, however, unsound in higher-order logic [36]. A simple solution around this is to augment the tableaux with the maintenance of a partial function R (called a variable condition in [32]) 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 $x \in R(w)$.

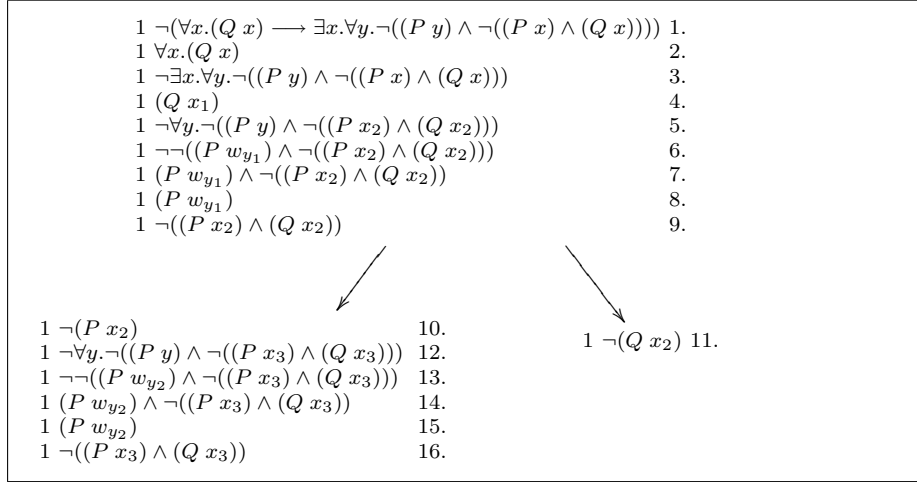


Fig. 10 Proof of $\forall x.(Q x) \longrightarrow \exists x.\forall y.\neg((P y) \wedge \neg((P x) \wedge (Q x)))$

4.3 Examples of Proof

We look at some examples of proof in this section. Example 17 is taken from an exercise in [31] 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 $P : \alpha \rightarrow \Omega$ and $Q : \alpha \rightarrow \Omega$ 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

$$\forall x.((x = A \vee x = B) \longrightarrow x \neq C). \quad (18)$$

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

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

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

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

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

$$\bullet B \varphi \longrightarrow B \bullet \varphi, \quad (20)$$

it can be shown that, for each $i \in \{1, \dots, 5\}$, $B \bullet^i \varphi_i$ is a theorem of the belief base. For example, Figure 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.

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 Section 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 [37] 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 $i = 1, \dots, n-1$, 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 $k_1 \dots k_{m_i}$,
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, \dots, 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 $i = 1, \dots, n-1$, there is either
 - (a) a tableau rule R from Figure 7 or Figure 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 1η ; or
 - (c) there is a result η of a computation of rank $k-1$ and a branch is extended with the prefixed formula 1η .
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 $j_1 \dots j_r$ 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 $f : Int \rightarrow \Omega$.

$$\mathbf{B} \forall x. ((f x) = (\text{even } x) \wedge (\bullet f x)) \quad (21)$$

$$\bullet \mathbf{B} \forall x. ((f x) = (\text{perfect } x)) \quad (22)$$

$$\bullet \mathbf{B} \varphi \longrightarrow \mathbf{B} \bullet \varphi \quad (23)$$

Suppose we want to compute the value of the term $(f \ 28)$ in the context of \mathbf{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

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

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 s in t is $j_1 \dots j_m$, 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,

$(f\ 28)$	[21]
$(\text{even } 28) \wedge (\bullet f\ 28)$	
\vdots	
$\top \wedge (\bullet f\ 28)$	
$(\bullet f\ 28)$	[M1]
$\bullet(f\ 28)$	[SM, 24]
1 $\neg \mathbf{B} \bullet \forall x. ((f\ x) = (\text{perfect } x))$	1.
1 $\bullet \mathbf{B} \forall x. ((f\ x) = (\text{perfect } x))$	2.
1 $\neg \bullet \mathbf{B} \forall x. ((f\ x) = (\text{perfect } x))$	3.
$\bullet(\text{perfect } 28)$	
\vdots	
$\bullet \top$	[M2]
\top	

Fig. 12 Computation of rank 1 using \mathbf{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 \mathbf{B} . At the step marked SM, the above algorithm kicks in and the conjecture

$$\mathbf{B} \bullet \forall x. ((f\ x) = (\text{perfect } x)) \quad (24)$$

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

$$((\text{proj}_1\ x) < 10) \wedge ((\text{proj}_2\ x) = 496) \wedge (\text{evenperfect } (\text{proj}_2\ x)) \wedge ((\text{proj}_1\ x) \neq (\text{proj}_2\ x)), \quad (25)$$

where x is a tuple and $\text{evenperfect} : \text{Int} \rightarrow \Omega$ is defined by

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

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

$$\forall x. (((\text{proj}_2\ x) = 496) \longrightarrow (\text{evenperfect } (\text{proj}_2\ x))) \text{ and} \quad (26)$$

$$\forall x. (((\text{proj}_1\ x) < 10) \wedge ((\text{proj}_2\ x) = 496) \longrightarrow (\text{proj}_1\ x) \neq (\text{proj}_2\ x)). \quad (27)$$

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	$\neg(evenperfect 496) = \perp$	6.
1	\perp	7.

Fig. 13 Proof of (26)

The proof of (26) is in Figure 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 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 $t_1 \wedge \dots \wedge t_n$, $n \geq 2$, that cannot be dealt with using computations of rank 0 alone. Let $t_{\bar{i}}$, for $i \in \{1, \dots, n\}$, denote $t_1 \wedge \dots \wedge t_{i-1} \wedge t_{i+1} \wedge \dots \wedge t_n$. We try to prove all possible formulas of the form $\forall(t_{\bar{i}} \longrightarrow t_i)$, $i \in \{1, \dots, n\}$. If we can prove $\forall(t_{\bar{i}} \longrightarrow t_i)$, then we are entitled to remove t_i from $t_1 \wedge \dots \wedge t_n$.

Computation problems of the kind just discussed arise naturally in belief acquisition and first-order decision-theoretic planning [38] applications. In belief acquisition, function definitions acquired from data can take the form of a decision list:

$$\lambda x. \text{if } p_1 \text{ then } v_1 \text{ else if } p_2 \text{ then } v_2 \text{ else if } \dots \text{ else if } p_n \text{ then } v_n \text{ else } v_0. \quad (28)$$

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 [38, 39], case statements having the form of (28) need to be multiplied and added together frequently in each value-iteration step. Effective formula-simplification routines are needed there to avoid unnecessary blow-ups in space and time complexity.

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 Sections 6.1 and 6.2 show the expressive power of Bach for modelling belief bases. Section 6.3 illustrates a particular approach to agent construction that makes the state density (often called the 'belief state' [40]) 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. This example illustrates probabilistic reasoning in Bach; this paper only discusses probabilistic reasoning briefly since it is covered in great detail in [8, 9]. Section 6.4

shows how a policy selects actions on the basis of maximum expected utility. Generally, while adopting the BDI concept of belief in our agent architectures, we prefer using rewards and/or utility to select actions rather than BDI concepts such as desires, intentions or goals.

6.1 Belief Bases

We first look at belief bases. Consider a TV agent that maintains a TV guide, that is, a database about television programs. There are different 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 .

$$\begin{aligned} \mathbf{B}_t(tv_guide = & \\ \lambda x. & \text{if } (x = ((1, 1, 2008), (21, 30), ABC)) \text{ then } ("The\ Bill", 50, Drama, M, "...") \\ & \text{else if } (x = ((1, 1, 2008), (19, 00), ABC)) \text{ then } ("ABC\ News", 30, News, G, "...") \\ & \text{else if } (x = ((1, 1, 2008), (20, 30), TEN)) \text{ then } ("Numb3rs", 60, Crime, M, "...") \\ & \text{else if } (x = ((1, 1, 2008), (19, 30), WIN)) \text{ then } ("Seinfeld", 30, Sitcom, PG, "...") \\ & \vdots \\ & \text{else } ("", 0, NA, NA, "") \end{aligned}$$

where \mathbf{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 \mathbf{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: $(\text{"Numb3rs"}, 60, Crime, M, \text{"When ..."})$.

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: $\{("The\ Bill", 50, Drama, M, "..."),$
 $(“Numb3rs”, 60, Crime, M, “...”), \dots\}$.

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

Query: $\{x \mid \exists y.((x = (tv_guide\ y)) \wedge (currentAffairs\ (projGenre\ x)))\}$.

Answer: $\{("ABC\ News", 30, News, G, "..."), \dots\}$.

Here $currentAffairs : Genre \rightarrow \Omega$ 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 like 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 Multi-Agent Systems in Temporal Domains

We next look at an application in belief acquisition in multi-agent systems. In particular, we discuss the TV recommender agent described in [41]. Suppose the current occupants of a household are Alice, Bob, and Cathy, and that our TV agent has acquired from training examples their television preferences in the form of current and past definitions for the function $likes : Program \rightarrow \Omega$ for each of them, where $likes$ is true for a program iff the person likes the program.

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

$$\begin{aligned}
& \mathbf{B}_t \mathbf{B}_a \forall x.((likes\ x) = \varphi_0) \\
& \bullet \mathbf{B}_t \mathbf{B}_a \forall x.((likes\ x) = \varphi_1) \\
& \quad \vdots \\
& \bullet^{n-1} \mathbf{B}_t \mathbf{B}_a \forall x.((likes\ x) = \varphi_{n-1}) \\
& \bullet^n \mathbf{B}_t \forall x.(\blacklozenge \mathbf{B}_a (likes\ x) = \perp) \\
& \mathbf{B}_t \mathbf{B}_b \forall x.((likes\ x) = \psi_0) \\
& \bullet \mathbf{B}_t \mathbf{B}_b \forall x.((likes\ x) = \psi_1) \\
& \quad \vdots \\
& \bullet^{k-1} \mathbf{B}_t \mathbf{B}_b \forall x.((likes\ x) = \psi_{k-1}) \\
& \bullet^k \mathbf{B}_t \forall x.(\blacklozenge \mathbf{B}_b (likes\ x) = \perp) \\
& \mathbf{B}_t \mathbf{B}_c \forall x.((likes\ x) = \xi_0)
\end{aligned}$$

$$\begin{aligned}
& \bullet \mathbf{B}_t \mathbf{B}_c \forall x. ((likes\ x) = \xi_1) \\
& \quad \vdots \\
& \bullet^{l-1} \mathbf{B}_t \mathbf{B}_c \forall x. ((likes\ x) = \xi_{l-1}) \\
& \bullet^l \mathbf{B}_t \forall x. (\blacklozenge \mathbf{B}_c (likes\ x) = \perp),
\end{aligned}$$

for suitable φ_i , ψ_i , and ξ_i . The form these can take is explained in [41].

In the beginning, the belief base contains the formula

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

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

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

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:

$$\begin{aligned}
& \mathbf{B}_t \forall x. ((x = (\text{"ABC news"}, 30, News, G, \dots)) \longrightarrow (group_likes\ x)) \\
& \mathbf{B}_t \forall x. (((projGenre\ x) = Sports) \longrightarrow (group_likes\ x)).
\end{aligned}$$

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

$$\begin{aligned}
& \mathbf{B}_t \forall x. ((group_likes\ x) = \\
& \quad \text{if } (\mathbf{B}_a \text{likes } x) \wedge (\mathbf{B}_b \text{likes } x) \wedge (\mathbf{B}_c \text{likes } x) \text{ then } \top \\
& \quad \text{else if } (\blacklozenge \mathbf{B}_a \text{likes } x) \wedge (\mathbf{B}_b \text{likes } x) \wedge (\blacklozenge \mathbf{B}_c \text{likes } x) \text{ then } \top \\
& \quad \quad \quad \vdots \\
& \quad \text{else } \perp).
\end{aligned}$$

The algorithm used to acquire the definition is a generalisation of Rivest's decision-list learning algorithm [42]. 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

$$x = (\text{"ABC news"}, 30, News, G, \dots) \wedge (\mathbf{B}_a \text{likes } x) \wedge (\mathbf{B}_b \text{likes } x) \wedge (\mathbf{B}_c \text{likes } x)$$

and

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

in the context of B_t , using the previously acquired definitions of *likes*, the standard equality theory, and global assumptions like

$$\begin{aligned}\spadesuit\varphi &= \varphi \vee \clubsuit\spadesuit\varphi \\ \clubsuit B_i\varphi &\longrightarrow B_i\clubsuit\varphi.\end{aligned}$$

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

6.3 Probabilistic Reasoning

For this subsection and the next, we will consider the following simplified version of Texas Hold'em poker. We have a deck of cards consisting of the following:

$$\begin{aligned}cards &: Card \rightarrow \Omega \\ cards &= \{(\spadesuit, 1), (\spadesuit, 2), (\spadesuit, 3), (\spadesuit, 4), (\spadesuit, 5), \\ &\quad (\clubsuit, 1), (\clubsuit, 2), (\clubsuit, 3), (\clubsuit, 4), (\clubsuit, 5)\}.\end{aligned}$$

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 two private player cards, 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.

$$\begin{aligned}OpFold &: Observation \\ NewCard &: Card \rightarrow Observation \\ OpCard &: Card \rightarrow Observation.\end{aligned}$$

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

$$\begin{aligned}
& \text{transModel} : \text{Action} \rightarrow \text{State} \rightarrow (\text{Density State}) \\
& \mathbf{B}_a((\text{transModel Fold } (c, o, l, \perp, \perp)) = \lambda s. \text{if } s = (c, o, l, \perp, \top) \text{ then } 1 \text{ else } 0) \\
& \mathbf{B}_a((\text{transModel Play } (c, o, l, \perp, \perp)) = \\
& \quad \text{if } (\text{length } l) < 4 \text{ then} \\
& \quad \quad \lambda s. \text{if } \exists x. ((\text{cards } x) \wedge (x \neq c) \wedge (x \neq o) \wedge \neg(\text{in } x \ l) \wedge (s = (c, o, (x \ \# \ l), \perp, \perp))) \\
& \quad \quad \quad \text{then } 1/(10 - (2 + (\text{length } l))) \times (\text{opAction o l Play}) \\
& \quad \quad \quad \text{else if } s = (c, o, l, \top, \top) \text{ then } (\text{opAction o l Fold}) \\
& \quad \quad \quad \text{else } 0 \\
& \quad \text{else } \lambda s. \text{if } s = (c, o, l, \perp, \top) \text{ then } (\text{opAction o l Play}) \\
& \quad \quad \text{else if } s = (c, o, l, \top, \top) \text{ then } (\text{opAction o l Fold}) \\
& \quad \quad \text{else } 0)
\end{aligned}$$

Here \mathbf{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*). 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.

$$\begin{aligned}
& \text{opAction} : \text{Card} \rightarrow (\text{List Card}) \rightarrow \text{Action} \rightarrow \text{Real} \\
& \mathbf{B}_a((\text{opAction o l Play}) = \text{if } \exists x. ((\text{in } x \ l) \wedge (\text{pair } o \ x)) \text{ then } 0.9 \\
& \quad \quad \text{else if } (\text{proj}_2 \ o) \geq 3 \text{ then } 0.6 \\
& \quad \quad \text{else } 0.5) \\
& \mathbf{B}_a((\text{opAction o l Fold}) = 1 - (\text{opAction o l Play}))
\end{aligned}$$

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.

$$\begin{aligned}
& \text{obsModel} : \text{State} \rightarrow (\text{Density Observation}) \\
& \mathbf{B}_a((\text{obsModel } (c, o, l, \top, \top)) = \lambda x. \text{if } x = \text{OpFold} \text{ then } 1 \text{ else } 0) \\
& \mathbf{B}_a((\text{obsModel } (c, o, (y \ \# \ l), \perp, \perp)) = \lambda x. \text{if } x = (\text{NewCard } y) \text{ then } 1 \text{ else } 0) \\
& \mathbf{B}_a((\text{obsModel } (c, o, l, \perp, \top)) = \lambda x. \text{if } x = (\text{OpCard } o) \text{ then } 1 \text{ else } 0)
\end{aligned}$$

Given the above transition and observation models, the posterior of the state can now be tracked with the following two standard update equations that can be easily derived from the axioms of probability theory.

transUpdate : *Action* \rightarrow *Density State* \rightarrow *Density State*

$$(\text{transUpdate } a \ d) = \lambda z. \sum_y (d \ y) \times (\text{transModel } a \ y \ z)$$

obsUpdate : *Observation* \rightarrow *Density State* \rightarrow *Density State*

$$(\text{obsUpdate } o \ d) = (\text{normalise } \lambda s. ((d \ s) \times (\text{obsModel } s \ o))).$$

Note that updating the state density using an observation is nothing more than applying Bayes rule.

For example, suppose the agent is dealt the private card ($\spadesuit, 1$) 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* ($\clubsuit, 1$)):

$$\begin{aligned} \lambda s. & \text{if } s = ((\spadesuit, 1), (\spadesuit, 2), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.0838 \\ & \text{else if } s = ((\spadesuit, 1), (\spadesuit, 3), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.3257 \\ & \text{else if } s = ((\spadesuit, 1), (\spadesuit, 5), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.1448 \\ & \text{else if } s = ((\spadesuit, 1), (\clubsuit, 2), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.0838 \\ & \text{else if } s = ((\spadesuit, 1), (\clubsuit, 5), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.1448 \\ & \text{else if } s = ((\spadesuit, 1), (\clubsuit, 4), [(\clubsuit, 1), (\spadesuit, 4), (\clubsuit, 3)], \perp, \perp) \text{ then } 0.2172 \\ & \text{else } 0. \end{aligned} \tag{29}$$

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 *transModel* into their extensional forms for processing by the summation function.

6.4 Search and Control

The previous subsection deals 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* \rightarrow *Real*

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

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

$$\text{else if } (\text{bestComb } (c, l)) > (\text{bestComb } (o, l)) \text{ then } 30$$

$$\text{else } -30$$

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:

$$\begin{aligned}
& \text{utility} : (\text{Density State}) \rightarrow \text{Real} \\
& (\text{utility } d) = \\
& \quad (\mathbb{E} \text{ } d \text{ reward}) + \\
& \quad \text{if } (\text{endOfGame } d) \text{ then } 0 \\
& \quad \text{else } (\max (\text{costOfFolding } d) \\
& \quad \quad \sum_o (\text{obsProb } (\text{transUpdate Play } d) o) \times \\
& \quad \quad \quad (\text{utility } (\text{obsUpdate } o (\text{transUpdate Play } d))))),
\end{aligned}$$

where

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

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

$$\begin{aligned}
& \text{obsProb} : (\text{Density } a) \rightarrow \text{Observation} \rightarrow \text{Real} \\
& (\text{obsProb } d o) = (\mathbb{E} d \lambda s.(\text{obsModel } s o)).
\end{aligned}$$

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

$$\begin{aligned}
& \text{policy} : (\text{Density State}) \rightarrow \text{Action} \\
& (\text{policy } d) = \text{if } (\text{costOfFolding } d) \\
& \quad < \sum_o (\text{obsProb } (\text{transUpdate Play } d) o) \times \\
& \quad \quad (\text{utility } (\text{obsUpdate } o (\text{transUpdate Play } d))) \\
& \quad \text{then Play else Fold.}
\end{aligned}$$

For example, the function *policy* applied to (29) 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 related work and indicates ongoing and future work for Bach.

7.1 Related Work

On Multi-Agent Programming Languages Multi-agent programming systems are described extensively in [43]. A useful survey paper of current agent programming languages is [44]. 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) [45], Jason [46], and 3APL [47] 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 [48], also based on BDI agent concepts, uses multi-dimensional dynamic logic programming [49] 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 [50] is based on a language that extends Prolog with deontic modalities and also temporal and probabilistic reasoning [51, 52]. 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 [53]. 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! [54] is a programming language for multi-agent systems that, like Bach, supports both functional and logic programming idioms.

The present paper can be seen as a proposal for an alternative foundation for multi-agent programming systems based on modal functional logic programming. There are several important differences between existing BDI-based agent programming languages and Bach. The computational models of Prolog and Bach were previously contrasted in Section 3.5. Agent beliefs are typically represented as clausal theories in (modal) first-order logic in existing BDI languages. In Bach, they are represented as equational theories in modal higher-order logic. Furthermore, instead of plans, we generally employ policies (in the sense of Markov decision processes) that maximise expected future reward.

On Modalities 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 [55] and [56]. A recent paper showing the current state of the art of modal logic programming languages is [57]. 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 [29] and [10], 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 [58].

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 [59] and [60]. Bach differs from these works in that modalities appear directly inside the language, not in a (meta-level) type system.

On Higher-Order/Functional Programming We turn now to related work in higher-order logic. 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 computational 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 λ Prolog [61]. The logical foundations of λ Prolog are provided by almost exactly the same logic as that underlying Bach (without modalities). However, a different sublogic is used for λ Prolog programs than the equational theories proposed here. In λ Prolog, 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 [62], which is the result of an international collaboration over the last decade. A survey of functional logic programming up to 1994 is in [63] and a more recent survey is given in [64].

On Representing Changes and the Effects of Actions Standard techniques for representing and reasoning about the effects of actions and changes in the world like the situation calculus [65] and variants 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 [66] can also be implemented in Bach. Further, as we saw in Section 6.3, support for probabilistic computation also allow Bayesian tracking concepts from probabilistic artificial intelligence to be implemented directly in Bach.

7.2 Ongoing and Future work

Much remains to be done on the development of Bach. New algorithms and design approaches need to be developed to speed up performance-critical aspects of the language. The basic computational model also needs to be extended with constructs like modules, I/O, and concurrency to make it a practical programming language; these constructs could be adopted directly from (concurrent) Haskell, for example. Also further research is needed on control aspects of the interface between the computation and proof components.

Bach is currently implemented efficiently and completely enough to be useful for agent applications. One interesting application under development at the moment is

a tracking system for vehicles at intersections that is part of a NICTA project aimed at improving the efficiency of road traffic in the Sydney metropolitan area. The tracking system is similar in structure to the example in Section 6.3 with state, a transition model, and an observation model, but considerably more complicated. Generally speaking, in its current form, Bach is an excellent prototyping tool for developing agent applications.

8 Conclusion

This paper has introduced the computational 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. A detailed account of how modalities and modal computations can be supported in a functional programming language is presented.
3. Computations are shown to support a range of agent programming tasks, which include different forms of reasoning with modal and uncertain beliefs.
4. A computational model that tightly integrates equational reasoning and theorem proving is presented. This computational model, with suitable restrictions and control, is shown to make possible complex computational tasks that arise naturally in agent applications.

Acknowledgements We thank Joshua Cole, Rajeev Goré, Will Uther, and Joel Veness for numerous helpful discussions. Joshua Cole provided an early implementation of the theorem prover. NICTA is funded by the Australian Government’s Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia’s Ability and the ICT Research Centre of Excellence programs. The research was partly supported by the Australian Research Council Discovery Project “Foundations and Architectures for Agent Systems”.

References

1. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press (1995)
2. Wooldridge, M.: Reasoning about Rational Agents. MIT Press (2000)
3. Gabbay, D.M., Kurucz, A., Wolter, F., Zakharyashev, M.: Many-Dimensional Modal Logics: Theory and Applications, *Studies in Logic and The Foundations of Mathematics*, vol. 148. Elsevier (2003)
4. Halpern, J.Y.: An analysis of first-order logics of probability. *Artificial Intelligence* **46**(3), 311–350 (1990)
5. Poole, D.: First-order probabilistic inference. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, pp. 985–991 (2003)
6. De Raedt, L., Kersting, K.: Probabilistic logic learning. *SIGKDD Explorations* **5**(1), 31–48 (2003)
7. Getoor, L., Taskar, B. (eds.): Introduction to Statistical Relational Learning. MIT Press (2007)
8. Ng, K.S., Lloyd, J.W.: Probabilistic reasoning in a classical logic. *Journal of Applied Logic* (2008). DOI doi:10.1016/j.jal.2007.11.008. In press
9. Ng, K.S., Lloyd, J.W., Uther, W.T.B.: Probabilistic modelling, inference and learning using logical theories. *Annals of Mathematics and Artificial Intelligence* (2009). In press

10. Fitting, M.: *Types, Tableaus, and Gödel's God*. Kluwer Academic Publishers (2002)
11. Peyton Jones, S., Hughes, J. (eds.): *Haskell98: A non-strict purely functional language*. <http://haskell.org/>
12. Lloyd, J.W.: *Foundations of Logic Programming*, 2nd edn. Springer (1987)
13. Hill, P.M., Lloyd, J.W.: *The Gödel Programming Language*. MIT Press, Cambridge MA (1994)
14. Lloyd, J.W.: Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming* **3** (1999)
15. Lloyd, J.W.: *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer (2003)
16. Lloyd, J.W.: Knowledge representation and reasoning in modal higher-order logic (2007). Available at <http://rsise.anu.edu.au/~jwl>
17. Muskens, R.: Higher order modal logic. In: P. Blackburn, J. van Benthem, F. Wolter (eds.) *Handbook of Modal Logic*. Elsevier (2006)
18. Andrews, P.B.: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press (1986)
19. van Benthem, J., Doets, K.: Higher-order logic. In: D. Gabbay, F. Guenther (eds.) *Handbook of Philosophical Logic*, vol. 1, pp. 275–330. Reidel (1983)
20. Leivant, D.: 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 University Press (1994)
21. Shapiro, S.: Classical logic II – Higher-order logic. In: L. Goble (ed.) *The Blackwell Guide to Philosophical Logic*, pp. 33–54. Blackwell (2001)
22. Church, A.: A formulation of the simple theory of types. *Journal of Symbolic Logic* **5**, 56–68 (1940)
23. Farmer, W.: The seven virtues of simple type theory. *Journal of Applied Logic* **6**(3), 267–286 (2008)
24. Lloyd, J.W., Ng, K.S.: 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 (2008)
25. Lloyd, J.W., Ng, K.S.: 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 (2008)
26. Lloyd, J.W., Ng, K.S.: 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 (2007)
27. Peyton Jones, S.L.: *The Implementation of Functional Programming Languages*. Prentice-Hall (1987)
28. Clark, K.: Negation as failure. In: H. Gallaire, J. Minker (eds.) *Logic and Databases*, pp. 293–322. Plenum Press (1978)
29. Fitting, M., Mendelsohn, R.L.: *First-order Modal Logic*. Kluwer Academic Publishers (1998)
30. del Cerro, L.F., Gasquet, O.: A general framework for pattern-driven modal tableaux. *Logic Journal of the IGPL* **10**(1), 51–83 (2002)
31. Fitting, M.: *First-Order Logic and Automated Theorem Proving*. Springer-Verlag (1990)
32. Kohlhase, M.: Higher-order automated theorem proving. In: W. Bibel, P.H. Schmidt (eds.) *Automated Deduction: A Basis for Applications. Volume I, Foundations: Calculi and Methods*. Kluwer Academic Publishers (1998)
33. Dowek, G.: Higher-order unification and matching. In: *Handbook of automated reasoning*, pp. 1009–1062. Elsevier Science Publishers B. V. (2001)
34. Huet, G.P.: A unification algorithm for typed λ -calculus. *Theoretical Computer Science* **1**, 27–57 (1975)
35. Giese, M.: 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 (2001)
36. Miller, D.A.: *Proofs in higher-order logic*. Ph.D. thesis, Mathematics Department, Carnegie-Mellon University (1983)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. No. 2283 in LNCS. Springer-Verlag (2005)

-
38. Sanner, S.: First-order decision-theoretic planning in structured relational environments. Ph.D. thesis, University of Toronto (2008)
 39. Sanner, S., Boutilier, C.: Practical solution techniques for first-order MDPs. *Artificial Intelligence* (2008). Doi:10.1016/j.artint.2008.11.003
 40. Russell, S.J., Norvig, P.: *Artificial Intelligence: A Modern Approach*, 2nd edn. Prentice-Hall (2002)
 41. Cole, J.J., Gray, M., Lloyd, J.W., Ng, K.S.: 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 (2005)
 42. Rivest, R.L.: Learning decision lists. *Machine Learning* **2**(3), 229–246 (1987)
 43. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): *Multi-Agent Programming: Languages, Platforms and Applications*. Springer (2005)
 44. Bordini, R.H., Braubach, L., Dastani, M., El Fallah Seghrouchni, A., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. *Informatica* **30**, 33–44 (2006)
 45. Rao, A.: 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 (1996)
 46. Bordini, R., Hübner, J., Vieira, R.: Jason and the golden fleece of agent-oriented programming. 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 (2005)
 47. Hindriks, K., de Boer, F., van der Hock, W., Meyer, J.J.C.: Agent programming in 3APL. *Journal of Autonomous Agents and Multi-Agent Systems* **2**(4), 357–401 (1999)
 48. Leite, J., Alferes, J., Pereira, L.: 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 (2002)
 49. Leite, J.A., Alferes, J.J., Pereira, L.M.: 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 (2001)
 50. Dix, J., Zhang, Y.: 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 (2005)
 51. Dix, J., Kraus, S., Subrahmanian, V.: Agents dealing with time and complexity. In: M. Gini, T. Ishida, C. Castelfranchi, W. Johnson (eds.) *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 912–919 (2002)
 52. Ng, R.T., Subrahmanian, V.S.: Probabilistic logic programming. *Information and Computation* **101**(2), 150–201 (1992)
 53. Fisher, M.: 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 (2005)
 54. Clark, K.L., McCabe, F.G.: Go! – A multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence* **41**(2-4), 171–206 (2004)
 55. Orgun, M.A., Ma, W.: 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 (1994)
 56. Gergatsoulis, M.: Temporal and modal logic programming languages. In: A. Kent, J. Williams (eds.) *Encyclopedia of Microcomputers*, vol. 27, pp. 393–408. Marcel Dekker (2001)
 57. Nguyen, L.A.: Multimodal logic programming. *Theoretical Computer Science* **360**, 247–288 (2006)
 58. Gallin, D.: *Intensional and Higher-order Modal Logic*. Kluwer Academic Publishers (1998)
 59. Nanevski, A.: Functional programming with names and necessity. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (2004)
 60. Fairtlough, M., Mendler, M., Moggi, E.: Special issue: Modalities in type theory. *Mathematical Structures in Computer Science* **11**, 507–509 (2001)
 61. Nadathur, G., Miller, D.: Higher-order logic programming. In: D. Gabbay, C. Hogger, A. Robinson (eds.) *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Oxford (1998)

62. Hanus, M. (ed.): Curry: An integrated functional logic language. <http://www.informatik.uni-kiel.de/~curry>
63. Hanus, M.: The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming* **19&20**, 583–628 (1994)
64. Hanus, M.: Multi-paradigm declarative languages. In: *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, LNCS 4670, pp. 45–75. Springer (2007)
65. Reiter, R.: *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press (2001)
66. Thielscher, M.: FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming* **5**(4-5), 533–565 (2005)