Modal Functional Logic Programming

J.W. Lloyd

Computer Sciences Laboratory
The Australian National University
john.lloyd@anu.edu.au

K.S. Ng J. Veness

Symbolic Machine Learning and Knowledge Acquisition, NICTA
{keesiong.ng, joel.veness}@nicta.com.au

Abstract

This paper introduces aspects of a novel modal functional logic programming language called Bach that is an extension of the existing functional logic language Escher. Language facilities available in Bach but not in Escher include (1) support for modalities and (2) an improved theorem-proving capability. We show how the increased expressiveness of Bach can be exploited to produce easy-to-understand programs for solving a variety of computational problems that arise in applications, especially agents applications.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory, Design

Keywords modal functional logic programming, modal logic, equational reasoning, theorem proving

1. Introduction

This paper continues one thread in the development of declarative programming languages that goes back about 15 years. The starting point was the recognition that Prolog (Lloyd 1987) has various flaws that reduces its credibility as a declarative programming language; these flaws include non-declarative meta-programming facilities and the lack of a type system. This motivated the Gödel programming language (Hill and Lloyd 1994) that was closely based on Prolog but had a polymorphic type system and declarative meta-programming facilities. The next step was Escher (Lloyd 1999) 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 (Lloyd 2003) that provides the logic programming idioms. Escher also avoided the highly problematical negation as failure rule by treating negation as just another function. Escher is related to the functional logic language Curry (Hanus ed.).

This paper introduces the language Bach that takes a significant step beyond Escher in that modalities and improved theorem-proving support are included. The improved theorem-proving capability of Bach extends the range of computational problems that can be solved automatically. The main motivation for introducing modal facilities is the existence of the extensive and important class of agent applications. When an agent is deciding what action to perform next it is common for modal considerations to be important; for example, epistemic modalities can be needed because it is necessary to reason about the beliefs of other agents and temporal modalities can be needed because it is necessary to reason about beliefs in the past, present, or future. Thus, Bach can be regarded as a general-purpose, declarative programming language that is particularly well-suited to the development of autonomous agents and multi-agent systems.

The paper is organised as follows. Section 2 contains a summary of the logic underlying Bach. The main equational reasoning component of Bach is described in Section 3. This is followed by a presentation of the theorem proving component in Section 4. Small instructive programming examples are sprinkled throughout the two sections to illustrate important concepts. More significant applications can be found in Section 5. Implementation issues are discussed in Section 6. Section 7 gives a list of related work and we conclude with a statement on what has been achieved in Section 8.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Logic

We outline the most relevant aspects of the logic here, 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 (Lloyd 2007).

Definition 1. An *alphabet* consists of three sets:

1. A set $\mathfrak{T}$ of type constructors.
2. A set $\mathfrak{C}$ of constants.
3. A set $\mathfrak{V}$ of variables.

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

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

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

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

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

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

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

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. If t is a term of type β and x a variable of type α, then $\lambda x.t$ is a term of type $\alpha \to \beta$.
4. If s is a term of type $\alpha \to \beta$ and t a term of type α, then $(s\ t)$ is a term of type β.
5. If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of type $\alpha_1 \times \cdots \times \alpha_n$.
6. If t is a term of type α and $i \in \{1, \ldots, m\}$, then $\Box_i t$ is a term of type α.

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).

Constants can be declared to be rigid; they then have the same meaning in each world (in the semantics). Except in the most sophisticated applications, it is entirely natural for some constants to be rigid; for example, all constants (data constructors and functions alike) in the Haskell prelude can be declared to be rigid. A term is *rigid* if every constant in it is rigid.

The polymorphic version of the logic extends what is given above by also having available parameters which are type variables (denoted by $a, b, c, \ldots$). 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 $\exists_\alpha$, $\forall_\alpha$, and $=_\alpha$, since the types associated with $\exists$, $\forall$, and $=$ are now inferred from the context. The universal closure of a formula φ is denoted by $\forall(\varphi)$.

As is well known, modalities can have a variety of meanings, depending on the application. Some of these are indicated here; more detail can be found in (Fagin et al. 1995), (Gabbay et al. 2003) and (Lloyd 2007), for example.

In multi-agent applications, one meaning for $\Box_i\varphi$ is that 'agent i knows φ'. In this case, the modality $\Box_i$ is written as $\boldsymbol{K}_i$. A weaker notion is that of belief. In this case, $\Box_i\varphi$ means that 'agent i believes φ' and the modality $\Box_i$ is written as $\boldsymbol{B}_i$.

The modalities also have a variety of temporal readings. We adopt the usual modalities $\bigcirc$ ('next'), $\Box$ ('always in the future'), $\Diamond$ ('sometime in the future'), and $\boldsymbol{U}$ ('until'). Dual to these are the past temporal modalities $\bullet$ ('last'), $\blacksquare$ ('always in the past'), $\blacklozenge$ ('sometime in the past'), and $\boldsymbol{S}$ ('since').

A novel feature of the logic is that modalities can be applied to terms that are not formulas. (A formula is a term of type Ω.) Thus terms such as $\boldsymbol{B}_i 42$ and $\bullet A$, where A is a constant, are admitted. Such terms are called modal terms.

The logic can be given a rather conventional semantics in the usual Kripke style for modal logics; the main novelty is giving a semantics to modal terms.

A theory in the logic, 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 interpretation, while local assumptions only have to be true in the actual world in the intended 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.

A Bach program is a theory in the logic. The inference mechanism underlying Bach combines an equational reasoning system and a theorem prover. The equational reasoning system is, in effect, a computational system that significantly extends existing functional programming languages by adding facilities for computing with modalities. The theorem prover is a fairly conventional tableau theorem prover for modal higher-order logic similar to what is proposed in (Fitting 2002). 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 makes possible all kinds of interesting reasoning tasks.

We describe the equational reasoning system next. This will be followed by a discussion of the proof system.

3. Computation

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 consequence of $\mathcal{T}$.

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

Here are the details about a mechanism that addresses the computational 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 the syntax tree of t to s. The notation $t[s/r]_o$ denotes the term obtained from t by replacing s at occurrence o with r. A modal path to a subterm is the sequence of indices of modalities whose scope one passes through 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 using* $\Box_{j_1} \cdots \Box_{j_r}$ *with respect to* $\mathcal{T}$ is a sequence $\{t_i\}_{i=1}^n$ of terms such that the following conditions are satisfied. For $i = 1, \ldots, n-1$, there is

1. a subterm s_i of t_i at occurrence o_i of type α_i, where the modal path to o_i in t_i is $k_1 \ldots k_{m_i}$,
2. (a) a formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall(u_i = v_i)$ in $\mathcal{L}$, or
   (b) a formula $\forall(u_i = v_i)$ in $\mathcal{G}$, or
   (c) a formula $\Box_{j_1} \cdots \Box_{j_r} \Box_{k_1} \cdots \Box_{k_{m_i}} \forall(u_i = v_i)$ that is the theorem of a proof 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 $\forall(u_i = v_i)$ 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.

Note: Some technical details in Definition 4 and Definition 6 below have been surpressed to ease the presentation. Full details can be found in (Lloyd 2007).

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. It is straightforward to extend Definition 4 so that multiple redexes can be selected at each step. Then a common selection rule is the *parallel-outermost* one that selects all outermost subterms that each satisfy the requirements of Definition 4.

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

3.1 Pattern Matching

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

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 matching 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. In the figure, the subterm of a term t at occurrence o is denoted $t|_o$.

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

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

$\theta := \{\}$;

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

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

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

    $s := s[\lambda x.v/\lambda z.(v\{x/z\})]_o$; % z a new variable

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

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

   then

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

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

   else return failure;

return θ;

Figure 1. Algorithm for finding a matching substitution

In the algorithm, the expression $\theta \circ \{x/t|_o\}$ 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 $(v\theta)\varphi$.

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 1. 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. $\lambda \underset{\uparrow}{x}.(f\ x\ (g\ y\ z))$ $\quad \lambda \underset{\uparrow}{z}.(f\ z\ (g\ A\ B))$
1. $\lambda w.(f\ w\ (g\ \underset{\uparrow}{y}\ z))$ $\quad \lambda w.(f\ w\ (g\ \underset{\uparrow}{A}\ B))$ $\quad \{y/A\}$
2. $\lambda w.(f\ w\ (g\ A\ \underset{\uparrow}{z}))$ $\quad \lambda w.(f\ w\ (g\ A\ \underset{\uparrow}{B}))$ $\quad \{z/B\}$
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 2. Let s be $(f\ x\ (g\ x))$ and t be $(f\ y\ (g\ A))$. Then the successive steps of the algorithm are as follows.

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

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

Example 3. Let s be $\lambda x.(f\ x\ y\ z)$ and t be $\lambda x.(f\ x\ A\ (g\ x))$. Then the successive steps of the algorithm are as follows.

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

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

3.2 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.

In general, 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 is 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 before.

Now we give a series of definitions of $=$, connectives, quantifiers, and so on, that constitute the standard equality theory. All substitutions appearing in these definitions are assumed to be ad-

missible. The first definition is that for $=$.

$$
\begin{aligned}
&= :\ a \to a \to \Omega\\
&(\boldsymbol{C}\ x_1 \dots x_n = \boldsymbol{C}\ y_1 \dots y_n) = (x_1 = y_1) \wedge \dots \wedge (x_n = y_n)\\
&\qquad \text{\% where } \boldsymbol{C} \text{ is a data constructor of arity } n.\\
&(\boldsymbol{C}\ x_1 \dots x_n = \boldsymbol{D}\ y_1 \dots y_m) = \bot\\
&\qquad \text{\% where } \boldsymbol{C} \text{ is a data constructor of arity } n, \boldsymbol{D} \text{ is a data}\\
&\qquad \text{\% constructor of arity } m, \text{ and } \boldsymbol{C} \neq \boldsymbol{D}.\\
&((x_1, \dots, x_n) = (y_1, \dots, y_n)) = (x_1 = y_1) \wedge \dots \wedge (x_n = y_n)\\
&\qquad \text{\% where } n = 2, 3, \dots.\\
&(\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}) = (less\ \lambda x.\boldsymbol{u}\ \lambda y.\boldsymbol{v}) \wedge (less\ \lambda y.\boldsymbol{v}\ \lambda x.\boldsymbol{u})
\end{aligned}
$$

The first two schemas in the above definition simply capture the intended meaning of data constructors, while the third captures an important property of tuples. (Note that for the first schema, if $n = 0$, then the body is $\top$.)

The fourth schema is more subtle. 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 the computational part of the reasoning system because it is not computationally useful: showing that $\forall x.((f\ x) = (g\ x))$ is not generally possible as there can be infinitely many values of x to consider. 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. In such cases, one can check for equality in a finite amount of time. The fourth schema relies on the two following definitions.

$$
\begin{aligned}
&less\ :\ (a \to b) \to (a \to b) \to \Omega\\
&less\ \lambda x.\boldsymbol{d}\ z = \top \qquad \text{\% where } \boldsymbol{d} \text{ is a default term.}\\
&less\ (\lambda x.if\ \boldsymbol{u}\ then\ v\ else\ \boldsymbol{w})\ z =\\
&\quad (\forall x.(\boldsymbol{u} \longrightarrow v = (z\ x))) \wedge (less\ (remove\ \lambda x.\boldsymbol{u}\ \lambda x.\boldsymbol{w})\ z)
\end{aligned}
$$

$$
\begin{aligned}
&remove\ :\ (a \to \Omega) \to (a \to b) \to (a \to b)\\
&remove\ s\ \lambda x.\boldsymbol{d} = \lambda x.\boldsymbol{d} \qquad \text{\% where } \boldsymbol{d} \text{ is a default term.}\\
&remove\ s\ \lambda x.if\ \boldsymbol{u}\ then\ v\ else\ \boldsymbol{w} =\\
&\qquad \lambda x.if\ \boldsymbol{u} \wedge \neg(s\ x)\ then\ v\ else\ ((remove\ s\ \lambda x.\boldsymbol{w})\ x)
\end{aligned}
$$

There is a default term for each type. For example, the default term of type Ω is $\bot$ and that of type Int is 0. The intended meaning of $less$ is best given by an illustration. Consider the multisets m and n. Then $less\ m\ n$ is true iff each item in the support of m is also in the support of n and has the same multiplicity there. For sets, $less$ is simply the subset relation.

The following definitions are for the connectives $\wedge$, $\vee$, and $\neg$.

$$
\begin{aligned}
&\wedge\ :\ \Omega \to \Omega \to \Omega\\
&\top \wedge x = x\\
&\bot \wedge x = \bot\\
&(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)\\
&(if\ u\ then\ v\ else\ w) \wedge t = if\ u \wedge t\ then\ v\ else\ w \wedge t\\
&\boldsymbol{u} \wedge (\exists x_1.\cdots\exists x_n.\boldsymbol{v}) = \exists x_1.\cdots\exists x_n.(\boldsymbol{u} \wedge \boldsymbol{v}) \qquad \text{(C1)}\\
&\quad \text{\% where } \boldsymbol{u} \text{ does not contain a free occurrence of any of the } x_i\\
&\boldsymbol{u} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v} = \boldsymbol{u}\{\boldsymbol{x}/\boldsymbol{t}\} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{t}\} \qquad \text{(C2)}\\
&\quad \text{\% where } \boldsymbol{x} \text{ is a variable free in } \boldsymbol{u} \text{ or } \boldsymbol{v} \text{ or both, but not free in } \boldsymbol{t},\\
&\quad \text{\% and } \boldsymbol{t} \text{ is not a variable.}
\end{aligned}
$$

$$
\begin{aligned}
&\vee\ :\ \Omega \to \Omega \to \Omega\\
&\top \vee x = \top\\
&\bot \vee x = x\\
&(if\ u\ then\ \top\ else\ w) \vee t = if\ u\ then\ \top\ else\ w \vee t\\
&(if\ u\ then\ \bot\ else\ w) \vee t = (\neg\ u \wedge w) \vee t
\end{aligned}
$$

$$
\begin{aligned}
&\neg\ :\ \Omega \to \Omega\\
&\neg\ \bot = \top\\
&\neg\ \top = \bot\\
&\neg\ (\neg\ x) = x\\
&\neg\ (x \wedge y) = (\neg\ x) \vee (\neg\ y)\\
&\neg\ (x \vee y) = (\neg\ x) \wedge (\neg\ y)\\
&\neg\ (if\ u\ then\ v\ else\ w) = if\ u\ then\ \neg\ v\ else\ \neg\ w
\end{aligned}
$$

Symmetric versions of some of the above equations have been omitted for brevity here.

These definitions are straightforward, except perhaps for the last two schemas in the definition of $\wedge$. The second last schemas allow the scope of existential quantifiers to be extended provided it does not result in free variable capture.

The last schema allows the elimination of some occurrences of a free variable ($\boldsymbol{x}$, in this case), thus simplifying an expression. A similar schema allowing this kind of simplification also occurs in the definition of Σ below. However, a few words about the expression $\boldsymbol{u} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v}$ are necessary. The intended meaning of this expression is that it is a term such that $(\boldsymbol{x} = \boldsymbol{t})$ 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 $((p \wedge q) \vee r) \wedge ((x = s) \wedge (t \vee u))$.

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 $(\Pi\ \lambda x.t)$.

$$
\begin{aligned}
&\Sigma\ :\ (a \to \Omega) \to \Omega\\
&\exists x.\top = \top\\
&\exists x.\bot = \bot\\
&\exists x_1.\cdots\exists x_n.(\boldsymbol{x} \wedge (x_1 = \boldsymbol{u}) \wedge \boldsymbol{y}) =\\
&\qquad\qquad \exists x_2.\cdots\exists x_n.(\boldsymbol{x}\{x_1/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_1/\boldsymbol{u}\}) \qquad \text{(E)}\\
&\qquad \text{\% where } x_1 \text{ is not free in } \boldsymbol{u}.\\
&\exists x_1.\cdots\exists x_n.(\boldsymbol{u} \vee \boldsymbol{v}) = (\exists x_1.\cdots\exists x_n.\boldsymbol{u}) \vee (\exists x_1.\cdots\exists x_n.\boldsymbol{v})\\
&\exists x_1.\cdots\exists x_n.(if\ \boldsymbol{u}\ then\ \top\ else\ \boldsymbol{v}) =\\
&\qquad if\ \exists x_1.\cdots\exists x_n.\boldsymbol{u}\ then\ \top\ else\ \exists x_1.\cdots\exists x_n.\boldsymbol{v}\\
&\exists x_1.\cdots\exists x_n.(if\ \boldsymbol{u}\ then\ \bot\ else\ \boldsymbol{v}) = \exists x_1.\cdots\exists x_n.(\neg\ \boldsymbol{u} \wedge \boldsymbol{v})
\end{aligned}
$$

$$
\begin{aligned}
&\Pi\ :\ (a \to \Omega) \to \Omega\\
&\forall x_1.\cdots\forall x_n.(\bot \longrightarrow \boldsymbol{u}) = \top\\
&\forall x_1.\cdots\forall x_n.(\boldsymbol{x} \wedge (x_1 = \boldsymbol{u}) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) =\\
&\quad \forall x_2.\cdots\forall x_n.(\boldsymbol{x}\{x_1/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_1/\boldsymbol{u}\} \longrightarrow \boldsymbol{v}\{x_1/\boldsymbol{u}\}) \qquad \text{(A)}\\
&\qquad \text{\% where } x_1 \text{ is not free in } \boldsymbol{u}.\\
&\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{v} \longrightarrow \boldsymbol{t}) =\\
&\quad (\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))\\
&\forall x_1.\cdots\forall x_n.((if\ \boldsymbol{u}\ then\ \top\ else\ \boldsymbol{v}) \longrightarrow \boldsymbol{t}) =\\
&\quad (\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))\\
&\forall x_1.\cdots\forall x_n.((if\ \boldsymbol{u}\ then\ \bot\ else\ \boldsymbol{v}) \longrightarrow \boldsymbol{t}) =\\
&\qquad\qquad \forall x_1.\cdots\forall x_n.(\neg\ \boldsymbol{u} \wedge \boldsymbol{v} \longrightarrow \boldsymbol{t})
\end{aligned}
$$

Next comes the definition for the if_then_else function.

$$if_then_else \; : \; \Omega \times a \times a \rightarrow a$$
$$if \; \top \; then \; u \; else \; v = u$$
$$if \; \bot \; then \; u \; else \; v = v$$

Note that a term of the form $if \; x \; then \; y \; else \; z$ is really a syntactic sugar for $if_then_else \; (x, y, z)$.

The next two equations involve function application and the if_then_else function.

$$(w \; (if \; x \; then \; y \; else \; z)) = if \; x \; then \; (w \; y) \; else \; (w \; z) \quad \text{(I1)}$$
$$((if \; x \; then \; y \; else \; z) \; w) = if \; x \; then \; (y \; w) \; else \; (z \; w) \quad \text{(I2)}$$

There is also the definition corresponding to β-reduction.

$$\lambda x.\boldsymbol{u} : \boldsymbol{\sigma} \rightarrow \boldsymbol{\tau}$$
$$\lambda x.\boldsymbol{u} \; t = \boldsymbol{u}\{x/t\} \quad \text{\% where } \boldsymbol{\sigma} \rightarrow \boldsymbol{\tau} \text{ is the type of } \lambda x.\boldsymbol{u}. \quad \text{(B)}$$

Also included in the standard equality theory is the schema

$$(\Box_i \mathbf{s} \; \mathbf{t}) = \Box_i(\mathbf{s} \; \mathbf{t}), \quad \text{(M1)}$$

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. A similar schema holds for the dual modality $\Diamond_i$ (when β is Ω). Another useful schema in the standard equality theory is

$$\Box_i \boldsymbol{t} = \boldsymbol{t}, \quad \text{(M2)}$$

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

3.3 Examples of Computation

Here are a few examples to illustrate computation.

Example 4. Consider the following definitions of the functions $append$, $permute$ and $delete$, which have been written in the relational style of logic programming.

$$append \; : \; List \; a \times List \; a \times List \; a \rightarrow \Omega$$
$$(append \; (u, v, w)) = ((u = []) \wedge (v = w)) \vee$$
$$\exists r.\exists x.\exists y.((u = r \sharp x) \wedge (w = r \sharp y) \wedge (append \; (x, v, y)))$$

$$permute \; : \; List \; a \times List \; a \rightarrow \Omega$$
$$(permute \; ([], x)) = (x = [])$$
$$(permute \; (x \sharp y, w)) = \exists u.\exists v.\exists z.((w = u \sharp v) \wedge$$
$$(delete \; (u, x \sharp y, z)) \wedge (permute \; (z, v)))$$

$$delete \; : \; a \times List \; a \times List \; a \rightarrow \Omega$$
$$(delete \; (x, [], y)) = \bot$$
$$(delete \; (x, y \sharp z, w)) = ((x = y) \wedge (w = z)) \vee$$
$$\exists v.((w = y \sharp v) \wedge (delete \; (x, z, v)))$$

The intended meaning of $append$ is that it is true iff its third argument is the concatenation of its first two arguments. The intended meaning of $permute$ is that it is true iff its second argument is a permutation of its first argument. The intended meaning of $delete$ is that it is true iff its third argument is the result of deleting its first argument from its second argument.

The notable feature of the above definitions is the presence of existential quantifiers on the RHS of the input equations, so not surprisingly the key statement that makes all this work is concerned with the existential quantifier. To motivate this, consider the computation in Figure 2 that results from the goal $(append \;\; (1 \sharp [], 2 \sharp [], x))$. At one point in the computation, the following term is reached:

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

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

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

Similarly, one can eliminate x' to obtain

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

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

This example illustrates how the definitions in the standard equality theory allow the traditional functional programming style to be extended to encompass the relational style of logic programming. This general technique is called *programming with abstractions* (Lloyd 2003).

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

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

and the goal $(permute \; (1 \sharp 2 \sharp [], x))$ will be reduced to the answer

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

Example 5. Consider the following definition of $f : \sigma \rightarrow Nat$:

$$(f \; x) = if \; x = A \; then \; 42 \; else \; if \; x = B \; then \; 21$$
$$else \; if \; x = C \; then \; 42 \; else \; 0,$$

where $A, B, C : \sigma$. With such a definition, it is straightforward to compute in the 'forward' direction. Thus, for example, $(f \; B)$ can be computed in the obvious way to produce the answer 21.

Less obviously, the definition can be used to compute in the 'reverse' direction. For example, consider the computation of $\{ x \mid (f \; x) = 42 \}$ in Figure 3, which produces the answer $\{A, C\}$. (The notation $\{ x \mid t \}$ is syntactic sugar for the term $\lambda x.t$.) The computation makes essential use of Equations (I1) and (I2), which comes from the definition of if_then_else in the standard equality theory.

Example 6. Consider a theory that includes definitions of the function $f : \sigma \rightarrow Nat$ at the current time and some recent times.

$$\forall x.((f \; x) = if \; (p_4 \; x) \; then \; (\bullet f \; x) \; else \; (\bullet^2 f \; x))$$
$$\bullet \forall x.((f \; x) = if \; (p_3 \; x) \; then \; (\bullet f \; x) \; else \; 0)$$
$$\bullet^2 \forall x.((f \; x) = if \; (p_1 \; x) \; then \; 42 \; else \; 21)$$
$$\bullet^3 \forall x.((f \; x) = 0).$$

Now suppose t is a rigid term of type σ and consider the computation of $(f \; t)$ in Figure 4. Note how earlier definitions for f get used in the computation: at the step $\bullet\underline{(f \; t)}$, the definition at the last time step gets used, and at the step $\bullet^2\underline{(f \; t)}$, the definition from two time steps ago gets used.

Also needed in this computation are the global assumptions (M1) and (M2) from the standard equality theory.

3.4 A Comparison with Escher, Haskell, and Prolog

The equational reasoning component of Bach is essentially an extension of Escher. We end this section with a comparison between

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

Figure 2. Computation of $(append\ (1 \sharp [], 2 \sharp [], x))$. Redexes underlined.

$$\{\ x \mid (= \underline{(f\ x)}\ 42)\ \}$$
$$\{\ x \mid (\underline{(=\ \ if\ x = A\ then\ 42\ else\ if\ x = B\ then\ 21\ else\ if\ x = C\ then\ 42\ else\ 0)}\ 42)\ \}$$
$$\{\ x \mid \underline{(if\ x = A\ then\ (=\ 42)\ else\ (=\ \ if\ x = B\ then\ 21\ else\ if\ x = C\ then\ 42\ else\ 0)\ 42)}\ \}$$
$$\{\ x \mid if\ x = A\ then\ \underline{(42 = 42)}\ else\ ((=\ \ if\ x = B\ then\ 21\ else\ if\ x = C\ then\ 42\ else\ 0)\ 42)\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ (\underline{(=\ \ if\ x = B\ then\ 21\ else\ if\ x = C\ then\ 42\ else\ 0)}\ 42)\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ \underline{(if\ x = B\ then\ (=\ 21)\ else\ (=\ if\ x = C\ then\ 42\ else\ 0)\ 42)}\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \underline{(21 = 42)}\ else\ ((=\ if\ x = C\ then\ 42\ else\ 0)\ 42)\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ (\underline{(=\ if\ x = C\ then\ 42\ else\ 0)}\ 42)\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ \underline{(if\ x = C\ then\ (=\ 42)\ else\ (=\ 0)\ \ 42)}\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \underline{(42 = 42)}\ else\ (0 = 42)\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \top\ else\ \underline{(0 = 42)}\ \}$$
$$\{\ x \mid if\ x = A\ then\ \top\ else\ if\ x = B\ then\ \bot\ else\ if\ x = C\ then\ \top\ else\ \bot\ \}$$

Figure 3. Computation of $\{\ x \mid (f\ x) = 42\ \}$

Escher and Bach. Before examining how Bach is different from Escher, it will be helpful to first understand the relationship between Escher and Haskell, and that between Escher and Prolog.

Escher vs Haskell At the logic level, every Haskell program is an Escher program [1], and every Escher program is a (syntactically-correct) Haskell program which may not compile. In that sense, Escher is a superset of Haskell. The difference between Escher and Haskell comes down to the following two points.

- Haskell allows pattern matching only on data constructors. Escher extends this by allowing pattern matching on function symbols and lambda abstractions in addition to data constructors. Examples of equations that Haskell cannot handle include Equation (E) and several others in the standard equality theory. Thus Haskell cannot perform the kind of logic-programming-style computations illustrated in Example 4.
- 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 5.

The extra expressiveness afforded by Escher comes with a price, however. Some common optimisation techniques developed for efficient compilation of Haskell code (see (Peyton Jones 1987) for a survey) cannot be used in the implementation of Escher. In other words, efficiency is at present still an issue for Escher, although we expect this to become less of a problem over time as computing power increases and new implementation techniques are developed.

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. Perhaps surprisingly, there is actually a sig-

[1] This is not exactly true at the implementation level, however, since Haskell has several advanced language features not currently available in Escher. But this is only a software maturity issue. Given more development time, these language features will be implemented and Escher will then be able to run any valid Haskell program.

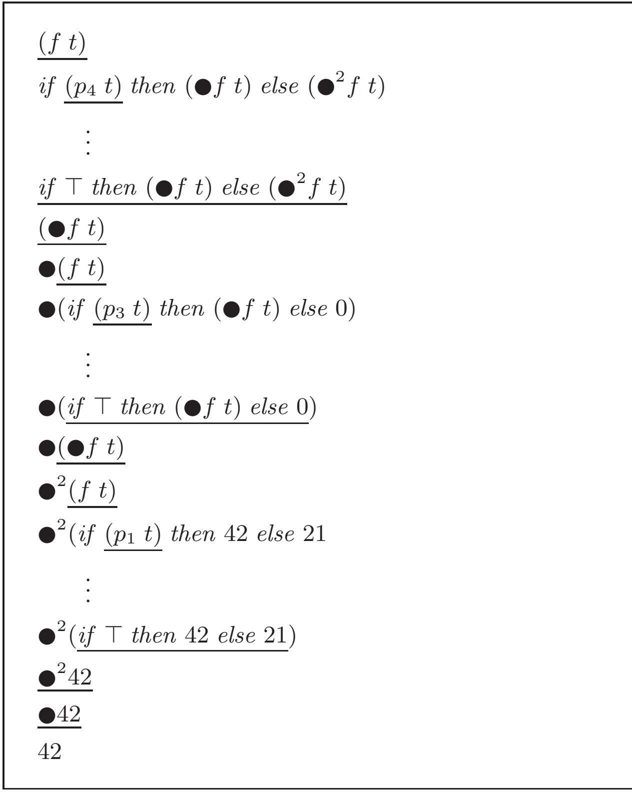

Figure 4. Computation of $(f\ t)$

nificant overlap between Escher and Prolog. In fact, any Prolog program defined without using cuts can be mechanically translated into Escher via Clark's completion algorithm (Clark 1978). For example, the definition of *append* given in Example 4 is essentially the completion of the following Prolog definition:

$append\ ([], L, L).$

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

There is of course nothing that can be done with a cut that cannot be done in a more natural way in the functional programming setting, so nothing is lost here.

Procedurally, there is also a difference between Escher and Prolog in that Prolog computes alternative answers one at a time via backtrackking whereas Escher returns all alternative answers in a disjunction (a set). This point is illustrated in Example 4.

Bach vs Escher We can now compare Bach and Escher. An Escher *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 f. In contrast, a Bach input equation is a term of the form

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

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

- The restriction on the form of the LHS of an Escher statement is dropped in Bach. Equation (I1), which we have seen serves an important role in supporting 'reverse'-direction computations in Example 5, is an example of an equation supported in Bach but not in Escher. The extra flexibility afforded in Bach comes at a small price in the form of a more computationally expensive pattern matching algorithm.
- Modalities are only supported in Bach; Escher cannot perform the kind of computations illustrated in Example 6.

We have concentrated on the 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 wide variety of tasks, this is fundamentally a limited set. In contrast, Bach has a full-scale theorem prover as a subsystem and the interaction between computation and proof makes possible all kinds of interesting computational tasks. The proof component is described next.

4. Theorem Proving

In general, an input equation to a computation can be a theorem that is proved by the theorem-proving component of Bach. Here are the details of a tableau proof system that can, given a theory $\mathcal{T}$ and a formula φ, determine whether φ is a consequence of $\mathcal{T}$.

The system employs prefixed formulas as is often the case for modal logics. 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 Figures 5 and 6. Generally speaking, these rules are well known (see, for example, (Fitting and Mendelsohn 1998) and (Fitting 2002)), but the versions here differ in some details, in particular, in the use of admissibility assumption in the universal, abstraction, and substitutivity rules.

Definition 6. Let $\mathcal{T}$ be a theory. A *proof 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 5 or 6 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.

   (b) there is a result η of a computation and a branch is extended with the prefixed formula $1\ \eta$.
3. Each branch of T_n contains nodes labelled by $\sigma\ \psi$ and $\sigma\ \neg\psi$, for some prefix σ and formula ψ.

Each T_i is called a *tableau*. A branch of a tableau 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 is *closed* if each branch is closed; otherwise, the tableau is *open*. The formula φ is called the *theorem* of the proof; this is denoted by $\mathcal{T} \vdash \varphi$.

The following soundness result can be proved.

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

Here are some examples of proof.

(Conjunctive rules) For any prefix σ,

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

(Disjunctive rules) For any prefix σ,

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

(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 $i \in \{1, \ldots, m\}$,

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

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

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

(Existential rules) For any prefix σ, if y is a variable of type α new to the branch,

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

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

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

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

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

(Reflexivity rule) If t is a term of type α 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 of type α, and $\{x/s\}$ and $\{x/t\}$ are admissible with respect to φ,

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

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

$$\frac{}{\sigma\ \ \psi}$$

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

$$\frac{}{1\ \ \psi}$$

($\top$ introduction rules) For any prefix σ,

$$\frac{}{\sigma\ \ \top} \qquad \frac{}{\sigma\ \ \neg\bot}$$

Figure 5. Tableau rules

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

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

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

$$\frac{1\ \ \varphi}{1\ \ \psi} \qquad \frac{1\ \ \neg\psi}{1\ \ \neg\varphi}$$

Figure 6. More tableau rules

Example 7. Suppose we have a theory that includes the following as global assumptions.

$$\mathit{proj_2} : \mathit{String} \times \mathit{Int} \rightarrow \mathit{Int}$$
$$(\mathit{proj_2}\ (x, y)) = y$$
$$\mathit{evenperfect} : \mathit{Int} \rightarrow \Omega$$
$$(\mathit{evenperfect}\ x) = \exists n.(n \in \mathbb{N} \wedge (x = 2^{n-1}(2^n - 1)))$$

Consider the problem of simplifying the term

$$((\mathit{proj_2}\ x) = 496) \wedge (\mathit{evenperfect}\ (\mathit{proj_2}\ x)). \qquad (1)$$

(This kind of problem arises naturally in belief acquisition applications (Lloyd and Ng 2007a).) There is not much the computation system can do other than to expand out the second conjunct. However, we can show using the proof system that

$$\forall x.(((\mathit{proj_2}\ x) = 496) \longrightarrow (\mathit{evenperfect}\ (\mathit{proj_2}\ x))). \qquad (2)$$

The proof is in Figure 7. An explanation of the proof follows: Item

1 $\neg\forall x.(((\mathit{proj_2}\ x) = 496) \longrightarrow (\mathit{evenperfect}\ (\mathit{proj_2}\ x)))$ 1.
1 $\neg((\mathit{proj_2}\ y) = 496) \longrightarrow (\mathit{evenperfect}\ (\mathit{proj_2}\ y))$ 2.
1 $((\mathit{proj_2}\ y) = 496)$ 3.
1 $\neg(\mathit{evenperfect}\ (\mathit{proj_2}\ y))$ 4.
1 $\neg(\mathit{evenperfect}\ 496)$ 5.
1 $(\mathit{evenperfect}\ 496) = \top$ 6.
1 $\neg\top$ 7.
1 $\top$ 8.

Figure 7. Proof of Formula (2)

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 by lemma introduction from a computation of $(\mathit{evenperfect}\ \ 496)$; 7 is from 5 and 6 by the substitutivity rule; 8 is by the $\top$ introduction rule; the tableau now closes by 7 and 8.

Recognising the fact that $(p \longrightarrow q) = ((p \wedge q) = p)$, we can construct a new input equation

$$((\mathit{proj_2}\ x) = 496) \wedge (\mathit{evenperfect}\ (\mathit{proj_2}\ x)) = ((\mathit{proj_2}\ x) = 496),$$

from which (1) can be simplified to $((\mathit{proj_2}\ x) = 496)$.

Example 8. Suppose we have a theory that includes the following

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

as local assumptions. Using the global assumption

$$\bullet \boldsymbol{B}\boldsymbol{\varphi} \longrightarrow \boldsymbol{B}\bullet\boldsymbol{\varphi} \qquad (3)$$

it is easy to show that, for each $i \in \{1, \ldots, 5\}$, $\boldsymbol{B}\bullet^i\varphi_i$ is a theorem of the belief base. Figure 8 shows the proof of $\boldsymbol{B}\bullet^2\varphi_2$.

An explanation of the proof is as follows. 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 implicational assumption (3); 4 is from 3 by a possibility rule; 5 is from 4 by a derived rule from (3); 6 is from 2 by a necessity rule; the tableau now closes by 5 and 6.

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

Figure 8. Proof of $\boldsymbol{B} \bullet^2 \varphi$.

Theoremhood is, of course, undecidable in the case of higher-order logic, but it is still possible to prove theorems in many cases of practical interest. For any one application, it is possible to examine the kind of theorem-proving tasks that will arise in that application and show that they will all terminate. (Otherwise the application has to be redesigned and re-implemented in order to achieve this.) What we have in mind here is that each application is a substantial system and that it makes sense to expend effort on ensuring the desired termination, in the same way as it makes sense to use conventional software engineering techniques to ensure the correctness of the software that implements an application.

5. Applications

We look at some larger-scale applications in this section.

5.1 Logic Puzzles

Bach can be used to express and solve all kinds of logic puzzles. We present here a beautifully declarative Sudoku solver written in Bach, using the puzzle in Figure 9 for illustration.

6			1		8	2		3
	2			4			9	
8		3			5	4		
5		4	6		7			9
	3						5	
7			8		3	1		2
		1	7			9		6
	8			3			2	
3		2	9		4			5

Figure 9. A Sudoku puzzle: Fill in the grid so that every row, every column and every 3×3 block contains the digits 1 – 9.

We represent the puzzle as a tuple of integers. The problem then is to find a suitable instantiation of the variables.

$Square = Int \times \cdots \times Int$ -- dimension = 81

$puzzle : Square$

$$
\begin{aligned}
puzzle = (&6, x_1, x_2, 1, x_3, 8, 2, x_4, 3,\\
&x_5, 2, x_6, x_7, 4, x_8, x_9, 9, x_{10},\\
&8, x_{11}, 3, x_{12}, x_{13}, 5, 4, x_{14}, x_{15},\\
&5, x_{16}, 4, 6, x_{17}, 7, x_{18}, x_{19}, 9,\\
&x_{20}, 3, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, 5, x_{26},\\
&7, x_{27}, x_{28}, 8, x_{29}, 3, 1, x_{30}, 2,\\
&x_{31}, x_{32}, 1, 7, x_{33}, x_{34}, 9, x_{35}, 6,\\
&x_{36}, 8, x_{37}, x_{39}, 3, x_{40}, x_{41}, 2, x_{42},\\
&3, x_{43}, 2, 9, x_{44}, 4, x_{45}, x_{46}, 5 \quad)
\end{aligned}
$$

Specific rows, columns, and blocks can be retrieved from $puzzle$ using the following operations.

$row : Int \to Square \to List\ Int$

$(row\ x\ z) = (map\ \lambda y.(proj\ (y + 9x)\ z)\ idxs)$

$col : Int \to Square \to List\ Int$

$(col\ x\ z) = (map\ \lambda y.(proj\ (9y + x)\ z)\ idxs)$

$block : Int \to Square \to List\ Int$

$(block\ x\ z) =$

$(map\ \lambda y.(proj\ ((proj\ y\ offsets) + (start_cell\ x))\ z)\ idxs)$

$idxs : List\ Int$

$idxs = [0, 1, 2, 3, 4, 5, 6, 7, 8]$

$offsets : Int \times \cdots \times Int$ -- dimension = 9

$offsets = (0, 1, 2, 9, 10, 11, 18, 19, 20)$

$start_cell : Int \to Int$

$(start_cell\ x) = 9(3\lfloor x/3 \rfloor) + 3(x \,\%\, 3)$

Blocks are numbered from left-to-right, top-to-bottom. Adding the index i into the top-left cell ($start_cell$) of any given block with the numbers in $offsets$ gives us the indices into all the cells in the block.

Every row, column, and block must be a permutation of the numbers 1 to 9. This translates into a total of 27 constraints on the variables. These constraints can be folded together using $\wedge$ to form one big constraint on the variables, which when simplified gives us the answer to the Sudoku puzzle. This kind of reasoning leads us to the following solver.

$sdkConstraint : List\ Int \to \Omega$

$(sdkConstraint\ x) = (permute\ ([1, 2, 3, 4, 5, 6, 7, 8, 9], x))$

$solve : Square \to \Omega$

$$
\begin{aligned}
(solve\ x) = (&fold\ \wedge\ (map\ \lambda y.(fold\ \wedge\\
&[(sdkConstraint\ (row\ y\ x)), (sdkConstraint\ (col\ y\ x)),\\
&(sdkConstraint\ (block\ y\ x))])\ idxs))
\end{aligned}
$$

Given the term $(solve\ puzzle)$, Bach returns the following answer:

$$
\begin{aligned}
&(x_1 = 4) \wedge (x_2 = 5) \wedge (x_3 = 9) \wedge (x_4 = 7) \wedge (x_5 = 1) \wedge\\
&(x_6 = 7) \wedge \cdots \wedge (x_{44} = 6) \wedge (x_{45} = 8) \wedge (x_{46} = 1).
\end{aligned}
$$

The Sudoku solver just described makes essential use of the programming with abstractions technique presented in Example 4.

5.2 Databases

We next look at databases. 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, either in the form of a relational database or a Prolog program. But this standard relational representation is not a good one because it 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:

$tv_guide : Occurrence \to Program,$

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.

$$
\begin{aligned}
&\boldsymbol{B}_t\,\forall x.((tv_guide\ x) = \\
&\quad \mathit{if}\ (x = ((1,1,2007),(21,30),ABC)) \\
&\quad \mathit{then}\ (\text{“}\mathit{The\ Bill}\text{”}, 50, \mathit{Drama}, M, \text{“}\mathit{Sun}\ \ldots\text{”}) \\
&\quad \mathit{else\ if}\ (x = ((1,1,2007),(19,00),ABC)) \\
&\quad \mathit{then}\ (\text{“}\mathit{ABC\ News}\text{”}, 30, \mathit{News}, G, \text{“}\mathit{The}\ \ldots\text{”}) \\
&\quad \mathit{else\ if}\ (x = ((1,1,2007),(20,30),TEN)) \\
&\quad \mathit{then}\ (\text{“}\mathit{Numb3rs}\text{”}, 60, \mathit{Crime}, M, \text{“}\mathit{When}\ \ldots\text{”}) \\
&\quad \vdots \\
&\quad \mathit{else}\ (\text{“”}, 0, NA, NA, \text{“”})),
\end{aligned}
$$

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

Listed below are some typical queries we can answer using the definition. Each query is stated formally and the answer computed is given. The computational mechanism with which the answers are computed is explained earlier in Example 5. All computations are done in the context of $\boldsymbol{B}_t$.

1. Find the program at occurrence $((1,1,2007),(20,30),TEN)$.

   Query: $(tv_guide\ ((1,1,2007),(20,30),TEN))$.

   Answer: $(\text{“}\mathit{Numb3rs}\text{”}, 60, \mathit{Crime}, M, \text{“}\mathit{When}\ \ldots\text{”})$.

2. Find the time and channel “The Bill” is screened on 1 Jan 2007.

   Query: $\exists y.((y = (tv_guide\ ((1,1,2007),t,c))) \wedge ((projTitle\ y) = \text{“}\mathit{The\ Bill}\text{”}))$.

   Answer: $(t = (21,30)) \wedge (c = ABC)$.

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

   Query: $\{\, x \mid \exists y.((x = (tv_guide\ y)) \wedge ((projClassification\ x) = M)) \,\}$.

   Answer: $\{\, (\text{“}\mathit{The\ Bill}\text{”}, 50, \mathit{Drama}, M, \text{“}\ldots\text{”}), (\text{“}\mathit{Numb3rs}\text{”}, 60, \mathit{Crime}, M, \text{“}\ldots\text{”}), \ldots \,\}$.

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

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

   Answer: $\{\, (\text{“}\mathit{ABC\ News}\text{”}, 30, \mathit{News}, G, \text{“}\ldots\text{”}), \ldots \}$.

   This last example shows how tv_guide can be used in conjunction with other functions, in the same way relational databases 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 described in Example 5, albeit more efficiently.

5.3 Belief Acquisition in Multi-Agent Systems

We end the section with an application in belief acquisition in multi-agent systems. In particular, we'll look at a TV recommender agent described in (Cole et al. 2005). 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 $\boldsymbol{B}_t$ be the belief modality for the TV agent, $\boldsymbol{B}_a$ the belief modality for Alice, $\boldsymbol{B}_b$ the belief modality for Bob, and $\boldsymbol{B}_c$ the belief modality for Cathy. Thus part of the TV agent's belief base has the following form:

$$
\begin{aligned}
&\boldsymbol{B}_t\boldsymbol{B}_a\ \forall x.((likes\ x) = \varphi_0) \\
&\bullet\boldsymbol{B}_t\boldsymbol{B}_a\ \forall x.((likes\ x) = \varphi_1) \\
&\quad\vdots \\
&\bullet^{n-1}\boldsymbol{B}_t\boldsymbol{B}_a\ \forall x.((likes\ x) = \varphi_{n-1}) \\
&\bullet^{n}\boldsymbol{B}_t\forall x.(\blacklozenge\boldsymbol{B}_a(likes\ x) = \bot) \\
&\boldsymbol{B}_t\boldsymbol{B}_b\ \forall x.((likes\ x) = \psi_0) \\
&\bullet\boldsymbol{B}_t\boldsymbol{B}_b\ \forall x.((likes\ x) = \psi_1) \\
&\quad\vdots \\
&\bullet^{k-1}\boldsymbol{B}_t\boldsymbol{B}_b\ \forall x.((likes\ x) = \psi_{k-1}) \\
&\bullet^{k}\boldsymbol{B}_t\forall x.(\blacklozenge\boldsymbol{B}_b(likes\ x) = \bot) \\
&\boldsymbol{B}_t\boldsymbol{B}_c\ \forall x.((likes\ x) = \xi_0) \\
&\bullet\boldsymbol{B}_t\boldsymbol{B}_c\ \forall x.((likes\ x) = \xi_1) \\
&\quad\vdots \\
&\bullet^{l-1}\boldsymbol{B}_t\boldsymbol{B}_c\ \forall x.((likes\ x) = \xi_{l-1}) \\
&\bullet^{l}\boldsymbol{B}_t\forall x.(\blacklozenge\boldsymbol{B}_c(likes\ x) = \bot),
\end{aligned}
$$

for suitable φ_i, ψ_i, and ξ_i. The form these can take is explained in (Cole et al. 2005).

In the beginning, the belief base contains the formula

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

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

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

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

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

$$group_likes : Program \rightarrow \Omega$$

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

$$
\begin{aligned}
&\boldsymbol{B}_t\forall x.((x = (\text{“}\mathit{ABC\ news}\text{”}, \mathit{30}, \mathit{News}, G, \text{“}\ldots\text{”})) \longrightarrow (group_likes\ x)) \\
&\boldsymbol{B}_t\forall x.(((projGenre\ x) = Sports) \longrightarrow (group_likes\ x)).
\end{aligned}
$$

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

$$\begin{aligned}&\boldsymbol{B}_t \forall x.((\mathit{group_likes}\ x) = \\ &\quad \mathit{if}\ ((\wedge_3\ \boldsymbol{B}_a \mathit{likes}\ \boldsymbol{B}_b \mathit{likes}\ \boldsymbol{B}_c \mathit{likes})\ x)\ \mathit{then}\ \top \\ &\quad \mathit{else\ if}\ \ ((\wedge_3\ \blacklozenge \boldsymbol{B}_a \mathit{likes}\ \boldsymbol{B}_b \mathit{likes}\ \blacklozenge \boldsymbol{B}_c \mathit{likes})\ \ x)\ \mathit{then}\ \top \\ &\quad \vdots \\ &\quad \mathit{else}\ \bot).\end{aligned}$$

The algorithm used to acquire the definition is a generalisation of Rivest's decision-list learning algorithm (Rivest 1987). We shall not be concerned with the actual algorithm here, details of which can be found in (Lloyd and Ng 2007b) and (Lloyd and Ng 2007a). 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

$$(x = (\text{“}\mathit{ABC\ news}\text{”}, \mathit{30}, \mathit{News}, G, \text{“}\ldots\text{”})) \wedge \\ ((\wedge_3\ \boldsymbol{B}_a \mathit{likes}\ \boldsymbol{B}_b \mathit{likes}\ \boldsymbol{B}_c \mathit{likes})\ \ x)$$

and

$$((\mathit{projGenre}\ x) = \mathit{Sports}) \wedge \\ ((\wedge_3\ \blacklozenge \boldsymbol{B}_a \mathit{likes}\ \boldsymbol{B}_b \mathit{likes}\ \blacklozenge \boldsymbol{B}_c \mathit{likes})\ \ x)$$

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

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

It should be clear that all the different computational (sub)tasks illustrated in Examples 6, 7 and 8 are needed in tight integration of each other to solve the above computational problems.

6. Implementation Issues

Implementation Language We have a prototype implementation of a Bach interpreter written in C++. An industrial-strength implementation is currently being developed using Lisp. We opted for Lisp as the implementation language because it has a good balance of computational efficiency and language expressiveness, an important feature given our limited programming resources. There is a rich set of library functions in Lisp. Bootstrapping from these provides us with a mature and efficient low-level system call library. We find the following two features of Lisp particularly useful in our development:

- built-in support for garbage collection and symbolic types;
- strong support for numeric computation, most notably large integers and complex numbers.

Another noteworthy advantage that comes from using Lisp is the ease with which the existing interpreter can be converted into a compiler once it reaches a more mature state. The macro facility of Lisp can be used to transform performance-critical portions of code into specialised Lisp routines that can then be compiled and optimised by the Lisp environment. Orders-of-magnitude improvement in efficiency can be obtained with minimal effort this way.

Standard Equality Theory Most of the equations in the standard equality theory can be accepted as is by the Bach interpreter. The more complex ones are implemented as system-level subroutines. These include the first three equations for $= : a \to a \to \Omega$ and Equations (C1), (C2), (E), (A), and (B). These equations are tried before any other equations in pattern matchings.

Syntactical Variables In theory, syntactical variables in a schema must first be instantiated *before* the schema can be pattern-matched against a candidate redex. This is realized in practice by adding to the pattern matching algorithm (Figure 1) a special case for when $s|_o$ is a syntactical variable, in which case $s|_o$ is instantiated with $t|_o$, subject to $t|_o$ satisfying all the side conditions (if there are any) on the syntactical variable $s|_o$.

Redex Selection The overall redex-selection strategy is leftmost-outermost reduction, which gives lazy evaluation. This is, however, not strictly followed. Input equations can be graded, in which case we perform leftmost-outermost reduction using only level 1 input equations to begin with, and in general moving from level i to level $i+1$ only when no redex can be found using level i input equations. Fine-grained control over evaluation order can be achieved using this mechanism.

Theorem Prover The tableau theorem proving system was implemented using the framework presented in (del Cerro and Gasquet 2002). Most of the tableau rules in Figures 5 and 6 can be easily implemented. The main exception is the universal rule. It is in general very hard to know what is a good choice of term to instantiate a universally quantified variable. We use a variant of the incremental closure algorithm described in (Giese 2001) to implement the universal rule. Backtracking algorithms can be used instead.

Interactions between Computation and Proof The rules and strategies governing the interactions between computation and proof are not well developed at the moment. We are investigating the use of tactics and tacticals to improve this aspect of the system.

We plan to release the system later in the year.

7. Related Work

This section contains a discussion of related work.

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 (Orgun and Ma 1994) and (Gergatsoulis 2001). A recent paper showing the current state of the art of modal logic programming languages is (Nguyen 2006). 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 either provide epistemic modalities or temporal modalities, but not both. Bach extends typical modal higher-order theorem proving systems, such as those in (Fitting and Mendelsohn 1998) and (Fitting 2002), largely in that it also has an equational reasoning component.

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 (Nanevski 2004) and (Fairtlough et al. 2001). Bach differs from these works in that modalities appear directly inside the language, not in a (meta-level) type system.

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 also contains a theorem-proving system, it is modal, and it admits logic programming idioms through programming with abstractions.

In the 1980s, higher-order programming in the logic programming community was introduced through the language λProlog (Nadathur and Miller 1998). The logical foundations of λProlog are provided by almost exactly the same logic as that underlying Bach (with the modal facilities removed). However, a different sublogic is used for λProlog programs than the equational theories proposed here. In λProlog, program statements are higher-order hereditary Harrop formulas, a generalisation of the definite clauses used by Prolog. The language provides an elegant use of λ-terms as data structures, meta-programming facilities, universal quantification and implications in goals, amongst other features. A more modern and well developed higher-order logic programming system is Mercury (Henderson et al. 2006).

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 (`http://www.informatik.uni-kiel.de/~curry`), which is the result of an international collaboration over the last decade. A survey of functional logic programming up to 1994 is in (Hanus 1994).

8. Conclusion

This paper has introduced a novel modal functional logic programming language called Bach. The main innovation in Bach is the provision of native support for modalities as a basic language construct in the functional programming context. As shown in the paper, this increased expressiveness can be put to good use in the development of dynamic and multi-agent systems.

A second contribution of Bach is the integration of equational reasoning and theorem proving in a unified computational system. The resultant enhanced support for logic programming in the functional context represents another useful step in the development of functional logic programming languages.

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 language also need to be extended with constructs like modules and I/O to make it a practical programming language.

Acknowledgments

Many thanks to Joshua Cole and Rajeev Goré for numerous helpful discussions. We are particularly grateful to Joshua Cole for providing an early implementation of the theorem prover. NICTA is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

J. J. Cole, M. Gray, J. W. Lloyd, and K. S. Ng. Personalisation for user agents. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 603–610, 2005.

L. F. del Cerro and O. Gasquet. A general framework for pattern-driven modal tableaux. *Logic Journal of the IGPL*, 10(1):51–83, 2002.

R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

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

M. Fitting. *Types, Tableaus, and Gödel's God*. Kluwer Academic Publishers, 2002.

M. Fitting and R. L. Mendelsohn. *First-order Modal Logic*. Kluwer Academic Publishers, 1998.

D. M. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-Dimensional Modal Logics: Theory and Applications*, volume 148 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 2003.

M. Gergatsoulis. Temporal and modal logic programming languages. In A. Kent and J. Williams, editors, *Encyclopedia of Microcomputers*, volume 27, pages 393–408. Marcel Dekker, 2001.

M. Giese. Incremental closure of free variable tableaux. In *Proceedings of International Joint Conference on Automated Reasoning, Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer-Verlag, 2001.

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

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

F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, and M. Brown. *The Mercury Language Reference Manual*. 2006.

P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge MA, 1994.

J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer, 2003.

J. W. Lloyd. Knowledge representation and reasoning in modal higher-order logic. `http://rsise.anu.edu.au/~jwl`, 2007.

J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.

J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

J. W. Lloyd and K. S. Ng. Belief acquisition for agents. In preparation, 2007a.

J. W. Lloyd and K. S. Ng. Learning modal theories. In S. Muggleton and R. Otero, editors, *Proceedings of the 16th International Conference on Inductive Logic Programming*, 2007b.

G. Nadathur and D. Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Oxford, 1998.

A. Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, School of Computer Science, Cargenie Mellon University, 2004.

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

M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In D. Gabbay and H. Ohlbach, editors, *Proceedings of the First International Conference on Temporal Logics (ICTL'94)*, volume 827 of *LNAI*, pages 445–479. Springer, 1994.

S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.