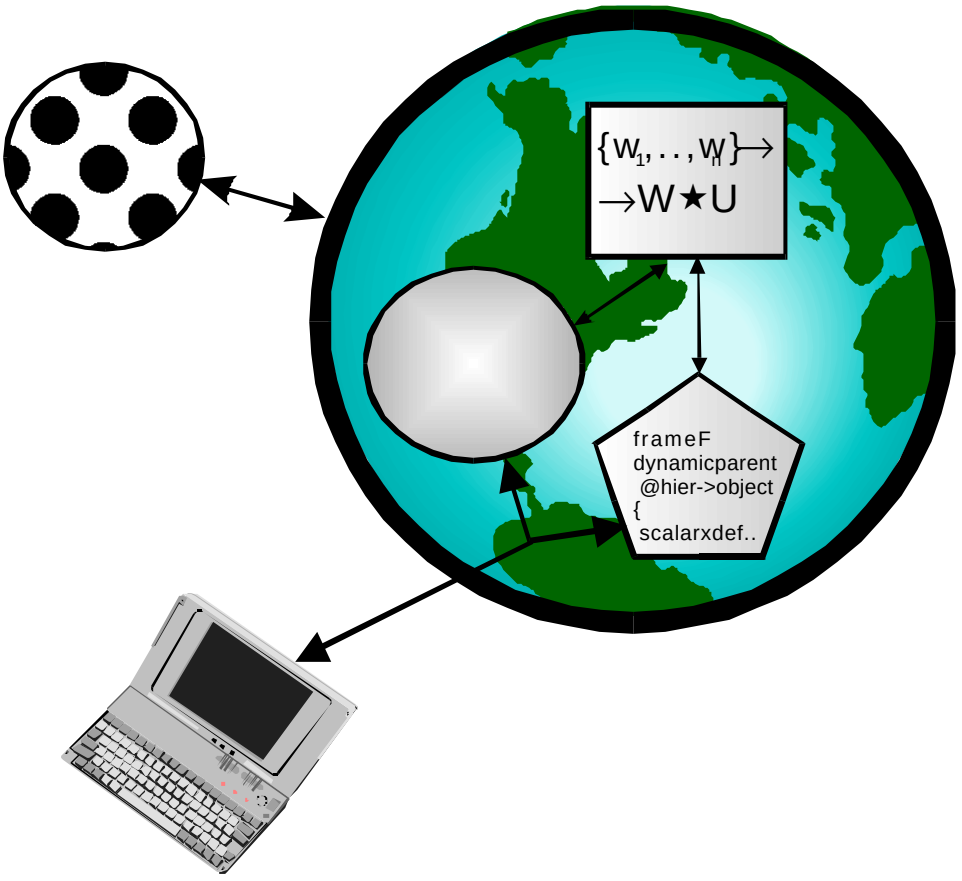
Ñîøíèêîâ Ä.Â.

Ëîãè÷åñêèé âûâîä íà îñíîâå óäàëåííîãî âûçîâà è âêëþ÷åíèÿ â ñèñòåìàõ ñ ðàñïðåäåëåííîé ôðåéìîâîé èåðàðõèåé



Сошников Д.В.

Логический вывод на основе удаленного вызова и включения в системах с распределенной фреймовой иерархией

Под редакцией к.ф.-м.н., доц. ЗАЙЦЕВА В.Е.

Рекомендовано к изданию
кафедрой 806
31 октября 2001 г.

Москва
«Вузовская книга»
2002

ББК 32.973.26 – 018.2
С 54

Р е ц е н з е н т ы :
доктор технических наук, профессор *В.Э.Вольфенгаген*
кандидат физ.-мат. наук *Д.А.Иванов*

Сошников Д.В.
С 54 Логический вывод на основе удаленного вызова и включения в системах с распределенной фреймовой иерархией // Под ред. В.Е.Зайцева. — М.: «Вузовская книга», 2002. — 48 с.: ил.
ISBN 5-9502-0025-X

В работе рассматриваются основные принципы построения распределенных интеллектуальных систем на базе продукционно-фреймового представления знаний, в которых знания, сосредоточенные в различных узлах компьютерной сети, могут использоваться совместно для решения определенной задачи в процессе коллективного распределенного вывода. Фреймовая модель позволяет также строить интерфейс базы знаний с объектно-ориентированными императивными языками, интегрируя объекты на традиционных языках программирования в единую объектно-фреймовую иерархию, а также с реляционными базами данных, путем введения экстенсионалов или фреймовых классов, порождающих необходимые фреймы-экземпляры на основании данных из реляционной таблицы. Рассматривается семантика распределенного логического вывода, основанная на операциях вычисления статических и мобильных ссылок. Для комбинированного монотонного локального вывода определяется семантика неподвижной точки, основанная на функциях вычисления значения $\|\cdot\|$ и целенаправленного прямого вывода $\Phi_{\langle f,s\rangle}$. Кратко рассматриваются вопросы реализации на основе предложенной модели программного инструментария JULIA, основанного на открытых стандартах CORBA и XML.

ББК 32.973.26 – 018.2

ISBN 5-9502-0025-X

Оглавление

1. Введение

В последние годы наблюдается стремительный рост инфраструктуры компьютерных сетей, как общемировых (Интернет), так и масштаба предприятия (интранет/экстранет). Расширение технической базы требует соответствующего развития системного и прикладного программного обеспечения для адаптации его к сетевой среде, а также разработки методологических подходов к организации процесса распределенной деятельности. Разработаны развитые методологии и технологии построения распределенных систем баз данных, информационных систем автоматизации документооборота и др., однако применительно к интеллектуальным системам в настоящее время еще не существует законченных решений. Следует отметить, что развитие теории интеллектуальных распределенных систем представляет особый интерес как для создания эффективных средств представления, структурирования и поиска информации (знаний) в компьютерных сетях, так и с точки зрения создания систем искусственного интеллекта нового поколения по принципу "перехода количества в качество".

В области распределенного искусственного интеллекта основное внимание исследователей сосредоточилось на **многоагентных системах**, которые строятся из множества взаимодействующих агентов (зачастую представляющих собой полноценные интеллектуальные системы с символьным представлением знаний), совместно решающих поставленную задачу в распределенной среде. Для взаимодействия агентов различной природы разработаны специализированные языки обмена знаниями (KQML, KIF), а для обеспечения единого пространства знаний создаются **онтологии** — эксплицитные спецификации концептуализации предметной области, как правило в виде таксономии концептов и некоторого количества сопровождающих знаний (common knowledge) в виде системы аксиом-правил [1, 2].

При построении многоагентной системы разработчики сталкиваются со многими трудностями: необходимостью создания оригинальной архитектуры системы (методологии построения распределенных агентных систем пока слабо развиты, учитывая разнообразие существующих в рамках агентного подхода направлений), проектирования онтологии предметной области, выбора инструментария и транспортной среды для взаимодействия агентов, а также выбором программных средств для реализации интеллектуального наполнения агента, и, наконец, сопряжением всех указанных компонентов в единую функционирующую систему.

При реализации интеллектуального наполнения агентов в большинстве случаев приходится либо разрабатывать внутреннее представление знаний и машину вывода "с нуля" на традиционных языках программирования высокого уровня (C++, Java, Python, Lisp) или языках искусственного интеллекта (Пролог), либо использовать существующую оболочку для создания интеллектуальных систем. На сегодняшний день существует не так много доступных оболочек с достаточно развитым программным интерфейсом: наибольшего внимания заслуживают CLIPS, ее Java-аналог JESS и система программирования AMZI Prolog. Система JESS используется во многих проектах, однако ее интеграция в агентные системы и веб-

приложения представляет определенные сложности. Кроме того, CLIPS и JESS являются классическими продукционными системами прямого вывода, что несколько ограничивает возможности по управлению знаниями, а также требует представления знаний на языке класса OPS5, доступного только профессиональным инженерам по знаниям.

Следовательно, актуальной задачей является создание технологии и программного обеспечения, совмещающих в себе способы преодоления отмеченных проблем для определенного класса типовых задач. Можно выделить множество таких задач, связанных с распределенным накоплением и использованием знаний в компьютерных сетях, для решения которых классическая агентная архитектура представляется излишне общей. В то время как при построении агентных систем внимание как правило акцентируется на унифицированном внешнем представлении знаний с целью обмена между агентами различной природы, для некоторых задач представляется разумным базировать принципы распределения знаний на классической многоуровневой модели взаимодействия (в простейшем случае — модели клиент-сервер), в которой набор интеллектуальных систем или их составных компонентов взаимодействует и обменивается знаниями в некотором внутреннем представлении. Предлагаемая автором новая архитектура распределенной фреймовой иерархии является одной из возможных реализаций такого подхода.

В то время как для агентных систем онтология представляет собой некоторое внешнее по отношению к агентам соглашение, при таком унифицированном подходе сами компоненты системы неявно задают структуру предметной области. Заметим, однако, что при достаточно широком понимании понятия агента такая архитектура может быть отнесена к классу статических делиберативных коллаборативных агентов.

С другой стороны, немаловажную роль при построении реальных систем играет возможность взаимодействия с хранилищами структурированных данных (в первую очередь с реляционными и объектно-ориентированными базами данных и хранилищами слабоструктурированной информации на базе XML/HTML), а также интеграции в существующие информационные системы и веб-приложения. Представляет существенный интерес разработка такой модели представления знаний, которая обеспечивала бы естественную интеграцию в единую модель реляционных баз данных, объектно-ориентированных языком программирования и компонентных моделей (COM, JavaBeans, CORBA-объекты и др.). Такая модель при достаточном богатстве представления знаний позволила бы в некотором роде объединить в себе идеологию **активных и дедуктивных баз данных** (безусловно, с некоторыми модификациями и потерей производительности) с возможностью распределения и удаленного использования знаний.

2. Архитектура распределенной фреймовой иерархии

Получившая широкое распространение в рамках распределенного искусственного интеллекта агентная архитектура является весьма общей и для решения конкретных задач каждый раз должна уточняться специ-

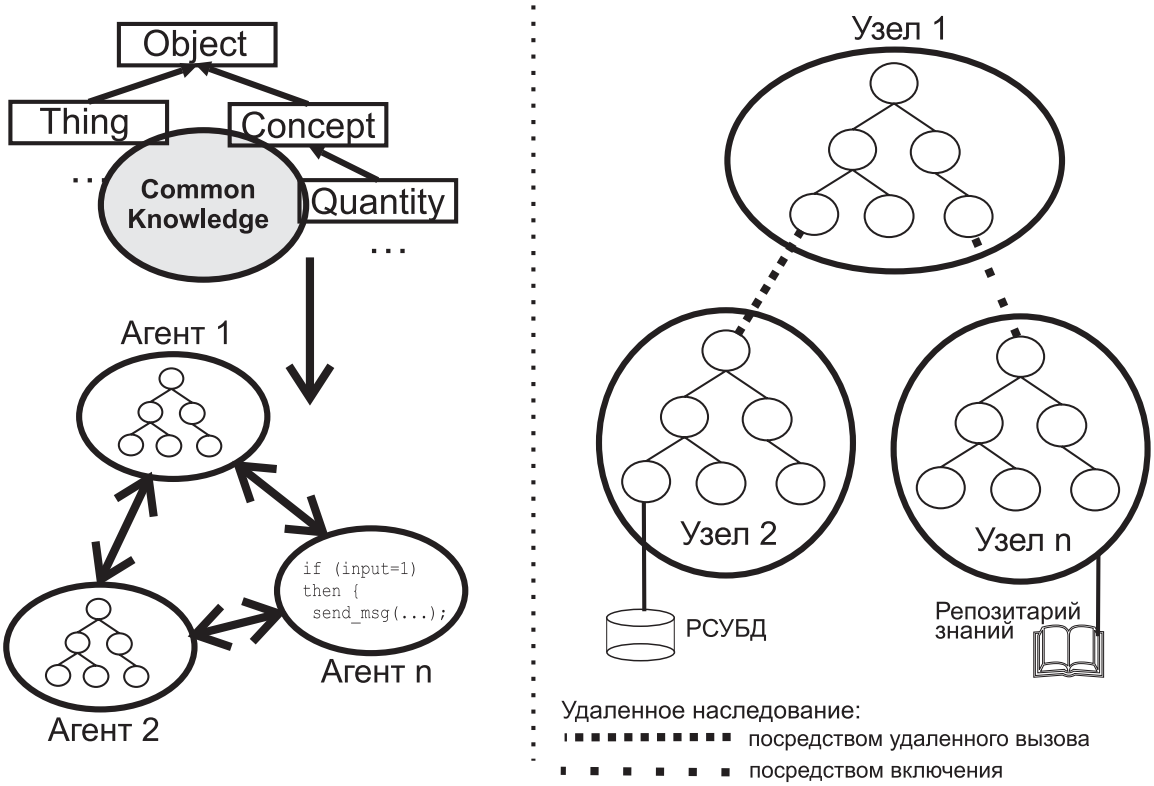


Рис. 1. Классическая агентная архитектура (слева) и архитектура распределенной фреймовой иерархии (справа)

фическими архитектурными решениями. При агентном подходе составные агенты в системе примерно равноправны и взаимодействуют друг с другом либо заранее определенным образом, либо динамически, выстраивая граф взаимодействия в процессе работы. Однако можно привести множество примеров, в основном связанных с распределенным накоплением и использованием знаний, когда взаимодействие сводится к использованию агента для получения ответа на запрос (**удаленная консультация**), либо к использованию знаний агента для их дальнейшего расширения и дополнения с целью получения более полной или специализированной интеллектуальной системы. Такое расширение весьма напоминает наследование в терминах объектно-ориентированного или фреймового подхода, что наводит на мысль об иерархической организации системы в виде **распределенной фреймовой иерархии**. Предлагаемая организация основана на использовании фреймового представления знаний для отображения на фреймовую иерархию множества концептов таксономической онтологии, при этом распределенная иерархия соответствует распределенному характеру расширяемой онтологической системы.

Архитектура **распределенной фреймовой иерархии** строится на основе иерархического комбинирования узлов интеллектуальной системы с фреймовой моделью представления знаний, так что вся распределенная система представляет собой единую иерархию фреймов, связанную отношением наследования, которое может распространяться между узлами системы. Каждый из узлов содержит в себе некоторый участок фреймовой иерархии, т.е. свои базу знаний, процессор вывода и рабочую память, обра-

зующие фрейм-мир, но может ссылаться (отношением наследования, агрегации или использования) на фреймы, расположенные на других узлах. Возможны различные модели комбинирования фреймовых субиерархий на распределенных узлах сети: все миры могут объединяться в единую фреймовую иерархию с общим родителем, некоторый фрейм может по очереди присоединяться (наследоваться) к различным иерархиям, образуя аналог модели доски объявлений, или же фрейм-миры могут взаимодействовать более сложным образом, представляя собой сеть с распределенными статическими знаниями. Все фрейм-миры, участвующие в некотором общем процессе вывода, образуют **кластер**. При этом взаимодействие фрейм-миров в рамках кластера сводится к следующим видам взаимодействия:

- Вычисление удаленного слота (статическое или мобильное)
- Удаленное наследование (статическое или мобильное)

Предложенная архитектура дополнена также средствами для обмена динамическими знаниями (в виде продукционных правил прямого и обратного вывода) по сети и использования **репозитария знаний** с загрузкой правил по мере необходимости в процессе вывода (on-demand rule loading).

Логический вывод в системе в целом происходит аналогично выводу в нераспределенной иерархии с учетом удаленного вызова методов на удаленных фреймах. Такой удаленный вызов позволяет осуществлять обмен значениями слотов и таким образом построить обмен знаниями на основе обмена статической информацией о состоянии задачи. Альтернативным подходом является обмен динамическими знаниями, реализуемый посредством включения.

2.1. Вычисление удаленного слота

В простейшем случае фрейм-миры могут объединяться в кластер без образования единой иерархии с общим родителем. Мы будем называть такие структуры **распределенной фреймовой коллецией**, оставляя название **иерархия** для таких способов комбинирования, при которых в иерархии имеется единственный общий родитель, так как именно такие структуры представляют особый интерес для распределенных онтологических описаний.

Для обеспечения распределенной функциональности синтаксис обращения к слоту фрейма расширяется возможностью указания имени удаленной иерархии, к которой относится требуемый фрейм, т.е. ссылка на фрейм может иметь вид `WorldID->FrameID`. При указании в некотором фрейм-мире имени удаленного слота для его вычисления может применяться два подхода:

- **Статическое вычисление путем удаленного вызова** состоит в том, что текущий узел сети производит запрос к удаленному узлу на вычисление значения слота, при этом процесс логического вывода также переносится на удаленный узел, пока значение запрашиваемого слота не будет возвращено назад[1].

[1] В процессе удаленного вывода может также происходить удаленное обращение к слотам фреймов

- **Мобильное вычисление путем включения**. При этом необходимые для вывода статические и динамические знания из удаленного мира (а возможно и весь удаленный фрейм-мир) переносятся на текущий узел сети, где принимают участие в процессе локального вывода для получения значения требуемого слота.

2.2. Удаленное наследование

Удаленное наследование является в некотором смысле частным случаем удаленного вычисления слота и основой для организации распределенной иерархии. Различают мобильное и статическое[2] удаленное наследование.

При **мобильном удаленном наследовании** в случае, когда требуется использовать ссылку на родительский фрейм, расположенный на удаленном узле, производится передача требуемого фрейма (либо всей включающей его иерархии) на локальный узел, где затем ссылка на него производится обычным образом. При этом логический вывод все время производится на одном и том же сетевом узле, а удаленные знания в виде дополнительных фрейм-миров подгружаются по сети по мере необходимости.

Аналогом рассмотренного выше мобильного наследования является **частичное мобильное наследование**, при котором по сети передаются не все знания с удаленного узла, а лишь те правила, которые необходимы для вычисления значения слота локального фрейма. В этом случае функциональность напоминает передачу знаний из репозитария, но организация удаленных знаний в виде правил для слота некоторого фрейма обеспечивает более прозрачную их поддержку.

Статическое удаленное наследование реализуется удаленным вызовом процедур, при этом логический вывод переходит с текущего узла сети на удаленный. Так как знания на удаленном узле должны применяться к значениям слотов текущего базового фрейма, то при удаленном вызове передается также ссылка на базовый фрейм, находящийся на текущем узле, чтобы значения его слотов могли быть определены посредством callback-вызовов.

2.3. Репозитарии знаний и загрузка правил по требованию

В некоторых случаях может оказаться полезным иметь возможность использовать в процессе логического вывода знания из некоторых удаленных источников, организованные некоторым отличным от фреймовой иерархии образом (например, в виде коллекции правил, подчиняющихся некоторой онтологии). Для продукционно-фреймовой системы с преимущественно обратным выводом может быть легко реализована возможность

исходного фрейм-мира (callback)

[2]Термин “статическое” в данном случае является весьма неудачным, так как в предыдущем разделе статическое наследование противопоставлялось динамическому и означало неизменность родителя в процессе вывода. В данном случае термин применяется как антоним мобильному наследованию, по аналогии с общепринятой терминологией для классификации агентов. Предполагается, что из контекста всегда можно определить, о какой “статичности” наследования идет речь.

загрузки правил обратного вывода из внешнего хранилища по мере необходимости, наподобие того, как реализуется поочередная загрузка правил в компонентной архитектуре [3]. Действительно, алгоритм обратного вывода требует лишь знания порядка применения правил до начала вывода (который во многих случаях не требует знания самих правил), а в процессе вывода правила используются поочередно до тех пор, пока не будет получено значение слота.

3. Семантика локального логического вывода

3.1. Назначение формального описания

Для более строгого описания механизма взаимодействия фреймми-ров в распределенной фреймовой иерархии, а также собственно процесса логического вывода в продукционно-фреймовой системе необходимо построить некоторую математическую модель такой системы и описать ее семантику. Применительно к исходному языку представления знаний предлагается двухуровневая модель формальной семантики, основанная на трансляции конструкций языка в теоретико-множественную модель представления знаний, на которой затем определяется семантика логического вывода. Необходимость и полезность такого построения обусловлена следующими соображениями:

- Формальная семантика позволяет математически строго доказать завершимость, а также другие полезные свойства логического вывода во фреймовой системе.
- На основе формального синтаксиса и семантики $\mathfrak{L}$ может непосредственно производиться разработка компилятора с ЯПЗ во внутреннее представление, что и было проделано автором при создании программной реализации инструментария JULIA. Более того, формальная семантика может лечь в основу реализации соответствующего программного прототипа на функциональном языке программирования (например, ML или Haskell).
- Формальное описание позволяет однозначно и строго описать синтаксис и семантику языка, исключая тем самым различные толкования как языковых конструкций, так и процесса логического вывода.

В качестве альтернативного подхода к определению семантики логического вывода может быть предложена декларативная интерпретация выражений языка с точки зрения логики предикатов, или расширенной объектной логики F-Logic [9]. Однако такая семантика как правило оказывается идеализированной, не отражающей реальные вычислительные аспекты, лежащие в основе программной реализации процессора вывода интеллектуальной системы.

3.2. Формализация представления знаний

Знания $\mathcal{K}$ в интеллектуальной системе можно разделить на статические $\mathcal{K}_s$, служащие для представления состояния предметной области в некоторый момент времени, и динамические $\mathcal{K}_d$, описывающие множество возможных правил перехода по графу состояний, вершинами которого являются элементы $\mathcal{K}_s$. В классической продукционной системе $\mathcal{K}_d$ описывается множеством продукционных правил, образующих базу знаний, а $\mathcal{K}_s$ представляет собой множество атрибутов, которое формируется и пополняется в процессе решения конкретной задачи.

В предлагаемой модели продукционно-фреймовой системы ситуация обстоит несколько сложнее. Предметная область при фреймовом представлении подвергается таксономической декомпозиции на множество представляемых фреймами концептов $\mathcal{F}$ при помощи отношения наследования :. Эта декомпозиция предметной области может быть отнесена к статическим знаниям, и в зависимости от сложности вводимой семантики она может меняться либо оставаться постоянной в процессе логического вывода. В большинстве систем множество фреймов описывается в базе знаний и остается более или менее постоянным[1] в процессе решения, однако следует отличать такие знания (назовем их **структурными** $\mathcal{K}_{ss}$) от множества атрибутов задачи, аналогичных рабочей памяти и представляемых значениями слотов всех фреймов (назовем соответствующее множество $\mathcal{K}_{sv}$, или состоянием рабочей памяти). База знаний (являющаяся плодом разработки инженеров по знаниям и задаваемая в некотором внешнем представлении на специализированном языке до начала работы системы) таким образом состоит из структурной $\mathcal{K}_{ss_1}$ и процедурной (или динамической) $\mathcal{K}_d$ составляющих; рабочая память состоит из множества значений слотов $\mathcal{K}_{sv}$ и, возможно, некоторой структурной составляющей $\mathcal{K}_{ss_2}$.

3.2.1. Представление статических знаний

Статические знания во фреймовой системе, представленные в основном составляющей $\mathcal{K}_{sv}$, образованы множеством значений слотов всех фреймов. Предположим, что слоты могут принимать значения из некоторой системы типов с множеством значений $\mathbb{T}$, являющимся полной решеткой. Эта система типов может быть представлена одним достаточно широким линейно упорядоченным доменом (например, множеством строк), либо быть дизъюнктной суммой некоторых полезных атомарных типов.

Для описания фреймовой структуры введем функцию состояния $\mathcal{W} : I \to \mathcal{F}$, которая отображает множество идентификаторов I в множество фреймов $\mathcal{F}$. Каждый фрейм $f \in \mathcal{F}$ представляет собой функцию $f : I_f \to \mathcal{S}$, отображающую множество идентификаторов слотов данного фрейма в множество слотов. Структура слотов может быть различной в зависимости от сложности рассматриваемой модели; в простейшем случае можно предположить $\mathcal{S} \equiv \mathrm{expr}(\mathbb{T})$, где под $\mathrm{expr}(\mathbb{T}) \subset \Lambda(\mathbb{T})$ подразумева-

[1] То, что в процессе вывода в модель добавляются новые фреймы, или же некоторый фрейм подвергается процессу классификации, является весьма частой ситуацией; однако основное дерево концептов предметной области остается постоянным, и таксономические связи в нем в процессе вывода не изменяются (за исключением псевдомножественного наследования).

ется множество λ-выражений над системой типов $\mathbb{T}$, включающих в себя, возможно, ссылки на другие слоты.

3.2.2. Представление динамических знаний

В более сложном случае имеет смысл рассматривать слот как кортеж, состоящий из нескольких компонент, представляющих собой текущее значение слота и значение по умолчанию, процедуры-демоны и процедуры запросы, ограничения и т.д. Таким образом можно будет совмещать в рамках одной функции $\mathcal{W}$ как собственно состояние системы, характеризуемое множеством значений слотов, так и правила, управляющие процессом смены состояний. Это позволит, в принципе, определять семантику с динамической модификацией множества правил и/или других характеристик логического вывода (например, стратегий выбора правил) в процессе вывода. Мы не будем останавливаться на такого рода семантиках, так как их строгое математическое изложение сопряжено с определенными трудностями, и может составить предмет отдельного исследования.

Мы будем рассматривать множество слотов вида $\mathcal{S} = \{\langle v, u, \{Q_i\}, \{D_j\}, \{C_k\}, \sqsubseteq_q, \sqsubseteq_d, \alpha\rangle\}$, где:

- $v \in \mathbb{T}$ — текущее значение слота или $\perp$ в случае, если значение не определено
- $u \in \mathbb{T}$ — значение слота по умолчанию
- $\{Q_i\}$ — множество присоединенных к слоту процедур-демонов. Каждая процедура может быть произвольным выражением из некоторого множества выражений $\mathbb{E}$, структура которого будет рассмотрена более подробно ниже в разделе 3.4.
- $\{D_j\}$ — множество процедур-демонов, срабатывающих при присваивании слоту некоторого значения. В соответствии с описанной ниже в разделе 3.4 семантикой, процедуры-демоны $D_j : \mathbb{E} \times \mathbb{W} \to \mathbb{W}$ применяются к функции состояния $\mathcal{W} \in \mathbb{W}$ в случае истинности некоторого выражения, и порождают тем самым новое состояние системы.
- $\{C_k\}$ — множество ограничений на значение слота, сформулированное в виде выражения-предиката, $C_k \in \mathbb{E}$.
- $\sqsubseteq_d, \sqsubseteq_q$ — линейные порядки на множествах $\{D_j\}$ и $\{Q_i\}$ соответственно, определяющие порядок применения соответствующих процедур в процессе вывода.
- $\alpha \in \{\mathbf{true}, \mathbf{false}\}$ — флаг, указывающий на участие слота в процессе рекуррентного обратного вывода и служащий для предотвращения бесконечного зацикливания.

Для доступа к элементам $s = \langle u, v, Q, D, C, \sqsubseteq_q, \sqsubseteq_d, \alpha\rangle$ будем использовать обозначения $s.value$, $s.defvalue$, $s.rules$, $s.daemons$, $s.constraints$, $s.rss_q$, $s.rss_d$ и $s.busy$ соответственно. Аналогично, будем обозначать $f.s = \langle f, s\rangle \in \mathbb{I}, f \in I, s \in I_f$ для именования фрейма, а для доступа к слоту писать $\mathcal{W}.f.s = \mathcal{W}(f, s)$.

Для дальнейших построений нам будет необходимо ввести функцию присваивания значения слоту $write : \mathbb{I} \times \mathbb{T} \to \mathbb{W} \to \mathbb{W}$, формирующую новое состояние, которое мы будем обозначать как $\mathcal{W}[f.s \leftarrow v] = write(\mathcal{W}, \langle f, s\rangle, v)$. Эта функция может быть определена следующим образом:

$$\mathcal{W}[f.s \leftarrow x] = \lambda f_1 \lambda s_1.(\langle f_1, s_1\rangle = \langle f, s\rangle \to \mathcal{W}(f, s)[1 \leftarrow x], \mathcal{W}(f_1, s_1)),$$

где $(b \to u, v)$ — операция условного вычисления, а через $s[n \leftarrow x]$ обозначена функция замены n-го компонента кортежа s на x: $(s[n \leftarrow x])_i = (i = n) \to x, s_i$. Аналогично будем определять операцию $\mathcal{W}[f.s.busy \leftarrow$ **true**/**false**$]$ для присвоения логического значения компоненте $s.busy$.

Определим также операцию разности между состояниями $|\cdot| : \mathbb{W} \times \mathbb{W} \to \mathcal{P}(\mathbb{I})$, возвращающую список идентификаторов слотов, значения которых отличаются в двух данных состояниях:

$$\forall \langle f, s\rangle \in \mathbb{I} \quad \langle f, s\rangle \in |\mathcal{W}_1 - \mathcal{W}_2| \iff \mathcal{W}_1(f, s) \neq \mathcal{W}_2(f, s) \tag{3.1}$$

3.3. Свойства состояний фреймовой модели

3.3.1. Множество и фактор-множество состояний

Функция состояния $\mathcal{W}$ описывает не только текущее состояние в процессе логического вывода, но также и множество правил, которое мы в данной работе будем полагать постоянным. Таким образом, возможно выделить функцию $\mathcal{W} : \mathbb{I} \to \mathbb{T} : \ \mathcal{W}(f, s) = \mathcal{W}(f, s).value \ \forall f \in I, s \in I_f$, которая будет характеризовать чисто статическую составляющую состояния системы, а также комплементарную к ней функцию, которая в свою очередь будет постоянной в процессе логического вывода.

Множество состояний системы $\mathfrak{W}$ можно представить себе в виде бесконечного графа, вершинами которого будут различные состояния $\mathcal{W} \in \mathfrak{W}$, а дуги будут задаваться правилами логического вывода. Бесконечность графа будет, в первую очередь, вызвана потенциальной бесконечностью (в теоретическом плане — континуальностью) множества значений ($\mathbb{T}$) каждого из слотов. Однако, говоря практически, число различимых состояний в каждой базе знаний будет конечно, так как во множестве посылок всех правил базы знаний содержится конечное число сравнений.

Для формализации этого понятия введем в рассмотрение отношение эквивалентности $\cong$, при котором $\mathcal{W}_1 \cong \mathcal{W}_2 \iff \mathcal{W}_1$ и $\mathcal{W}_2$ неразличимы с точки зрения базы знаний, т.е. для всего множества посылок C в левых частях правил базы знаний $\|c\|_{\mathcal{W}_1} = \|c\|_{\mathcal{W}_2} \ \forall c \in C$, где через $\|c\|_{\mathcal{W}}$ обозначено значение посылки c в состоянии $\mathcal{W}$. Тогда фактор-пространство $\mathfrak{W}_{\cong} = \mathfrak{W}/\cong$ пространства $\mathfrak{W}$ относительно отношения $\cong$ будет обладать следующим свойством:

Утверждение 3.1. *Фактор-пространство $\mathfrak{W}_{\cong}$ содержит конечное число элементов.*

Доказательство. Фактор-пространство $\mathfrak{W}_{\cong}$ можно строить следующим образом. Пусть $c \in C$ — посылка некоторого правила, которая может при-

нимать истинное или ложное значение на состоянии $\mathcal{W}$. Тогда она разбивает множество состояний на два подмножества: $\mathfrak{W}_t(c) = \{\mathcal{W} \in \mathfrak{W} | \, \|c\|_{\mathcal{W}} = \mathbf{true}\}$ и $\mathfrak{W}_f(c) = \{\mathcal{W} \in \mathfrak{W} | \, \|c\|_{\mathcal{W}} = \mathbf{false}\}$.

Пусть множество всех посылок C (конечное) имеет вид $C = \{c_1, \ldots, c_n\}$. Будем строить фактор-множество $\mathfrak{W}_{\cong}$ следующим образом: положим $\mathfrak{W}_{\cong}{}^{(1)} = \{\mathfrak{W}_f(c_1), \mathfrak{W}_t(c_1)\}$. Далее определим

$$\mathfrak{W}_{\cong}{}^{(k)} = \bigcup_{x \in \mathfrak{W}_{\cong}{}^{(k-1)}} [(x \cap \mathfrak{W}_f(c_k)) \cup (x \cap \mathfrak{W}_t(c_k))] \tag{3.2}$$

Тогда $\mathfrak{W}_{\cong} = \mathfrak{W}_{\cong}{}^{(n)}$ и будет искомым фактор-пространством. Действительно, для произвольного множества $A \in \mathfrak{W}_{\cong}$, если $\mathcal{W}_1, \mathcal{W}_2 \in A$, то по построению $\forall c \in C \; \|c\|_{\mathcal{W}_1} = \|c\|_{\mathcal{W}_2}$, т.е. $\mathcal{W}_1 \cong \mathcal{W}_2$. □

Очевидно, что каждый задаваемый правилом переход в пространстве $\mathfrak{W}$ индуцирует соответствующий переход в пространстве $\mathfrak{W}_{\cong}$. Таким образом, исследование процесса вывода в системе можно свести к исследованию конечного графа состояний с конечным множеством переходов.

3.3.2. Определение отношения порядка на множестве состояний

На множестве состояний $\mathfrak{W}$ может быть определено естественное отношение порядка $\sqsubseteq$ следующим образом:

$$\mathcal{W}_1 \sqsubseteq \mathcal{W}_2 \iff \forall \langle f, s \rangle \in \mathbb{I} \quad \mathcal{W}_1(f, s) \sqsubseteq \mathcal{W}_2(f, s) \tag{3.3}$$

где во втором случае отношение порядка связывает элементы множества системы типов $\mathbb{T}$. То, что такое определение действительно задает отношение порядка, можно легко убедиться стандартными для теории решеток рассуждениями.

Очевидно, что если не рассматривать метаправила, модифицирующие динамическую часть базы знаний в процессе вывода, то заданное таким образом отношение порядка также индуцирует отношение порядка на множестве $\mathbb{W}$. Кроме того, введенное формулой 3.3 отношение порядка требует постоянства множества $\mathbb{I}$ на протяжении вывода, т.е. имена всех фреймов и слотов должны быть известны заранее, до начала вывода. Эти ограничения безусловно сужают класс рассматриваемых динамических интеллектуальных систем, и изучение более богатой семантики представляет значительный интерес для дальнейших исследований[2].

3.3.3. Отношение и иерархия наследования

В системе фреймов вводится отношение наследования ":" между фреймами, определяющее иерархию понятий предметной области и со-

[2] Действительно, считается, что способность использовать мета-правила и "рассуждения о рассуждениях" является одной из существенных особенностей интеллектуальных систем и необходимым свойством для семиотического моделирования человеческой интеллектуальности. Исследование математических свойств мета-теорий и семантики мета-правил, таким образом, может пролить свет на такие вопросы, как самосознание, самовосприятие и др.

ответственно используемое для заимствования свойств (правил) от родительских фреймов. Различают одиночное и множественное наследование: в первом случае отношение : является функцией по правому аргументу, т.е. одному дочернему фрейму может соответствовать только один родитель[3]; во втором случае такое ограничение отсутствует. Кроме того, различают статическое наследование (используемое, например, в формализме F-логики [9]), при котором отношение наследования остается постоянным в процессе логического вывода, и динамическое наследование, когда конфигурация иерархии наследования в процессе вывода изменяется.

Будем рассматривать отношение наследования, индуцированное значением слотов `parent` дочерних фреймов, т.е. $F : G \iff \|F.parent\| = G$, где через $\|\cdot\|$ обозначена операция вычисления значения слота, которая будет уточняться далее в соответствии с определяемой семантикой. Более того, если допустить значение слота `parent` некоторого спискового типа, то возможно определить отношение множественного наследования как $F : G \iff \|F.parent\| \ni G$. Определенное таким образом отношение наследования будет динамическим, так как значение слота будет вычисляться в соответствии с правилами определенной ниже семантики, т.е. будет возможно определение родителя посредством продукционных правил либо операции спецификации.

3.4. Операционная семантика логического вывода

Для определения семантики процесса вывода в системе воспользуемся, аналогично [6], отображением $\mathcal{E} : \mathbb{E} \times \mathbb{W} \times \mathbb{C} \to \mathbb{T} \times \mathbb{W}$, которое вычисляет значение произвольного выражения $E \in \mathbb{E}$ в некотором состоянии $\mathcal{W}$ и контексте $\mathbb{C}$, возвращая полученное значение $v \in \mathbb{T}$ и новое состояние $\mathcal{W}'$. Будем также использовать более удобное обозначение $v = \|E\|^C_{\mathcal{W}\to\mathcal{W}'} \iff \mathcal{E}(E, \mathcal{W}, C) = (v, \mathcal{W}')$.

Семантика логического вывода подразумевает, что при вычислении значения выражения может быть инициирован вывод значений тех слотов, для которых значение не известно заранее и не было получено ранее в процессе вывода. Определим также отношение $\mathcal{E}' : \mathbb{E}\times\mathbb{W}\times\mathbb{C} \to \mathbb{T}\times\mathbb{W}$, аналогичное $\mathcal{E}$, но не инициирующее процесс вывода. Для этого отображения будем использовать обозначение $v = \|E\|'^C_{\mathcal{W}\to\mathcal{W}'} \iff \mathcal{E}'(E, \mathcal{W}, C) = (v, \mathcal{W}')$. В дальнейшем, если это не оговорено отдельно, все приведенные для $\mathcal{E}$ результаты будут аналогичным образом формулироваться для $\mathcal{E}'$.

Понятие контекста вычисления будет необходимо для описания семантики наследования, чтобы корректно применять правила для фрейма-родителя к значениям в дочерних фреймах. В нашем случае достаточно будет положить $\mathbb{C} = I$, хотя в более сложных моделях в понятие контекста может потребоваться включить и другие конструкции. Фрейм, передаваемый через контекст, будем называть *базовым фреймом*, и этот фрейм будет использоваться при вычислении выражения вместо зарезервированного идентификатора *this*. Так как в большинстве случаев при описании семантики вычисления выражений значение базового фрейма передается в функции вычисления подвыражений без изменения, то мы будем верхний

[3] Следует заметить, что в этом случае иерархия наследования представляет собой *дерево*.

индекс C опускать. Таким образом, если контекст вычисления не указан в некотором равенстве, то имеется в виду, что в обоих частях равенства фигурирует один и тот же контекст.

3.4.1. Синтаксис множества выражений

Для корректного определения семантики необходимо сначала задать формальный синтаксис множества выражений $\mathbb{E}$. Для формального задания синтаксиса можно воспользоваться нотацией Бэкуса-Наура (БНФ) либо описанием грамматики. Мы не будем подробно останавливаться на формальном описании синтаксиса, ограничившись описанием структуры выражений $E \in \mathbb{E}$.

- Константа из множества типов $\mathbb{T}$.
- Ссылка на слот вида $f.s,\ f \in I \cup \{this\},\ s \in I_f$.
- Арифметическая или логическая операция вида $E_1 \circledast E_2$, где $\circledast \in \{+, -, *, /, \mathbf{or}, \mathbf{and}, \to\}$, или отрицание $\mathbf{not}\, E$.
- Операция вызова функции-оракула $\mathcal{A}(d),\ d \in \mathbb{D}$

Строго говоря, арифметические и логические операции в множестве выражений $\mathbb{E}$ могут вводиться в соответствии с операциями, определенными на множестве типов. Приведенная в следущем разделе семантика вычисления выражений сопоставляет каждой операции в синтаксическом множестве $\mathbb{E}$ некоторый денотат, основанный на определении операции в множестве $\mathbb{T}$.

3.4.2. Интерпретация констант, арифметических и логических выражений

Весьма очевидно, что интерпретация констант не зависит от состояния $\mathcal{W}$, и в качестве денотата константы возвращается само ее значение, т.е. $\forall x \in \mathbb{T}\ \|x\|_{\mathcal{W}\to\mathcal{W}} = x$.

Значением выражения $E = E_1 \circledast E_2$, являющегося арифметической операцией над парой выражений E_1 и E_2, будет соответствующая арифметическая операция $\circledast_{\mathbb{T}}$ в множестве типов $\mathbb{T}$. Таким образом:

$$\|E_1 \circledast E_2\|_{\mathcal{W}_1\to\mathcal{W}_2} = \begin{cases} \bot, \|E_1\|_{\mathcal{W}_1\to\mathcal{W}_2} = \bot \\ \|E_1\|_{\mathcal{W}_1\to\mathcal{W}'} \circledast_{\mathbb{T}} \|E_2\|_{\mathcal{W}'\to\mathcal{W}_2} \end{cases}$$

В данном описании учитывается т.н. *укороченное* вычисление выражения, при котором в случае, если первое подвыражение возвращает $\bot$, значение второго не вычисляется. Возможен также более простой подход, когда оба подвыражения вычисляются в любом случае:

$$\|E_1 \circledast E_2\|_{\mathcal{W}_1\to\mathcal{W}_2} = \|E_1\|_{\mathcal{W}_1\to\mathcal{W}'} \circledast_{\mathbb{T}} \|E_2\|_{\mathcal{W}'\to\mathcal{W}_2}$$

В случае с вычислением логических выражений вычисление производится аналогично, так как в динамической системе типов явное различие

между логическим и другими типами не проводится. Как правило, при вычислении логических выражений *всегда* используется укороченная схема. Аналогично по укороченной схеме вычисляется условное выражение $\to$:

$$\|E \to E_1, E_2\|_{\mathcal{W}_1 \to \mathcal{W}_2} = \begin{cases} \|E_1\|_{\mathcal{W}' \to \mathcal{W}_2}, & \text{isTrue}(\|E\|_{\mathcal{W}_1 \to \mathcal{W}'}) \\ \|E_2\|_{\mathcal{W}' \to \mathcal{W}_2}, & \neg\,\text{isTrue}(\|E\|_{\mathcal{W}_1 \to \mathcal{W}'}) \end{cases}$$

3.4.3. Получение значений слотов и обратный вывод

В случае, если выражение E имеет вид $f.s$, для получения его значения требуется извлечь значение соответствующего слота из состояния $\mathcal{W}$, в случае необходимости применив обратный вывод. Интерпретация при помощи функции $\mathcal{E}'$ не инициирует обратный вывод, а лишь возвращает значение слота или $\perp$ (состояние в этом случае не изменяется):

$$\|f.s\|'_{\mathcal{W} \to \mathcal{W}} = \mathcal{W}(f, s).value$$

Применение обратного вывода для вычисления значения слота s состоит в последовательном (в соответствии с линейным порядком $\sqsubseteq_q$) применении правил из $s.rules$. Применение правила состоит в вычислении соответствующего этому правилу условного или безусловного выражения в соответствии со следующей таблицей.

Продукционное правило	Выражение
`IF` E `THEN` $f.s = F$	$E \to F$
`SET` $f.s = E$	E
`ASK` q $f.s$	$\mathcal{A}(q)$

Пусть $\langle Q, \sqsubseteq \rangle$ — (конечное) линейно упорядоченное множество. Обозначим $\hat{Q} = Q \setminus \{\inf Q\}$. Введем функцию $\mu_{\mathcal{W} \to \mathcal{W}'}(Q)$ последовательного вычисления упорядоченного семейства выражений Q в начальном состоянии $\mathcal{W}$:

$$\mu_{\mathcal{W} \to \mathcal{W}'}(Q) = \begin{cases} \|\inf Q\|_{\mathcal{W} \to \mathcal{W}'}, & \|\inf Q\|_{\mathcal{W} \to \mathcal{W}'} \neq \perp \\ \mu_{\mathcal{W}'' \to \mathcal{W}'}(\hat{Q}), & \|\inf Q\|_{\mathcal{W} \to \mathcal{W}''} = \perp \\ \perp, & \mathcal{W}' = \mathcal{W}, Q = \varnothing \end{cases} \tag{3.4}$$

Утверждение 3.2. *Данное рекурсивное определение корректно определяет функцию μ, которая определена для любого начального состояния и конечного множества Q.*

Идея доказательства. Для доказательства существования функции $\mu : \mathbb{Q} \times \mathbb{W} \to \mathbb{T} \times \mathbb{W}$ введем в рассмотрение некоторую функцию $\mu' : \mathbb{Q} \times \mathbb{W} \to \mathbb{Q} \times \mathbb{W}$, такую, что $\mu_{\mathcal{W} \to \mathcal{W}'}(Q) = \|\inf Q'\|_{\mathcal{W}' \to \mathcal{W}'}$, где $Q' = \mu'_{\mathcal{W} \to \mathcal{W}'}(Q)$ (считая, что $\|\perp\| = \perp$). Определим μ' как неподвижную точку некоторой функции $\mathcal{M} : [\mathbb{Q} \times \mathbb{W} \to \mathbb{Q} \times \mathbb{W}] \to [\mathbb{Q} \times \mathbb{W} \to \mathbb{Q} \times \mathbb{W}]$ более высокого порядка:

$$\mathcal{M}(f) = \lambda f.\lambda(Q, W).\begin{cases} \langle Q, \mathcal{W}' \rangle, & \|\inf Q\|_{\mathcal{W} \to \mathcal{W}'} \neq \perp \\ f_{\mathcal{W}'' \to \mathcal{W}'}(\hat{Q}), & \|\inf Q\|_{\mathcal{W} \to \mathcal{W}''} = \perp \\ \langle \varnothing, \mathcal{W} \rangle, & Q = \varnothing \end{cases} \tag{3.5}$$

Следует отметить, что $\mathcal{P}(Q)$ является решеткой относительно операции включения, а на множестве $\mathbb{W}$ также индуцируется отношение порядка (см. раздел 3.3.2) таким образом, что оно является решеткой, на котором функция $\|\cdot\|_{\cdot\to\cdot}$ будет монотонной (см. утверждение 3.8). В соответствии с этим у отображения $\mathcal{M}$ будет существовать неподвижная точка $\mu' = \operatorname{fix}\mathcal{M}$, которая и будет определять искомую функцию μ'.

При этом сама функция μ' будет монотонной на решетке, порожденной $\mathbb{Q}\times\mathbb{W}$ с рассмотренными выше отношениями порядка, и по теореме Тарского о неподвижной точке [8] она будет определять конечное состояние $\langle Q', \mathcal{W}'\rangle = \bigsqcup_{n=1}^{\infty}\mu'^n(Q,\mathcal{W})$. Следует отметить, что ввиду конечности исходного множества Q и строгого характера монотонности данное равенство будет выполняться для некоторого конечного N, т.е.

$$\forall Q\in\mathbb{Q}\;\exists N\le\#Q:\;\langle Q',\mathcal{W}'\rangle = \bigsqcup_{n=1}^{N}\mu'^n(Q,\mathcal{W})$$

□

При помощи определенной таким образом функции μ можно легко определить функцию $\|\cdot\|$ для слота с учетом обратного вывода:

$$\|f.s\|_{\mathcal{W}\to\mathcal{W}'} = \begin{cases} \mathcal{W}(f,s).value, \mathcal{W}(f,s).value \neq \bot \text{ (при этом } \mathcal{W}'=\mathcal{W}) \\ \bot, \mathcal{W}(f.s).busy = \mathbf{true} \\ x = \mu_{\mathcal{W}[f.s.busy\leftarrow\mathbf{true}]\to\mathcal{W}''}(\langle\mathcal{W}(f,s).rules, \mathcal{W}(f,s).\sqsubseteq_q\rangle), \\ \qquad \mathcal{W}' = \mathcal{W}''[f.s\leftarrow x, f.s.busy\leftarrow\mathbf{false}] \quad (x\neq\bot) \\ y = \mu_{\mathcal{W}'''\to\mathcal{W}''''}(\{p.s | p\in\|f.parent\|_{\mathcal{W}''\to\mathcal{W}'''}\}), \\ \qquad \mathcal{W}' = \mathcal{W}''''[f.s\leftarrow y, f.s.busy\leftarrow\mathbf{false}] \quad (y\neq\bot) \\ \bot, \text{ в противном случае} \end{cases} \tag{3.6}$$

Следует отметить, что приведенная здесь семантика не учитывает возможность применения метаправил, осуществляющих смену стратегии выбора правил в процессе вывода, а также динамическое изменение множества самих правил. Рассмотрение такой возможности привело бы нас к существенным затруднениям, так как при доказательстве утверждения 3.2 было использовано постоянство и конечность множества Q и отношения порядка на нем. Также, вычисление множества родительских фреймов производилось до осуществления процесса вывода в первом родительском фрейме, что не учитывает возможность изменения этого множества в процессе дальнейшего вывода. Рассмотрение более богатой семантики и ее математических свойств безусловно представляет существенный интерес, однако выходит за рамки данной работы (см. также сноску на стр. 13).

Следует также для полноты картины отдельно определить вычисление значения выражения $this.s$ следующим образом:

$$\|this.s\|^f_{\mathcal{W}\to\mathcal{W}'} = \|f.s\|^f_{\mathcal{W}\to\mathcal{W}'}$$

а также выражения с нулевым контекстом:

$$\|f.s\|^{\bot}_{\mathcal{W}\to\mathcal{W}'} = \|f.s\|^f_{\mathcal{W}\to\mathcal{W}'}$$

Для описания возможности динамического получения значений извне в ходе обратного логического вывода можно воспользоваться двумя подходами: введением множества пар "вопрос-ответ" в контекст вычислений [6], либо **функцией-оракулом** (oracle function) [12]. В последнем случае мы определяем некоторую внешнюю функцию взаимодействия со средой $\mathcal{A} : \mathbb{D} \to \mathbb{T}$, определенную на некотором множестве дескрипторов $\mathbb{D}$, и соответствующий класс выражений, для которого операция $\|\cdot\|$ вводится следующим образом:

$$\|\mathcal{A}(d)\|_{\mathcal{W}\to\mathcal{W}} = \mathcal{A}(d) \tag{3.7}$$

Использование такого подхода более естественно выражает характер диалога, управляемого процессом вывода (вызовы функции-оракула происходят по мере необходимости в том порядке, в котором вопросы задаются пользователю), однако затрудняет изучение семантики, делая ее недетерминированной (т.е. результат зависит от неформализованного в модели поведения внешних функций). Введение множества ответов пользователя в контекст вывода приводит к детерминированной семантике, которая не отражает динамику задаваемых вопросов (в этом случае приходится фиксировать все множество "ответов" заранее). Заметим, что разновидностью такого подхода является полный отказ от операции задания вопроса и замена ее первоначальным заполнением значениями всех известных слотов.

3.4.4. Прямой вывод

Для формализации прямого вывода существует несколько подходов. В классическом алгоритме прямого вывода [10] на каждом шаге формируется конфликтное множество применимых правил, из которых затем применяется одно, что приводит к изменению состояния. В случае монотонного вывода функция применения одного правила будет монотонной, что приводит к весьма естественному описанию семантики с помощью неподвижной точки на основании теоремы Тарского. Похожий подход применительно к логическому программированию описан в [11]. Однако такой "классический" подход подразумевает использование всего множества правил на каждом шаге вывода и не учитывает привязанность правил к слотам фреймов.

Другой крайностью являлась бы попытка математической формализации используемых алгоритмов прямого вывода семейства Rete [7]. Такая формализация по всей видимости оказалась бы излишне сложной, а также слишком привязанной к конкретному алгоритму логического вывода.

Мы используем некоторый промежуточный подход, в котором правила прямого вывода будут присоединены к слотам фреймов, как это было описано в разделе 3.2.2. Таким образом, при присваивании слоту значения будут срабатывать только правила, связанные с этим слотом, т.е. в левой части которых фигурирует значение этого слота. Применению правил будет соответствовать изменение состояния в соответствии с некоторой функцией. Ниже мы покажем, что такой подход, при правильном построении сети демонов, в некотором смысле является эквивалентным классическому, но при этом он непосредственно соответствует возможному алгоритму реализации прямого вывода.

Каждое присоединенное правило-демон имеет вид $D = \langle E, A \rangle$, где $E \in \mathbb{E}$ — условная часть правила, представляющая собой произвольное выражение, $A : \mathbb{W} \to \mathbb{W}$ — некоторая функция изменения состояния, которая, как правило, будет представлять собой функцию изменения значения слота либо композицию таких функций. В некоторых случаях будет полезно также параметризовать эти функции текущим контекстом вычисления $C \in \mathbb{C}$, тогда $A : \mathbb{C} \to \mathbb{W} \to \mathbb{W}$, или $A(C) : \mathbb{W} \to \mathbb{W}$. Имеет смысл рассматривать композиции следующих элементарных функций:

$$\begin{array}{lcl} [f.s \leftarrow v](C, \mathcal{W}) & = & \mathcal{W}[f.s \leftarrow v] \\ [this.s \leftarrow v](C, \mathcal{W}) & = & \mathcal{W}[C.s \leftarrow v] \end{array} \tag{3.8}$$

Для каждой функции A введем в рассмотрение множество модифицируемых слотов $\Delta(A)$. Очевидно, что $\Delta([f.s \leftarrow v]) = \{f.s\}$, и $\Delta(A_1 \circ \ldots \circ A_n) = \Delta(\{A_1, \ldots, A_n\}) = \cup_{i=1}^{n} \Delta(A_i)$.

Для описания процесса прямого вывода нам понадобится понятие функции активации $\alpha : \mathbb{I} \to \{free, todo, applied\}$, где

$$\alpha(f, s) = \begin{cases} free, & \text{если слот не принимает участие в прямом выводе} \\ todo, & \text{если демоны для слота еще нужно применить} \\ applied, & \text{если демоны для слота уже были применены} \end{cases} \tag{3.9}$$

Множество функций активации обозначим через $\mathcal{A}$.

На множестве $\{free, todo, applied\}$ определим отношение порядка следующим образом:

$$free \sqsubseteq todo \sqsubseteq applied \tag{3.10}$$

Тогда

$$\alpha_1 \sqsubseteq \alpha_2 \iff \forall \langle f, s \rangle \in \mathbb{I} \quad \alpha_1(f, s) \sqsubseteq \alpha_2(f, s) \tag{3.11}$$

Для формализации прямого вывода введем функцию $\Phi^C : \mathcal{A} \times \mathbb{W} \to \mathcal{A} \times \mathbb{W}$, которая по функции активации и состоянию применяет правила-демоны для этого состояния и помечает в функции активации те слоты, которые были изменены в процессе обработки, а также одноименные слоты для родителей данного фрейма.

Обозначим через $\alpha[f.s] = \lambda\langle f_x, s_x \rangle.(\langle f, s \rangle = \langle f_x, s_x \rangle) \to (todo \vee \alpha(f, s)); \alpha(f_x, s_x)$. Индуктивно можно определить $\alpha[\{f_i.s_i\}_{i=1}^{n}] = \alpha[f_1.s_1] \ldots [f_n.s_n]$.

Для применения множества правил $S \subset \mathbb{D}$ к состоянию определим функцию $\phi_{\sqsubseteq}^{C} : \mathbb{D} \times \mathcal{A} \times \mathbb{W} \to \mathbb{D} \times \mathcal{A} \times \mathbb{W}$ следующим образом[4]:

$$\phi^C(S, \alpha, \mathcal{W}) = \begin{cases} \langle \varnothing, \alpha, \mathcal{W} \rangle, \text{ если } S = \varnothing \\ \langle \hat{S}, \alpha[\inf S], \text{isTrue}\, \|E\|'_{\mathcal{W} \to \mathcal{W}} \to A(C)(\mathcal{W}); \mathcal{W} \rangle \end{cases} \tag{3.12}$$

Лемма 3.1. $\forall \alpha \in \mathcal{A} \quad \alpha[f.s] \sqsupseteq \alpha$

Лемма 3.2. $\forall \langle E, A \rangle \in \mathbb{D}, \forall \mathcal{W} \quad A(C)(\mathcal{W}) \sqsupseteq \mathcal{W}$

Следствие 3.3. *Функция ϕ^C является монотонной относительно решетки $\mathbb{D} \times \mathcal{A} \times \mathbb{W}$ с отношением порядка, порожденным естественным отношением $\subseteq$ на $\mathbb{D}$, описанным формулой 3.11 на $\mathcal{A}$ и введенным в разделе 3.3.2 отношением порядка на $\mathbb{W}$.*

[4] В данном равенстве операции inf и $\hat{\cdot}$ понимаются в смысле отношения $\sqsubseteq$.

Утверждение 3.4. *Существует функция* $\varphi : \mathbb{C}\times\mathbb{D}\times\mathcal{A}\times\mathbb{W} \to \mathcal{A}\times\mathbb{W}$, *которая по начальному состоянию* $\mathcal{W}$ *и функции активации* α *применяет множество правил* $\{D_i\}_{i=1}^N$ *в контексте* C, *возвращая конечное состояние* $\mathcal{W}'$ *и функцию активации* α', *причем* $\mathcal{W} \sqsubseteq \mathcal{W}'$, $\alpha \sqsubseteq \alpha'$, *т.е.* φ *как функция* α *и* $\mathcal{W}$ *будет монотонной.*

Доказательство. Из следствия 3.3 вытекает, что по теореме Тарского [8] функция ϕ имеет неподвижную точку, которая может быть получена как $\langle S', \alpha', \mathcal{W}'\rangle = \bigsqcup_{i=1}^{\infty} \phi^i(S, \alpha, \mathcal{W})$. Легко видно, что ввиду строгой монотонности по первому аргументу данное равенство принимает вид

$$\bigsqcup_{i=1}^{N} \phi^i(S, \alpha, \mathcal{W}) = \langle \varnothing, \alpha', \mathcal{W}'\rangle \tag{3.13}$$

Можно таким образом определить функцию φ, положив $\varphi^C(S, \alpha, \mathcal{W}) = \langle \alpha', \mathcal{W}'\rangle$. Условия на упорядоченность начального и конечного состояний следует из монотонности ϕ. □

Определим функцию $\mathrm{Anc}_{\mathcal{W}\to\mathcal{W}'}(f)$, возвращающую множество родительских фреймов для f (при этом динамическое вычисление родителя требует предусмотреть возможность изменения состояния $\mathcal{W}$ в процессе вычисления):

$$\mathrm{Anc}_{\mathcal{W}\to\mathcal{W}'}(f) = \begin{cases} \varnothing, \text{ если } \|f.parent\|_{\mathcal{W}\to\mathcal{W}'} = \bot \\ \|f.parent\|_{\mathcal{W}\to\mathcal{W}_1} \cup \left(\bigcup\limits_{x\in\|f.parent\|_{\mathcal{W}\to\mathcal{W}''}} \mathrm{Anc}_{\mathcal{W}_i\to\mathcal{W}_{i+1}}(x) \right), \\ \qquad \text{где } \mathcal{W}' = \mathcal{W}_N \end{cases} \tag{3.14}$$

С учетом введенных обозначений, получим

$$\Phi^C(\alpha, \mathcal{W}) = \begin{cases} \langle \alpha, \mathcal{W}\rangle, & \text{если } \forall \langle f, s\rangle \in \mathbb{I} \quad \alpha(f, s) \neq todo \\ \langle \alpha', \mathcal{W}'\rangle, & \text{где } \langle \alpha'', \mathcal{W}''\rangle = \varphi^C_{\mathcal{W}(f,s).\sqsubseteq_d}(\mathcal{W}(f, s).daemons, \alpha, \mathcal{W}), \\ & \langle f, s\rangle = \min\{\langle f_x, s_x\rangle | \alpha(f_x, s_x) = todo\} \\ \alpha''' = & \alpha'' \left[\Delta(\mathcal{W}(f, s).daemons) \cup \left(\bigcup\limits_{p\in\mathrm{Anc}_{\mathcal{W}''\to\mathcal{W}'}(f)} \{p.s\} \right) \right] \\ & \alpha' = \alpha'''[f.s \leftarrow applied] \end{cases} \tag{3.15}$$

Функция Φ находит первый (в смысле лексикографической упорядоченности) слот $f.s$, помеченный в α как $todo$, использует φ для применения всех связанных с ним правил-демонов с соответствующим отношением порядка, помечает все измененные слоты, а также одноименные слоты всех родителей f, как кандидатов на дальнейшую обработку, и наконец устанавливает значение $\alpha(f, s)$ как $applied$, что исключает повторное применение демонов для этого слота.

Утверждение 3.5. *Функция* Φ *является монотонной в смысле естественного индуцированного отношения порядка на* $\mathcal{A}\times\mathbb{W}$ *для любого контекста* $C \in \mathbb{C}$.

Доказательство. следует из лемм 3.1 и 3.2 и утверждения 3.4. □

Утверждение 3.6. *Существует функция прямого вывода для слота* $f.s$ $\Phi : \mathbb{I} \to \mathcal{W} \to \mathcal{W}$, *которая для состояния* $\mathcal{W}$ *возвращает конечное состояние* $\mathcal{W}' = \Phi_{f.s}(\mathcal{W})$, *полученное применением всех ассоциированных со слотом правил прямого вывода.*

Доказательство. Из монотонности функции Φ следует существование решетки неподвижных точек, и, следовательно, $\forall \alpha \in \mathcal{A}, \mathcal{W} \in \mathbb{W} \quad \exists \langle \alpha', \mathcal{W}' \rangle = \bigsqcup_{n=1}^{\infty} \Phi^n(\alpha, \mathcal{W})$. Положим $\Phi_{\alpha}^{C}(\mathcal{W}) = \mathcal{W}'$. Тогда можно определить функцию прямого вывода для слота $f.s$ следующим образом:

$$\Phi_{f.s}(\mathcal{W}) = \Phi^{f}_{\alpha_0[f.s \leftarrow todo]}(\mathcal{W}) \tag{3.16}$$

где $\alpha_0(f, s) = free \; \forall \, \langle f, s \rangle \in \mathbb{I}$. □

Функцию $\Phi_{f.s}(\mathcal{W})$ будем называть **функцией направленного прямого вывода**, подчеркивая, что вывод производится начиная от слота $f.s$. Можно аналогичным образом определить функцию **исчерпывающего прямого вывода** $\Phi(\mathcal{W})$, которая будет рассматривать все слоты и все правила на предмет возможного их применения. Для ее определения мы можем вообще отказаться от функции активации, а в качестве одношаговой функции вывода рассматривать $\phi' : \mathcal{W} \to \mathcal{W}$, получая $\Phi(\mathcal{W}) = \operatorname{fix} \phi(\mathcal{W})$. Следующее утверждение устанавливает связь между введенными понятиями:

Утверждение 3.7. *Процесс направленного прямого вывода является полным относительно процесса исчерпывающего прямого вывода, т.е. если* $\mathcal{W}_0$ *— состояние, для которого* $\Phi(\mathcal{W}_0) = \mathcal{W}_0$ *(такое состояние будем называть* **устойчивым относительно прямого вывода***), то существует такая стратегия разрешения конфликтов исчерпывающего вывода* $\sqsubseteq$, *что* $\Phi_{f.s}(\mathcal{W}_0[f.s \leftarrow v]) = \Phi^{\sqsubseteq}(\mathcal{W}_0[f.s \leftarrow v])$.

Идея доказательства. Данное утверждение подразумевает корректное построение сети демонов, т.е. такое, при котором все правила, которые зависят от изменения значения слота $f.s$, содержатся среди демонов этого слота, т.е множество правил, применяемых исчерпывающим и направленным выводом на первом шаге будут совпадать. Более строго можно показать по индукции, что если $|\mathcal{W} - \mathcal{W}_0| = \{f.s\}$ и $\mathcal{W}_0 = \operatorname{fix} \Phi$, то можно построить такую стратегию выбора правил $\sqsubseteq$, что $\Phi_{f.s}(\mathcal{W}) = \Phi^{\sqsubseteq}(\mathcal{W})$. Откуда и следует истинность доказываемого утверждения. □

3.4.5. Проверка ограничений

В предложенной модели с каждым слотом ассоциировано множество ограничений $\{C_i\}_{i=1}^{N}$, каждое из которых представляет собой выражение, которое должно быть истинно для данного слота. В этих выражения используется ссылка на текущий фрейм $this$, позволяющая организовать наследование ограничений для слота рекурсивным вызовом ограничений родительского слота. Опишем такую семантику ограничений, при которой

текущее состояние $\mathcal{W}$ не изменяется. Более совершенная семантика, учитывающая возможность обратного вывода необходимых слотов в ограничениях, нами рассматриваться и реализовываться не будет.

Для проверки ограничений введем функцию check : $\mathbb{C} \to \mathcal{W} \to \mathbb{I} \to \{true, false\}$, определенную как

$$\text{check}^{C}_{f.s}(\mathcal{W}) = \text{isTrue}(\|C_1\|'^{C}_{\mathcal{W}\to\mathcal{W}} \textbf{ or } \ldots \textbf{ or } \|C_N\|'^{C}_{\mathcal{W}\to\mathcal{W}}) \tag{3.17}$$

где $\mathcal{W}(f,s).constraints = \{C_i\}_{i=1}^{N}$.

Для проверки ограничений с учетом наследования, подобно тому, как это было сделано выше, введем функцию $\text{Anc}'_{\mathcal{W}}(f)$, возвращающую множество тех родительских фреймов для f, значения которых уже были получены ранее или заданы явно:

$$\text{Anc}_{\mathcal{W}}(f) = \begin{cases} \varnothing, \text{ если } \|f.parent\|'_{\mathcal{W}\to\mathcal{W}} = \bot \\ \|f.parent\|'_{\mathcal{W}\to\mathcal{W}} \cup \left(\bigcup\limits_{x\in\|f.parent\|_{\mathcal{W}\to\mathcal{W}}} \text{Anc}'_{\mathcal{W}}(x) \right) \end{cases} \tag{3.18}$$

Тогда функция полной проверки ограничений

$$\text{ensure}_{f.s}(\mathcal{W}) = \text{check}^{f}_{f.s}(\mathcal{W}) \textbf{ or } \left(\textbf{or}\limits_{p\in\text{Anc}'_{\mathcal{W}}(f)} \text{check}^{f}_{p.s}(\mathcal{W}) \right) \tag{3.19}$$

Для определения семантики проверки ограничений в процессе вывода переопределим функцию изменения значения слота следующим образом:

$$write(\mathcal{W}, \langle f, s\rangle, v) = \text{isTrue}\,\text{ensure}_{f.s}(\mathcal{W}[f.s \leftarrow v]) \to \mathcal{W}[f.s \leftarrow v]; \mathcal{W} \tag{3.20}$$

3.4.6. Комбинированный вывод

Под комбинированным логическим выводом подразумевается такой механизм вывода, при котором поочередно применяются прямой и обратный вывод. Такой подход сочетает в себе достоинства двух основных методов вывода, а кроме того является более универсальным, так как оставляет возможность применять как прямой и обратный вывод в чистом виде, так и их комбинацию. В этом случае обратный вывод определяет основной маршрут движения по дереву решений, а также отвечает за получение необходимой информации от пользователя в процессе вывода, в то время как прямой вывод позволяет “расширить” множество полученных данных за счет того, что может быть получено из уже имеющегося множества без дополнительной информации.

Нами будет рассмотрен подход, при котором после каждого шага обратного вывода, т.е как только получено значение некоторого слота, будет применяться направленный исчерпывающий прямой вывод. Для описа-

ния такой семантики, выражение 3.6 следует записать следующим образом:

$$\|f.s\|_{\mathcal{W}\to\mathcal{W}'} = \begin{cases} \mathcal{W}(f,s).value, \mathcal{W}(f,s).value \neq \bot \text{ (при этом } \mathcal{W}' = \mathcal{W}) \\ \bot, \mathcal{W}(f.s).busy = \mathbf{true} \\ x = \mu_{\mathcal{W}[f.s.busy\leftarrow\mathbf{true}]\to\mathcal{W}''}(\langle \mathcal{W}(f,s).rules, \mathcal{W}(f,s).\sqsubseteq_q\rangle), \\ \qquad \mathcal{W}' = \Phi_{f.s}(\mathcal{W}''[f.s \leftarrow x, f.s.busy \leftarrow \mathbf{false}]) \quad (x \neq \bot) \\ y = \mu_{\mathcal{W}'''\to\mathcal{W}''''}(\{p.s | p \in \|f.parent\|_{\mathcal{W}''\to\mathcal{W}'''}\}), \\ \qquad \mathcal{W}' = \Phi_{f.s}(\mathcal{W}''''[f.s \leftarrow y, f.s.busy \leftarrow \mathbf{false}]) \quad (y \neq \bot) \\ \bot, \text{ в противном случае} \end{cases} \tag{3.21}$$

Отличие от предыдущего определения состоит в том, что после обратного вывода к заключительному состоянию применяется функция направленного прямого вывода $\Phi_{f.s}$, которая обеспечивает срабатываение всех релевантных правил прямого вывода для только что обновленного слота, а также для всех слотов, значения которых изменились в процессе прямого вывода. Если в данном определении полагать, что функция $\mathcal{W}[\ldots]$ осуществляет проверку ограничений, то мы получим описание семантики вывода с возможностью отката по невыполнению ограничений. Такой откат приведет к использованию других правил для выяснения значения слота, но при этом не будет нарушать монотонности вывода.

3.4.7. Некоторые полезные свойства определенной семантики

При определении отношения $\|\cdot\|$ в доказательстве утверждения 3.2 было использовано свойство монотонности операции вывода, которое может быть сформулировано следующим образом:

Утверждение 3.8. *Для любого контекста вызова $C \in \mathbb{C}$ и выражения $E \in \mathbb{E}$ функция $\|E\|^C_{\cdot\to\cdot} : \mathbb{W} \to \mathbb{W}$ определена и монотонна по $\mathcal{W}$.*

Доказательство. проводится рассмотрением определений $\|\cdot\|$ для различных классов выражений и следует из сформулированного ниже свойства монотонности элементарной операции над состояниями *write*. Корректность рекурсивного определения функции $\|\cdot\|$ была показана, а монотонность следует из представимости фиксированной точки монотонной функции в виде конечной композиции ее применения. □

Лемма 3.3. $\forall\, \mathcal{W} \in \mathbb{W} \quad \mathcal{W}[f.s \leftarrow v] \sqsupseteq \mathcal{W}$

3.5. Комбинированный логический вывод как процесс поиска в графе состояний

Как было показано в утверждении 3.1 в разделе 3.3.1, множество состояний системы может быть представлено в виде конечного фактор-множества $\mathfrak{W}_{\cong}$. Это множество можно также рассматривать в виде направленного биграфа $\mathfrak{G} = \langle \mathfrak{W}_{\cong}, \mathfrak{U}, \mathfrak{V}\rangle$, с двумя множествами дуг $\mathfrak{U}$ и $\mathfrak{V}$, соответствующих возможным переходам в процессе прямого и обратного вывода. При этом:

$$\langle \hat{\mathcal{W}}_1 \in \hat{\mathcal{W}}_2\rangle \in \mathfrak{U} \iff \forall\, \mathcal{W}_1 \in \hat{\mathcal{W}}_1\ \exists \mathcal{W}_2 \in \hat{\mathcal{W}}_2 \ :\ \mathcal{W}_2 = \varphi(\mathcal{W}_1) \tag{3.22}$$

где φ — одношаговая функция исчерпывающего логического вывода, а

$$\langle \hat{\mathcal{W}}_1 \in \hat{\mathcal{W}}_2 \rangle \in \mathfrak{V} \iff \forall\, \mathcal{W}_1 \in \hat{\mathcal{W}}_1\ \exists \mathcal{W}_2 \in \hat{\mathcal{W}}_2 \ :\ \mathcal{W}_2 \triangleleft \mathcal{W}_1 \qquad (3.23)$$

Введенное здесь отношение $\triangleleft$ означает, что состояние $\mathcal{W}_1$ может быть выведено из $\mathcal{W}_2$ за один шаг обратного вывода, т.е. если $|\mathcal{W}_1 - \mathcal{W}_2| = \langle f, s \rangle$, то значение $f.s$ может быть получено как $\|f.s\|_{\mathcal{W}_1 \to \mathcal{W}_2}$ без промежуточных состояний (это гарантируется одноэлементностью разности двух состояний).

Такое представление в виде графа позволяет наглядно визуализировать процесс логического вывода, а также формализовывать семантику этого процесса как процесса поиска в графе, а не путем введения теоретико-решеточных построений. Говоря нестрого, при комбинированном виде производится один шаг по графу в соответствии с дугой обратного вывода ($\mathfrak{V}$), а затем оттуда максимальное количество шагов по дугам прямого вывода $\mathfrak{U}$, пока мы не окажемся в вершине, из которой нет больше исходящих дуг прямого вывода. Тогда производится еще один шаг обратного вывода и т.д.

Более строгая формализация процесса вывода в графе $\mathfrak{G}$ и на множестве $\mathfrak{W}_{\cong}$, а также доказательство ее эквивалентности приведенной выше семантике потребовали бы значительных усилий и здесь не приводятся, хотя исследование полезных свойств графовой модели представляет самостоятельный интерес.

3.6. Язык представления знаний и его семантика

В отличие от классических языков программирования [6], непосредственное описание семантики для языка представления знаний (ЯПЗ) не имеет смысла, так как конструкции языка не носят процедурного смысла и сами по себе не влияют на состояние выполнения. Конструкции языка представления знаний (по крайней мере большинство из них, называемые **декларативными**) формируют начальное состояние задачи $\mathcal{W}_0$, т.е. те его составляющие, которые отвечают за представление базы знаний ($\mathcal{K}_{ss_1}$ и $\mathcal{K}_d$, представленные структурой фреймовой иерархии и присоединенными к слотам правилами). После того, как начальное состояние сформировано, запрашивается значение некоторого слота (используется соответствующая **процедурная** команда языка либо другие, определяемые реализацией, способы), что инициирует логический вывод из начального состояния $\mathcal{W}_0$ с описанной выше семантикой.

Для полного формального описания языка представления знаний необходимо определить правила трансляции синтаксических конструкций языка в математическую модель, представленную множеством $\mathcal{W}_0$, а также привести формальное описание синтаксиса. В данной работе мы из-за громоздкости не будем описывать правила формальной трансляции предложений языка в теоретико-множественные конструкции, а ограничимся лишь пояснениями.

Получение состояния $\mathcal{W}_0$ по тексту на ЯПЗ по сути представляет собой описание семантики языка, при котором денотатом всего языкового представления базы знаний будет искомое состояние $\mathcal{W}_0$, а денотатами языковых конструкций — функции преобразования состояния. Со-

ответственно, построение формальной модели будет строиться на основе семантической функции $\mathcal{L} : A^* \to (\mathcal{W} \to \mathcal{W})$, где A^* — текст программы[5], $\mathcal{W} \to \mathcal{W}$ — соответствующее отображение состояния программы. Начальное состояние $\mathcal{W}_0$ получается из текста программы P как $\mathcal{W}_0 = \mathcal{L}(P)(0_{\mathbb{W}})$, где $0_{\mathbb{W}}$ — нулевое функция состояния, задаваемая соотношением $0_{\mathbb{W}}(f, s) = \langle \bot, \bot, \varnothing, \varnothing, \varnothing, \sqsubseteq_{def}, \sqsubseteq_{def}, \mathbf{false} \rangle$.

База знаний строится как последовательность элементарных конструкций, которые могут быть правилами или описаниями фреймов. Для последовательности конструкций очевидно $\mathcal{L}(P_1; P_2) = \mathcal{L}(P_2) \circ \mathcal{L}(P_1)$.

Конструкция описания фрейма интерпретируется следующим образом:

$$\begin{gathered}\mathcal{L}(\texttt{frame}\ f\ \texttt{parent}\ p\ \{s_1; \ldots; s_n\}) = \mathcal{L}^{(f)}(s_n) \circ \ldots \circ \mathcal{L}^{(f)}(s_1) \circ \\ \circ \mathrm{insert}(f.parent, \langle p, \bot, \varnothing, \varnothing, \varnothing, \sqsubseteq_{def}, \sqsubseteq_{def}, \mathbf{false} \rangle)\end{gathered} \tag{3.24}$$

где s_i — конструкция описания слота, а $\mathrm{insert} : \mathbb{I} \times \mathbb{T} \to (\mathcal{W} \to \mathcal{W})$ — операция добавления слота к функции состояния, определяемая следующим образом:

$$\mathrm{insert}(f.s, X)(\mathcal{W}) = \lambda f_x, s_x . \langle f_x, s_x \rangle = \langle f, s \rangle \to X \star \mathcal{W}(f_x, s_x); \mathcal{W}(f_x, s_x) \tag{3.25}$$

Здесь $\star$ — операция слияния кортежей, представляющих один и тот же слот:

$$\begin{gathered}(u \star v)_i = (u_i = \bot) \to v_i; u_i, \text{ для } i = 1, 2, 6, 7, 8; \\ (u \star v)_i = u_i \cup v_i, \text{ для } i = 3, 4, 5\end{gathered} \tag{3.26}$$

Описание отдельного слота интерпретируется следующим образом (для слота целого типа):

$$\begin{gathered}\mathcal{L}^{(f)}(\texttt{scalar}\ n\ \texttt{int def}\ v\ \texttt{ruleselectionstrategy}\ r\ ) = \\ \mathrm{insert}[f.n, \langle \bot, v, \varnothing, \varnothing, \varnothing, r, \sqsubseteq_{def}, \mathbf{false} \rangle]\end{gathered} \tag{3.27}$$

Более строго следует описать операцию определения слота через составные компоненты определения, как это сделано для описания фрейма, которые будут устанавливать соответствующие компоненты представляющего слот кортежа при помощи отдельных операций insert.

Для формального описания правил придется описывать преобразование выражений во "внешнем" синтаксисе ЯПЗ в синтаксис, описанный в разделе 3.4.1, а также преобразование правил прямого и обратного вывода в соответствующие наборы выражений. Для правил обратного вывода `if` *cond* `then` $f.s = expr$ проделывается следующее: вхождения имени фрейма f в обоих частях выражения заменяется на *this* (это необходимо для корректной работы механизма наследования), и к слоту $f.s$ при помощи операции insert добавляется правило-запрос вида $cond' \to expr'; \bot$, где $cond', expr'$ — модифицированные выражения *cond* и *expr* соответственно. Для правил прямого вывода `when` *cond* `then` *actions* из выражения-посылки *cond* выделяется множество фигурирующих в нем ссылок на слоты $\{\langle f_i, s_i \rangle\}_{i=1}^n$, и к каждому из этих слотов присоединяется правило-демон

[5]Заметим, что A^* не совпадает с областью определения функции $D(\mathcal{L})$, которая задается формальной грамматикой языка.

вида $\langle cond', actions' \rangle$. При этом в $actions$ операции $f.s = v$ заменяются на соответствующие вызовы функции $[f.s \leftarrow v]$, т.е. происходит преобразование из внешнего языка во внутренние структуры математической модели.

4. Семантика распределенного логического вывода

При построении модели распределенной фреймовой иерархии и описании семантики распределенного вывода мы будем действовать по следующей схеме: введем понятие распределенной фреймовой системы, расширим описанный в разделе 3.4.1 синтаксис выражений за счет введения удаленных ссылок ($\blacklozenge$ и $\Diamond$) и на базе этого введем операции комбинирования функций состояния ($\bigstar$, $\blacktriangleleft/\triangleleft$, $\bowtie$). При помощи этого мы определим два вида семантики распределенного вывода: эквивалентную семантику для порожденной нераспределенной системы и семантику, расширяющую описанную в разделе 3.4, определенную на множестве функций состояния и учитывающую процессы удаленного взаимодействия. Это позволит нам сформулировать утверждения об эквивалентности мобильного и статического взаимодействия в смысле их соответствия семантике порожденной системы. В заключение мы покажем, как при помощи функций-оракулов можно учесть особенности реального сетевого взаимодействия, наподобие того, как это сделано в [12].

4.1. Основные понятия и определения

Для описания семантики распределенного вывода нам потребуется рассматривать не только некоторый выделенный фрейм-мир, а всю совокупность входящих во фреймовую коллекцию функций состояния. Состояние всей фреймовой коллекции в этом случае будет описываться набором функций состояния, который мы и назовем распределенной фреймовой системой.

Определение 4.1. Под **распределенной фреймовой системой** $\tilde{\mathcal{W}}$ будем понимать множество функций состояния $\{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$ входящих в систему субиерархий.

Будем рассматривать только такие модели, в которых заранее известны все участвующие в выводе субиерархии. Это ограничение на первый взгляд может показаться слишком существенным, однако ввиду реальной конечности числа возможных фрейм-иерархий оно не накладывает никаких ограничений.

При рассмотрении модели может оказаться удобным вместо нумерования фрейм-миров обозначать их именующими индексами. Такое расширение обозначений по сути дела служит для повышения наглядности и не меняет проводимых рассуждений.

Будем также без ограничения общности предполагать, что фреймы, входящие в каждый из составляющих фрейм-миров, именованы уникальным образом, т.е.

$$\forall i, j \in 1 \ldots n \quad (i \neq j) \Rightarrow I_i \cap I_j = \varnothing \tag{4.1}$$

где I_j — множество идентификаторов фреймов, соответствующих $\mathcal{W}_j$. Этого можно добиться, например, присоединяя к имени фрейма номер или индекс соответствующего фрейм-мира — будем называть такое именование **расширенным именем фрейма**. Различие в именованиях фреймов будет служить для удобства введения дальнейших понятий, так как непересечение имен позволяет по имени фрейма сразу определить функцию состояния, его описывающую.

Определение 4.2. Объединением двух фрейм-миров $\mathcal{W}_1$ и $\mathcal{W}_2$ будем называть фрейм-мир $\mathcal{W}_1 \bigstar \mathcal{W}_2$, описываемый следующей функцией состояния:

$$\mathcal{W}_1 \bigstar \mathcal{W}_2 = \lambda f, s. \begin{cases} \mathcal{W}_1(f,s), & \text{если } \langle f, s\rangle \in \mathbb{I}_1 \\ \mathcal{W}_2(f,s), & \text{если } \langle f, s\rangle \in \mathbb{I}_2 \\ \bot, & \text{в противном случае} \end{cases} \tag{4.2}$$

Из свойства уникальности именования следует

Лемма 4.1.

$$\forall\, \mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3 \quad \mathcal{W}_1 \bigstar \mathcal{W}_2 = \mathcal{W}_2 \bigstar \mathcal{W}_1,\ u\ (\mathcal{W}_1 \bigstar \mathcal{W}_2) \bigstar \mathcal{W}_3 = \mathcal{W}_1 \bigstar (\mathcal{W}_2 \bigstar \mathcal{W}_3) \tag{4.3}$$

Таким образом, можно определить операцию $\bigstar$ для любой последовательности фунций состояния, причем результат не будет зависеть от порядка вычисления.

Определение 4.3. Назовем

$$\mathcal{W} = \bigstar_{i=1}^{n} \mathcal{W}_i \tag{4.4}$$

функцией состояния распределенной фреймовой системы $\tilde{\mathcal{W}} = \{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$. Будем также использовать обозначение $\mathcal{W} = \bigstar \tilde{\mathcal{W}}$.

Помимо операции $\bigstar$ полезно определить операцию **подчиненного наследования** двух фреймовых иерархий следующим образом:

Определение 4.4.

$$\mathcal{W}_1 \underset{f}{\blacktriangleleft} \mathcal{W}_2 = \mathcal{W}_1 \bigstar \mathcal{W}_2[\mathcal{W}_2.object.parent \leftarrow \mathcal{W}_1.f] \tag{4.5}$$

В этом определении, если $\mathcal{W}_1$ и $\mathcal{W}_2$ являются фреймовыми иерархиями (т.е. иерархиями с единственным общим родителем `object`), то результат также будет являться фреймовой иерархией, в которой общий родитель $\mathcal{W}_2$ будет унаследован от указанного фрейма f иерархии $\mathcal{W}_1$. В случае, если запись $\mathcal{W}_1 \underset{f}{\blacktriangleleft} \mathcal{W}_2$ рассматривается в контексте распределенной фреймовой системы, то подразумевается

$$\mathcal{W}_1 \underset{f}{\blacktriangleleft} \mathcal{W}_2 = \{\mathcal{W}_1, \mathcal{W}_2[object.parent \leftarrow \mathcal{W}_1.f]\} \tag{4.6}$$

Кроме подчиненного наследования, определим также полезную операцию **симметричного соединения** $\bowtie$:

Определение 4.5.

$$\mathcal{W}_1 \bowtie \mathcal{W}_2 = (\mathcal{W}_o \underset{object}{\blacktriangleleft} \mathcal{W}_1) \underset{\mathcal{W}_o.object}{\blacktriangleleft} \mathcal{W}_2 \tag{4.7}$$

где $\mathcal{W}_o$ — пустая фреймовая иерархия, содержащая только единственный фрейм *object* без слотов. Аналогично предыдущему определению, в контексте распределенной фреймовой системы можно показать, что имеет место следующая

Лемма 4.2.

$$\begin{aligned} \mathcal{W}_1 \bowtie \mathcal{W}_2 = \ & \{\mathcal{W}_o, \mathcal{W}_1[object.parent \leftarrow \mathcal{W}_o.object], \\ & \mathcal{W}_2[object.parent \leftarrow \mathcal{W}_o.object]\} \end{aligned} \tag{4.8}$$

Следствие 4.1.

$$\forall\, \mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3 \quad \mathcal{W}_1 \bowtie \mathcal{W}_2 = \mathcal{W}_2 \bowtie \mathcal{W}_1, \quad (\mathcal{W}_1 \bowtie \mathcal{W}_2) \bowtie \mathcal{W}_3 = \mathcal{W}_1 \bowtie (\mathcal{W}_2 \bowtie \mathcal{W}_3) \tag{4.9}$$

Введенные выше операции позволяют строить распределенные фреймовые системы, комбинируя их из существующих функций состояния. Таким образом, некоторое выражение, построенное в терминах операций $\bigstar$, $\blacktriangleleft$ и $\bowtie$ полностью определяет как саму фреймовую систему, так и соответствующую ей функцию состояния[1].

4.2. Семантика распределенного вывода в терминах порожденной системы

Так как распределенная система в свою очередь представляет собой единую фреймовую иерархию, то модель распределенного вывода представляет собой синхронный поиск в общем пространстве состояний задачи [13], т.е. общее пространство состояний распределено по фреймовой иерархии, и соответственно поиск управляется поочередно разными процессорами.

Проще всего определить семантику вывода в распределенной системе на основе ее функции состояния. Функция состояния распределенной системы является функцией состояния некоторой нераспределенной фреймовой иерархии, которую мы будем называть **порожденной фреймовой системой** для исходной распределенной фреймовой коллекции. Таким образом, при определении семантики распределенного вывода можно рассматривать порожденную систему, семантика для которой уже является определенной.

Однако такая семантика не позволяет нам исследовать процессы распределенного взаимодействия, возникающие во фреймовой коллекции. По сути дела порожденная система представляет собой множество всех фреймов распределенной иерархии, собранных локально в единый фрейм-мир. Таким образом в данной семантике не представляется возможным контролировать множество знаний, сосредоточенных в каждом из узлов сети в каждый момент времени.

[1] Строго говоря, для доказательства этого утверждения следует показать, что для всех операций $\circledast \in \{\bigstar, \blacktriangleleft, \bowtie\}$ верно $\mathcal{W}_1 \circledast \mathcal{W}_2 = \{\mathcal{W}'_1, \mathcal{W}'_2\} \Rightarrow \mathcal{W}_1 \circledast \mathcal{W}_2 = \mathcal{W}'_1 \bigstar \mathcal{W}'_2$.

4.3. Семантика распределенного вывода в терминах состояний исходной распределенной системы

Для рассмотрения распределенных взаимодействий требуется определять семантику на всем множестве состояний распределенной системы $\tilde{\mathcal{W}} = \{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$. По аналогии с семантикой, рассмотренной в разделе 3.4, будем рассматривать отображение $\tilde{\mathcal{E}} : \mathbb{E} \times \tilde{\mathbb{W}} \times \mathbb{C} \to \mathbb{T} \times \tilde{\mathbb{W}}$, которую для ясности будем обозначать как $v = [\![E]\!]^C_{\tilde{\mathcal{W}} \to \tilde{\mathcal{W}}'} \iff \tilde{\mathcal{E}}(E, \tilde{\mathcal{W}}, C) = (v, \tilde{\mathcal{W}}')$.

Синтаксис выражений расширим за счет введения **удаленных ссылок**, т.е. ссылок на имена слотов, расположенных в отличных от текущего фрейм-мирах. Введем также операции **статического** ($\blacklozenge$) и **мобильного** ($\lozenge$) вычисления удаленной ссылки. С учетом этих операций можно также определить две операции подчиненного наследования — статическую ($\blacktriangleleft$) и мобильную ($\triangleleft$) — следующим образом:

$$\begin{aligned} \mathcal{W}_1 \underset{f}{\blacktriangleleft} \mathcal{W}_2 &= \{\mathcal{W}_1, \mathcal{W}_2[object.parent \leftarrow \blacklozenge \mathcal{W}_1.f]\} \\ \mathcal{W}_1 \underset{f}{\triangleleft} \mathcal{W}_2 &= \{\mathcal{W}_1, \mathcal{W}_2[object.parent \leftarrow \lozenge \mathcal{W}_1.f]\} \end{aligned} \tag{4.10}$$

Расширим определение фреймовой системы $\tilde{\mathcal{W}}$, дополнив ее понятием "текущего" фрейм-мира, в котором в данный момент производится логический вывод. Для удобства будем записывать это, помечая текущий фрейм-мир чертой, например $\tilde{\mathcal{W}} = \{\mathcal{W}_1, \overline{\mathcal{W}}_2, \ldots, \mathcal{W}_n\}$[2].

Тогда семантика вычисления в распределенной системе произвольного выражения $E \in \mathbb{E}$, не содержащего удаленных ссылок, будет определяться через семантику локального вывода естественным образом:

$$[\![E]\!]_{\{\mathcal{W}_1, \ldots, \overline{\mathcal{W}}_i, \ldots, \mathcal{W}_n\} \to \{\mathcal{W}_1, \ldots, \overline{\mathcal{W}}'_i, \ldots, \mathcal{W}_n\}} = \|E\|_{\mathcal{W}_i \to \mathcal{W}'_i} \tag{4.11}$$

Особый интерес будет представлять семантика обработки удаленных ссылок, которая будет рассмотрена в следующих подразделах.

4.3.1. Семантика мобильного удаленного вывода

В случае использования мобильной ссылки ее вычисление сводится к тому, что упомянутый в ней фрейм-мир присоединяется (с помощью операции $\bigstar$) к текущему, и логический вывод продолжается в текущем фрейм-мире — при этом удаленная ссылка заменяется обычной:

$$[\![\lozenge \mathcal{W}_j(f, s)]\!]_{\{\mathcal{W}_1, \ldots, \overline{\mathcal{W}}_i, \ldots, \mathcal{W}_n\} \to \tilde{\mathcal{W}}'} = [\![f.s]\!]_{\{\mathcal{W}_1, \ldots, \overline{\mathcal{W}_i \bigstar \mathcal{W}_j}, \ldots, \mathcal{W}_n\} \to \tilde{\mathcal{W}}'} \tag{4.12}$$

В случае, если во всей распределенной фреймовой системе используется только мобильный удаленный вывод, то текущим всегда остается только один фрейм-мир; при этом он монотонно расширяется за счет присоединения к нему других фрейм-миров.

Мобильное наследование с поддержкой прямого и обратного вывода реализуется через вычисление удаленной ссылки естественным образом.

[2] Строго говоря, фреймовая система в данном случае будет представлять собой упорядоченный набор из множества фрейм-миров и идентификатора текущего фрейм-мира.

При необходимости обращения к удаленному родителю производится вычисление удаленной ссылки по формуле 4.12, что приводит к объединению двух фрейм-миров и проведению логического вывода в полном соответствии с локальной семантикой.

4.3.2. Семантика статического удаленного вывода

При статическом удаленном выводе модификация фрейм-миров не производится, но меняется "текущий" фрейм-мир, и дальнейший вывод производится уже с использованием другой функции-состояния (и, соответственно, определенных в ней правил):

$$[\![\blacklozenge \mathcal{W}_j(f,s)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = [\![f.s]\!]_{\{\mathcal{W}_1,\ldots,\mathcal{W}_i,\ldots,\overline{\mathcal{W}}_j,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} \qquad (4.13)$$

При удаленном наследовании ситуация обстоит несколько сложнее, чем в случае мобильного вывода. При обратном выводе, когда в результате вычисления значения слота управление переходит к удаленному родителю, и логический вывод продолжается для другой функции состояния, базовый фрейм, подставляемый в правила вместо $this$, должен соответствовать исходному фрейму, который с точки зрения новой функции состояния является удаленным. Таким образом, переход контекстного вывода на другую функцию состояния приводит к появлению удаленной ссылки в контексте вывода:

$$[\![\blacklozenge \mathcal{W}_j(f,s)]\!]^{F}_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = [\![f.s]\!]^{\blacklozenge\mathcal{W}_i(F)}_{\{\mathcal{W}_1,\ldots,\mathcal{W}_i,\ldots,\overline{\mathcal{W}}_j,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} \qquad (4.14)$$

При этом в процессе вывода для функции состояния $\mathcal{W}_j$ при каждом вычислении $this$ будет производиться повторное вычисление удаленной ссылки и перенос текущего фрейма назад к $\mathcal{W}_i$ — при этом операция вычисления удаленной ссылки будет снята, в том числе и с контекста вызова, в соответствии с правилом

$$[\![\blacklozenge \mathcal{W}_i(F)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}} = \|F\|_{\mathcal{W}_i\to\mathcal{W}_i} = \mathcal{W}_i(F) \qquad (4.15)$$

4.4. Свойства эквивалентности различных семантик распределенного вывода. Нормальные системы

Формальное описание семантики для распределенного мобильного и статического вывода позволяет сформулировать некоторые утверждения об эквивалентности распределенного вывода и вывода в эквивалентной фреймовой иерархии. Эти утверждения оказываются истинными в некотором классе систем, которые мы назовем **нормальными**. К этому классу относятся системы, построенные на одном виде взаимодействия (удаленный вызов или включение), **строго полиморфные** системы (в которых все правила являются внутрифреймовыми, и все ссылки на слоты в правилах полиморфны), а также системы, в которых доступ к каждой отдельно взятой фреймовой иерархии осуществляется только на основе одного вида взаимодействия (**сепаратные**). Следует отметить, что большинство возникающих в реальных задачах распределенного накопления и использования знаний систем является нормальными.

Утверждение 4.2 (о вычислении мобильной ссылки).

$$[\![\Diamond \mathcal{W}_j(f,s)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = \|\mathcal{W}_j(f,s)\|_{\mathcal{W}\to\mathcal{W}'} \tag{4.16}$$

где $\mathcal{W} = \mathcal{W}_1 \bigstar \ldots \bigstar \mathcal{W}_n$ *— общее состояние фреймовой системы, при этом* $\mathcal{W}' = \bigstar \tilde{\mathcal{W}}'$.

Идея доказательства. При вычислении мобильная ссылка преобразуется к обычной путем объединения двух соответствующих функций состояния, таким образом текущая функция состояния распределенной фреймовой системы постепенно включает в себя все участвующие в выводе фрейм-миры, играя по сути дела роль общей функции состояния фреймовой системы $\bigstar \mathcal{W}$. Распределенный вывод в этом случае сводится к локальному, и эквивалентность следует из единообразия определения семантики локального вывода. □

Утверждение 4.3 (о вычислении статической ссылки).

$$[\![\blacklozenge \mathcal{W}_j(f,s)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = \|\mathcal{W}_j(f,s)\|_{\mathcal{W}\to\mathcal{W}'} \tag{4.17}$$

где $\mathcal{W} = \mathcal{W}_1 \bigstar \ldots \bigstar \mathcal{W}_n$ *— общее состояние фреймовой системы, при этом* $\mathcal{W}' = \bigstar \tilde{\mathcal{W}}'$.

Идея доказательства. Следует для начала рассмотреть процесс вычисления ссылки в том случае, когда дальнейший распределенный вывод не инициируется, и затем перейти к индукции по длине цепочки удаленного вывода. Следует отметить, что дальнейшие удаленные ссылки не возникают в том случае, когда все необходимые для вычисления выражения слоты находятся в рамках текущей фрейм-иерархии $\mathcal{W}_i$, а в этом случае семантика вывода в ней будет соответствовать семантике вывода в иерархии $\mathcal{W} \supseteq \mathcal{W}_i$[3] (что следует строго показать на основании определения семантики в разделе 3.4). □

Утверждение 4.4 (Об эквивалентности семантики распределенного и локального вывода). *Для любого выражения* $E \in \mathbb{E}$ *в нормальной системе* $\tilde{\mathcal{W}}$

$$[\![E]\!]_{\tilde{\mathcal{W}}\to\tilde{\mathcal{W}}'} = \|E\|_{\mathcal{W}\to\mathcal{W}'} \tag{4.18}$$

где $\mathcal{W} = \bigstar \tilde{\mathcal{W}}$ *и* $\mathcal{W}' = \bigstar \tilde{\mathcal{W}}'$.

Идея доказательства. Следует воспользоваться индукцией по длине выражения E, используя результаты двух предыдущих утверждений. □

Это утверждение, в частности, еще раз показывает, что нормальная распределенная фреймовая иерархия с мобильным, статическим или смешанным наследованием в смысле семантики будет эквивалентна некоторой локальной фреймовой иерархии. Кроме того, отсюда следует эквивалентность статического и мобильного удаленного наследования в терминах определяемой ими семантики вычислений — разница проявляется лишь в

[3] В данном случае включение понимается в терминах множества слотов, т.е. $\mathcal{W}_1 \subseteq \mathcal{W}_2 \iff \mathbb{I}_1 \subseteq \mathbb{I}_2$, и $\forall\ \langle f,s\rangle \in \mathbb{I}_1\, \mathcal{W}_1(f,s) = \mathcal{W}_2(f,s)$, где $\mathbb{I}_1 = \mathcal{D}\mathcal{W}_1$, $\mathbb{I}_2 = \mathcal{D}\mathcal{W}_2$.

том, как общие знания в системе будут распределены между ее составными частями. При мобильном наследовании все знания (статические и динамические) по мере вывода аккумулируются в одной текущей субиерархии, в то время как при статическом наследовании они распределены по сети, и обмен статическими знаниями происходит каждый раз по мере необходимости.

4.5. Учет особенностей реального сетевого взаимодействия

При проведении удаленного вызова или при передаче фрейм-мира по сети в реальной распределенной системе могут возникать различного рода проблемы сетевого взаимодействия. Для учета такого рода проблем в [12] предлагается воспользоваться функцией-оракулом $download(P)$, моделирующей процесс сетевых взаимодействий. В нашем случае роль функции-оракула будут выполнять операции $\blacklozenge$ и $\Diamond$, которые в зависимости от успешности сетевых операций будут при интерпретации возвращать требуемое значение или $\perp$. С учетом этого формулы 4.12 и 4.13 преобразуются следующим образом:

$$[\![\Diamond \mathcal{W}_j(f,s)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = \begin{cases} [\![f.s]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}_i \bigstar \mathcal{W}_j},\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'}, \\ \text{если передача данных от узла} \\ \quad j \text{ к узлу } i \text{ прошла успешно} \\ \perp, \text{если возникла ошибка} \\ \text{сетевого взаимодействия} \\ \quad \text{между узлами } i \text{ и } j \end{cases} \tag{4.19}$$

$$[\![\blacklozenge \mathcal{W}_j(f,s)]\!]_{\{\mathcal{W}_1,\ldots,\overline{\mathcal{W}}_i,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'} = \begin{cases} [\![f.s]\!]_{\{\mathcal{W}_1,\ldots,\mathcal{W}_i,\ldots,\overline{\mathcal{W}}_j,\ldots,\mathcal{W}_n\}\to\tilde{\mathcal{W}}'}, \text{ если удаленный} \\ \quad \text{вызов узла } j \text{ узлом } i \text{ прошел успешно} \\ \perp, \text{ если возникла ошибка сетевого} \\ \quad \text{взаимодействия между узлами } i \text{ и } j \end{cases} \tag{4.20}$$

В случае такого определения вычисления удаленной ссылки семантика перестает быть детерминированной, так как результат вычисления выражения начинает зависеть от поведения внешних функций-оракулов. Однако такое определение семантики позволяет учитывать особенности реальных процессов передачи данных в компьютерных сетях и моделировать поведение процессов логического вывода в условиях неудачных сетевых взаимодействий.

5. Интеграция фреймовой модели с базами данных и императивными компонентами

При создании сложных интеллектуально-информационных систем возникает необходимость совместного использования интеллектуальной составляющей и императивного программного кода, элементов различных компонентных моделей (JavaBeans, COM/DCOM/ActiveX-компонентов,

CORBA-объектов), а также реляционных и объектных баз данных. Такие компоненты могут реализовывать корпоративные хранилища данных (data warehouses) либо традиционную корпоративную бизнес-логику, и возможность их прозрачного включения во фреймовую модель была бы чрезвычайно привлекательной.

Среди перечисленных выше можно выделить следующие классы объектов:

- **Пассивные реляционные структуры**, представленные реляционными таблицами баз данных. Хотя такие структуры могут содержать в себе некоторый активный программный код, реализующий бизнес-логику (триггеры, встроенные процедуры), с внешней точки зрения данные всегда представляются реляционным отношением, возможно, динамически вычисляемым.
- **Пассивные** или **активные объектные структуры**, представленные набором (иерархией) именованных объектов или интерфейсов. К этому классу можно отнести объектные базы данных и программные компонентные модели (ActiveX, CORBA, ...).
- **Смешанные структуры**, в которых попытка интеграции объектного и реляционного представления уже была сделана, и результирующая модель данных соединяет в себе черты обоих структур (например, СУБД PostgreSQL).

В следующих разделах мы покажем, что реляционные и объектные структуры имеют с фреймовой моделью много общего, и что несложными "топологическими" превращениями можно унифицировать эти структуры, превратив их в семейства фреймов, доступные обычным образом в процессе логического вывода.

5.1. Фреймовая модель и реляционные базы данных

При создании реальных интеллектуальных систем во многих случаях важно интегрировать интеллектуальную функциональность в существующую информационную систему, т.е. использовать уже накопленные данные как исходные для решения задачи. Так как в подавляющем большинстве случаев для хранения больших объемов данных используется реляционная модель, то особый интерес представляет интеграция инструментария с реляционными базами данных.

В области взаимодействия баз данных и знаний в настоящее время существуют два основных направления:

- **Активные базы данных** (active databases) [15], представляющие собой расширение реляционной модели за счет введения триггеров-правил, работающих по принципу прямого вывода.
- **Легковесные дедуктивные базы данных** (lightweight deductive databases) [14], в которых за основу берется логическая модель представления знаний, а реляционная модель данных "моделируется" многоместными предикатами-фактами. Соответственно процесс логического вывода как правило построен на методе резолюции с обратным

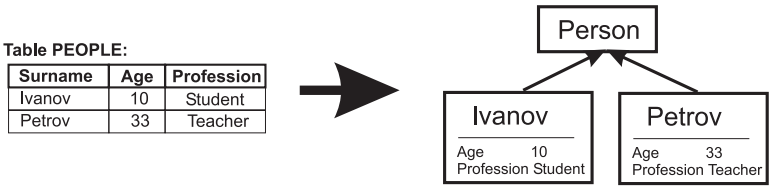


Рис. 2. Пример экстенсионала (фрейм-класса) для доступа к реляционной БД

выводом от цели к фактам.

Активные базы данных удобны для хранения больших объемов данных со сравнительно небольшими "интеллектуальными" вкраплениями, в то время как легковесные дедуктивные базы данных неэффективны на больших объемах данных (так как выборка данных происходит в соответствии с моделью логического программирования, т.е. на основе унификации, а не с использованием специальных методов индексирования), однако они представляют более широкие возможности для программирования интеллектуальной части. Соответственно, для обращения к активным базым данных используется стандартный для реляционных баз данных язык запросов (SQL), а для дедуктивных баз данных — язык запросов логического программирования.

В данной работе предлагается промежуточный подход, при котором реляционная база данных используется для задания некоторых атрибутов (исходных данных) решаемой задачи в виде значений слотов фреймов и семейств фреймов специального вида, которые затем используются в процессе логического вывода обычным образом, позволяя таким образом делать интеллектуальные заключения относительно содержащихся в базе данных сведений. В рамках инструментария JULIA это реализуется одним из следующих способов:

- **Экстенсионал или фрейм-класс реляционной таблицы** позволяет представить реляционное отношение $R = \langle R_1, \ldots, R_n \rangle$ или проекцию в качестве семейства фреймов с общим родителем. Некоторый ключевой атрибут отношения R_i выбирается в качестве **именующего** атрибута[1], в соответствии с которым именуются фреймы, соответствующие отдельным строкам отношения. Слоты фреймов именуются в соответствии с метаданным — именами столбцов отношения. Такое преобразование реляционной таблицы в множество фреймов будем называть **горизонтальным срезом** таблицы.
  Хотя фрейм-класс не позволяет напрямую связывать со слотами правила прямого или обратного вывода, такие правила могут определяться для родительского фрейма. В этом случае при запросе незаполненного атрибута фрейм-класса применяются правила (прямого и обратного вывода), определенные для соответствующего слота фрейма-

[1]В этом случае предполагается, что у отношения есть единственный ключевой атрибут. Хотя такое ограничение во многих случаях не является слишком сильным, теоретически возможно использовать несколько атрибутов в качестве именующих, формируя имя фреймов конкатенацией значений соответсвующих атрибутов

родителя. Правила прямого вывода могут срабатывать только при явном запросе значений соответствующих слотов фрейм-класса, либо при создании фрейм-класса может производиться просмотр всех входящих в него фреймов. Последнее может оказаться весьма длительным процессом и менее эффективным по сравнению с применяемыми в активных базах данных алгоритмами Gator, TREAT или RETE [7], однако неэффективность возникает по той причине, что интеллектуальная составляющая системы в нашем случае работает во времени независимо от реляционной, и такой предварительный просмотр множества данных необходим для того, чтобы обеспечить "синхронизацию" значений слотов во фреймовой модели с данными, т.е. отложенным образом применить все правила прямого вывода, которые возможны для набора данных.

Для выполнения операции выборки, т.е. поиска строк реляционной таблицы, удовлетворяющих определенным ограничениям (в том числе по интеллектуальным атрибутам), используется процедура вычисления выражения на множестве потомков родителя фрейм-класса (`MAP` или `subclass`). С точки зрения эффективности используемый линейный поиск уступает активным базам данных и сравним с перебором, используемым в классическом логическом программировании, но при этом имеется возможность применения обратного вывода в процессе поиска, совместно с прямым выводом.

- **БД-фрейм** или фрейм-выборка служит для явного представления одной или нескольких строк реляционной таблицы, удовлетворяющих некоторому явно заданному в виде SQL-выражения условию c. При выборке значения определенного слота s результат определяется SQL-запросом `SELECT` s `FROM table WHERE` c (если возвращаются несколько строк таблицы с различными значениями — результатом будет список), а при присваивании выполняется обновление всех строк таблицы с данным условием `UPDATE table SET` $s = v$ `WHERE` c. Полученный таким образом фрейм будем называть **горизонтальным фактор-срезом** таблицы, так как он характеризует множество строк таблицы, выделенных некоторым отношением эквивалентности, порожденным выражением c.

- **Вертикальный срез**, в некотором смысле противоположный горизонтальному, когда фреймы порождаются столбцами таблицы, а значения в строках заполняют слоты в соответствии с некоторым именующим столбцом. Такой вариант доступа ввиду его малой практической применимости в инструментарии не реализован и рассматриваться не будет.

- **Другие средства доступа к БД**. Процедуры и демоны для доступа к БД позволяют указать в качестве присоединенной процедуры (запроса или демона) некоторое SQL-предложение, или (в более простой форме) имя таблицы и атрибута. Кроме того, инструментарий содержит библиотеку функций для доступа к РСУБД, позволяющую получать значения из таблиц базы данных в составе произвольного выражения, например:

```
IMPORT LIBRARY 'com.shwars.julia.Libraries.DB';
SET Person.Children =
   &DB_SELECT("SELECT Name FROM Person WHERE Person.ID = " + Person.DB_ID);
WHEN Person.Deceased THEN
   &DB_UPDATE("DELETE FROM PERSON WHERE Person.ID = " + Person.DB_ID);
```

5.2. Семантика доступа к реляционным структурам

Для определения процесса доступа к внешним реляционным структурам нам потребуется ввести понятие фрейм-класса, а также функции доступа $\mathfrak{R}$ для получения отдельных компонентов реляционного отношения $\mathcal{R}$.

Определение 5.1. Фрейм-классом будем называть функцию $\hat{\mathcal{F}} : \mathbb{I} \to \mathcal{F}$, возвращающую фрейм-определяющую функцию по имени принадлежащего фрейм-классу фрейма, либо $\perp$.

Понятие фрейм-класса можно расширить, допустив применение его к состоянию $\mathcal{W}$ следующим образом:

$$\hat{\mathcal{F}}(\mathcal{W}) = \lambda f. \begin{cases} \hat{\mathcal{F}}(f), & \hat{\mathcal{F}}(f) \neq \perp \\ \mathcal{W}(f), & \text{в противном случае} \end{cases} \tag{5.1}$$

Таким образом, применив некоторый фрейм-класс к текущему состоянию, получим новое состояние, в котором все фреймы данного фрейм-класса будут доступными.

Определение 5.2. Пусть $\mathcal{R} = \{\langle x_0^{(i)}, \ldots, x_n^{(i)} \rangle\}_{i=1}^N$ — $n+1$-местное реляционное отношение с метаданными (именами столбцов) $\langle \bar{x}_0, \ldots, \bar{x}_n \rangle$. Без ограничения общности будем полагать x_0 ключевым атрибутом. Тогда соответствующую этому отношению функцию доступа определим следующим образом:

$$\mathfrak{R}_{\mathcal{R}}(f, s) = \begin{cases} x_j^{(i)}, & \text{если } \exists j : \bar{x}_j = s, \quad \exists i : x_0^{(i)} = f \\ \perp, & \text{в противном случае} \end{cases} \tag{5.2}$$

Тогда соответствующий горизонтальному срезу отношения $\mathcal{R}$ фрейм-класс можно будет определить как:

$$\hat{\mathcal{F}} = \lambda f, s. \mathfrak{R}_{\mathcal{R}}(f, s) \tag{5.3}$$

Определение вертикального среза будет производиться аналогично, где вместо функции $\mathcal{R}$ будет использоваться $\mathcal{R}'$, определенная как

$$\mathfrak{R}'_{\mathcal{R}}(f, s) = \begin{cases} x_j^{(i)}, & \text{если } \exists j : \bar{x}_j = f, \quad \exists i : x_0^i = s \\ \perp, & \text{в противном случае} \end{cases} \tag{5.4}$$

Для получения фактор-среза по некоторому SQL-выражению c определим функцию-оракул (oracle function) $\mathfrak{R}_{\mathcal{R}}(c)$, применяющую к отношению $\mathcal{R}$ SQL-выражение c и инкапсулирующую в себе всю функциональность РСУБД и семантику соответствующих SQL-предложений. Тогда в

качестве БД-фрейма будет выступать фрейм-функция специального вида, интерпретация которой будет задаваться следующим образом:

$$\|f.s\|_{\mathcal{W}\to\mathcal{W}} = \mathfrak{R}_f(\texttt{SELECT}\ s\ \texttt{FROM}\ f\ \texttt{WHERE}\ \mathcal{W}(f,s).value) \qquad (5.5)$$

В данном случае предполагается, что в качестве значения функция БД-фрейма возвращает SQL-подвыражение для фактор-среза.

Операция присваивания значения *write* будет определяться в этом случае как

$$\mathcal{W}[f.s \leftarrow v] = \mathcal{W},\ \text{и}\ \mathfrak{R}_f(\texttt{UPDATE}\ f\ \texttt{WHERE}\ \mathcal{W}(f,s).value) \qquad (5.6)$$

В этом случае состояния $\mathcal{W}$ и $\mathcal{W}[f.s \leftarrow v]$ будут формально совпадать — изменения коснутся лишь внешнего отношения, инкапсулированного в поведении функции-оракула[2]. Поэтому в случае применения функций-оракулов следует быть осторожным при изучении поведения логического вывода с помощью графов, так как два принципиально различных состояния могут оказаться неразличимыми.

5.3. Интеграция компонентных и объектных моделей с фреймовым представлением знаний

Во многих случаях при построении гибридных систем оказывается полезным иметь возможность расширить базу знаний процедурными включениями на императивных языках программирования. Такая возможность предусмотрена во многих существующих системах (CLIPS, JESS), однако, как правило, она представлена возможностью определения пользовательских функций на базовом языке (C++ или Java).

Использование фреймовой модели позволяет использовать императивные включения более широко, так как объектные и компонентные модели по своей сути весьма похожи на фреймы. В общем виде объект или класс представляет собой набор атрибутов и методов, при этом как правило делается различие между классом, определяющим только функциональность в виде набора методов, и собственно объектом, содержащим конкретные значения атрибутов и использующим функциональность родительского класса.

Фреймовый подход унифицирует понятие класса и объекта, а также атрибута и метода, так как фрейм слота может содержать как данные, так и применяемые к ним процедуры. Таким образом, возникает естественное представление классов и компонентов в виде фреймов — надо лишь оговорить правила трансляции имен атрибутов и методов. Поскольку методы во фреймовом подходе не могут быть вызваны явно, то они преобразуются в соответствии с некоторым правилом к слотам, при запросе или присваивании значений которым и срабатывают процедуры внешнего, ассоциированного с фреймом, объекта. Аналогичный метод представления распространяется на компоненты в той или иной компонентной модели (JavaBeans, CORBA и др.), так как они по сути дела являются наборами методов без атрибутов — т.е. частным случаем объектной модели.

[2]В случае с горизонтальным срезом таких сложностей не возникает, так как само отношение оказывается неявно сохраненным в состоянии $\mathcal{W}$ после применения фрейм-класса.

Таким образом, во фреймовой системе существует целый спектр возможностей по интеграции императивных включений во фреймовую модель:

- **Представление класса или компонента фреймом** (в соответствии с рассмотренными выше принципами). При этом во фреймовой иерархии такой фрейм представляется заглушкой (stub), которая при обращении к слотам вызывает соответствующие процедуры присоединенного экземпляра класса или компонента. Для определения атрибута с именем `X` в классе или компоненте должны присутствовать функции `getX` и `setX`, либо переменная с именем `theX`. Фрейм автоматически снабжается соответствующими слотами, исследуя класс или компонент при помощи Java Reflection API, либо репозитария интерфейсов CORBA. Использование соглашения об именовании, принятого в JavaBeans, позволяет использовать JavaBeans как фреймы без специального дополнительного программирования.
- Определение **функций пользователя** позволяет описывать на базовом языке программирования наборы тематических функций, присоединяемых динамически в виде библиотек.
- Создание **пользовательских стратегий выбора правил** при прямом и обратном выводе позволяет расширять имеющийся в инструментарии стандартный набор стратегий путем реализации соответствующего класса на базовом языке программирования.
- Реализация произвольных **процедур-демонов и процедур-запросов** на базовом языке программирования путем наследования от соответствующих суперклассов инструментария.

5.4. Семантика доступа к внешним объектам при логическом выводе

Для представления внешних объектов необходимо также использовать функцию-оракул $\mathfrak{E}$ для моделирования процесса выполнения внешнего программного кода. Определим ее следующим образом:

$$\mathfrak{E}(o.x) = \begin{cases} o.x, \text{ если } x \text{ — атрибут объекта } o \\ \text{результат выполнения } o.x, \text{ если } x \text{ — метод объекта } o \\ \bot, \text{ если объект } o \text{ не содержит метода или атрибута } o \\ \quad \text{или если возникла ошибка времени выполнения} \end{cases} \tag{5.7}$$

Тогда некоторому объекту o будем ставить в соответствие фрейм-функцию F_o специального вида $F_o = \lambda s.\mathfrak{E}(o.s)$, для которой правило интерпретации будет иметь вид

$$\|f.x\|_{\mathcal{W}\to\mathcal{W}} = \|F_o(s)\| = \mathfrak{E}(o.s), \text{ где } F_o = \mathcal{W}(f,s) \tag{5.8}$$

а функция *write* будет определяться следующим образом:

$$\mathcal{W}[f.s \leftarrow v] = \begin{cases} \mathcal{W}[f.s \leftarrow v], \text{ и } \mathfrak{E}(f.s), \text{ если } s \text{ — метод} \\ \mathcal{W}[f.s \leftarrow v], \text{ и } \mathfrak{E}(f.s := v), \text{ если } s \text{ — атрибут} \end{cases} \tag{5.9}$$

6. Вопросы реализации

На основе рассмотренной выше семантики автором был разработан открытый инструментарий JULIA (Java Universal Library for Intelligent Applications) для построения распределенных интеллектуальных систем на основе архитектуры распределенной фреймовой иерархии.

6.1. Основные принципы

При проектировании инструментария в первую очередь принимались во внимание следующие соображения:

- **Расширяемость**, т.е. возможность расширения инструментария произвольным программным кодом, включая пользовательские функции, реализованные на традиционном языке программирования, дополнительные типы данных, операции, стратегии выбора правил, процедуры-демоны, процедуры запросы и т.д.
- **Открытость** обеспечивается использованием открытых стандартов CORBA и XML для взаимодействия узлов системы.
- **Многоплатформенность** достигаемая за счет реализации инструментария на языке Java.
- **Высокая степень интеграции** инструментария с **реляционными хранилищами данных**, с произвольным программным кодом (Java-классами) и **компонентами** корпоративной бизнес-логики (CORBA-объектами, JavaBeans и Enterprise JavaBeans).
- **Гибкость** инструментария, позволяющая настраивать стратегию эвристического поиска при логическом выводе.
- **Встраиваемость** инструментария в состав программных и информационных систем, реализованных как на Java, так и на любом языке программирования с поддержкой CORBA. Инструментарий может также использоваться в интернет-приложениях в составе Java сервлетов, апплетов или специализированных CGI-модулей. Механизм *обратных вызовов* (callbacks) позволяет использовать инструментарий в режиме недиалогового (неинтерактивного) обратного вывода.

6.2. Архитектура инструментария

Предлагаемая архитектура реализации гибридного инструментария JULIA для построения распределенных интеллектуальных систем приведена на рис. 3. Инструментарий представляет собой набор Java-классов с соответствующим программным интерфейсом (API), реализующих также набор CORBA-объектов для удаленного доступа к системе со стороны других JULIA-серверов, а также со стороны программных систем, поддерживающих CORBA.

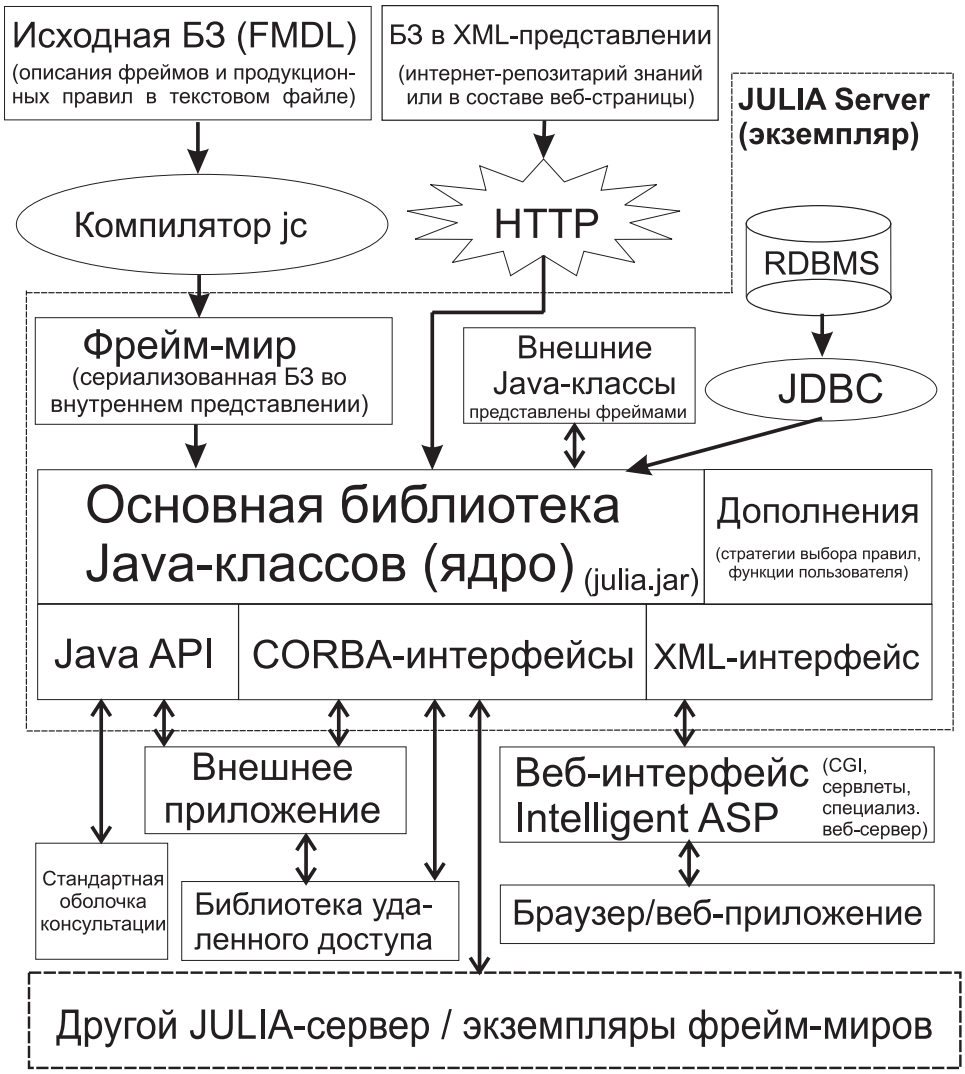


Рис. 3. Архитектура системы на базе инструментария JULIA

В основе системы лежит **ядро**, включающее в себя набор классов для поддержки фреймов и наследования, библиотеку типов данных, алгоритмы реализации прямого и обратного вывода, ряд дополнительных классов для обеспечения доступа к БД, к внешним Java-классам, компонентам JavaBeans, набор стратегий выбора правил при разрешении конфликтов, библиотека основных функций и др. В виде отдельных библиотек оформлены классы, обеспечивающие поддержку CORBA для распределенного вывода (`julia_remote.jar`), а также набор утилит (`julia_utils.jar`), в том числе компилятор с языка представления знаний JFMDL во внутреннее сериализованное представление базы знаний.

В верхней части диаграммы показаны различные способы, с помощью которых система получает знания и данные для своей работы; ниже ядра — различные интерфейсы для использования библиотеки в интерактивной консультации или из внешней информационной системы. Из рисунка видно, что инструментарий может использоваться как самостоятельная оболочка экспертных систем для интерактивной консультации, функционирующая в

виде Java-апплета в браузере клиента или сервлета на веб-сервере[1], а также как составная часть программных систем через программный интерфейс Java (Julia API) или набор CORBA-интерфейсов.

База знаний обычно описывается на специализированном языке высокого уровня декларативности **JFMDL** (Julia Frame Model Definition Language), который включает в себя средства описания статических закономерностей предметной области (фреймовой иерархии) и динамических знаний в виде продукций прямого и обратного вывода. Это описание затем преобразуется специальным транслятором в набор Java-объектов (так называемый **фрейм-мир**), соответствующих конструкциям языка и представляющих собой начальное состояние $\mathcal{W}_0$, из которого будет начат логический вывод. Это т.н. **внутреннее представление** может быть непосредственно использовано для логического вывода, либо в любой момент сериализовано в поток и сохранено в виде файла на диске, в виде внешнего XML-представления, либо в объектной базе данных. В частности, при построении реальных систем имеет смысл сохранять базу знаний в виде внутреннего сериализованного представления, а для использования такой базы знаний применять более компактную библиотеку времени выполнения (`julia_rt.jar`), не включающую в себя достаточно объемные компоненты компилятора.

6.3. Локальный логический вывод

Вне зависимости от способа внешнего представления знаний, продукционные правила обратного вывода транслируются в соответствующие процедуры-запросы, присоединенные к слотам, содержащимся в правой части правил, а правила прямого вывода — в соответствующие процедуры-демоны, присоединенные к слотам, содержащимся в левой части правил. Слоты, значения которых должны быть запрошены у пользователя в процессе обратного вывода, снабжаются соответствующими экземплярами класса, реализующего интерфейс с пользователем.

Обратный вывод реализуется в соответствии с рассмотренной ранее семантикой. Для вычисления значения слота используется процедура `get`, которой в качестве параметра передается ссылка на базовый фрейм. Эта процедура применяет все ассоциированные правила-демоны, заменяя в них все вхождения `this` на базовый фрейм, тем самым обеспечивая применение определенных во фрейме-родителе правил к текущему базовому фрейму. В случае, если ни одно из правил не выполнилось, или полученное значение не удовлетворяет ограничениям, запрашивается процедура `get` для фрейма-родителя (или нескольких фреймов-родителей).

Для реализации прямого вывода можно использовать алгоритм Rete [7] или ему подобные (TREAT, GATOR), при этом присоединенные к слотам демоны будут отвечать за обновление сети и инициировать выполнение правил. Однако из-за трудностей, возникающих с реализацией распределенного алгоритма Rete, в инструментарии реализован несколько менее общий алгоритм, разрешающий использование слотов в рамках одного

[1]Как будет показано ниже, система допускает также смешанные режимы выполнения, при которых данные с сервера подгружаются динамически.

фрейм-мира, с наследованием только по одному направлению. В этом случае правила целиком ассоциируются с некоторым слотом одного фрейма, а для запоминания срабатывания вводится специальная структура данных — **контекст активации**, роль которого также заключается в том, чтобы отслеживать определенность всех слотов фреймов, участвующих в левой (условной) части правила прямого вывода[2].

6.4. Реализация распределенного вывода

Будем называть **инсталляцией** ядро библиотеки JULIA (поддерживающее сетевые расширения) с загруженным фрейм-миром, работающее на некотором узле сети. Инсталляция включает в себя некоторую фреймовую иерархию (субиерархию), т.е. набор фреймов с уникальным именованием, связанных отношением наследования. С фреймами связаны процедуры-демоны и процедуры-запросы, соответствующие некоторой базе знаний.

В каждой из локальных субиерархий возможно использование двух видов ссылок на **удаленные фреймы** (т.е. на фреймы, принадлежащие фрейм-мирам других инсталляций): **статической** и **мобильной**. Эти виды ссылок соответствуют описанным в разделе 4.3 операциям ⧫ и ◊ и определяют, будет ли для вычисления ссылки применяться удаленный вызов или включение.

6.4.1. Распределенный вывод на основе удаленного вызова (invokation)

Когда в некотором фрейм-мире необходимо обратиться к удаленному фрейму по статической ссылке, создается т.н. **фрейм-посредник** (proxy frame), который дублирует атрибуты соответствующего удаленного фрейма, а в случае необходимости получения значения некоторого атрибута производит удаленный вызов соответствующих методов, реализованных интерфейсом удаленного фрейма, что может приводить к процессу обратного логического удаленного вывода на удаленном узле (см. рис. 4). Аналогично при присваивании значения слоту производится удаленный вызов, который ведет к прямому выводу на удаленной машине. Дублирование и локальное хранение значений атрибутов позволяет обеспечить кэширование значений во избежание их повторной передачи по сети.

Существует два вида статических ссылок на удаленные фреймы: **ранние статические ссылки** и **динамические ссылки**. В первом случае (т.н. **раннее связывание**) еще на этапе процессирования исходного текста на JFMDL создается фрейм-посредник, который жестко связывается со ссылкой на удаленный фрейм посредством ссылок используемого механизма удаленного взаимодействия (CORBA, RMI или др.). В случае динамической ссылки (т.н. **позднее связывание**) фрейм-посредник создается только в процессе выполнения и динамически находит удаленный фрейм используя доступные средства именования (например, COS Naming Services).

[2]таким образом, контекст активации выполняет роль α-памяти в алгоритме Rete, а роль β-памяти может выполняться самими деревьями условной части выражения при использовании кеширующих вычислений с запоминанием промежуточных значений.

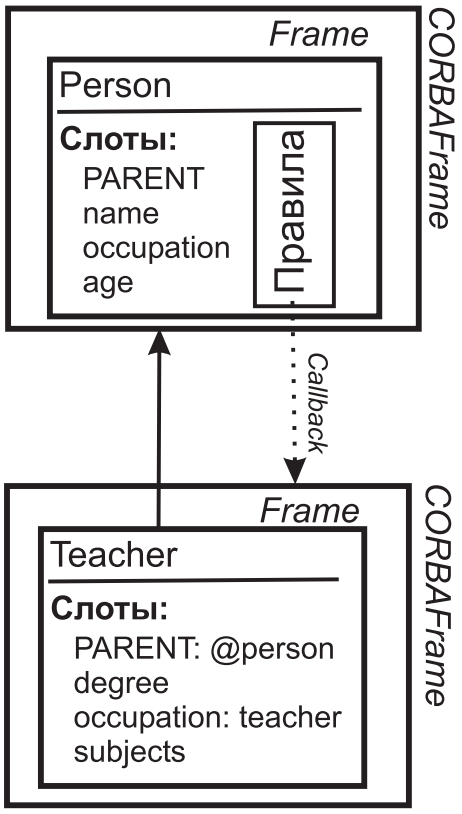


Рис. 4. Удаленный вывод с использованием фрейма-посредника

В распределенной фреймовой иерархии различают два случая использования удаленных ссылок: **наследование** от удаленного фрейма и **агрегация** удаленного фрейма. В свою очередь, в случае агрегации удаленный фрейм может использоваться в посылках правила прямого вывода, либо в выражении. Последний случай не представляет особого интереса, так как мало чем отличается от удаленного вызова с целью получения значения слота — следует однако иметь в виду, что такой удаленный вызов зачастую приводит к процессу обратного вывода. Если в случае обратного вывода встречаются правила, запрашивающие значения у пользователя — осуществляется обратный вызов (callback) вызвавшей инсталляции, которая и осуществляет интерфейс с пользователем. Таким образом, в стандартном случае исходные данные запрашиваются на одной ЭВМ (что обычно и требуется); однако всегда возможно использовать процедуры, которые будут осуществлять распределенный сбор данных, что характерно, например, для задач распределенного интеллектуального контроля и управления.

В случае наследования от удаленного фрейма обычно происходит переход процесса обратного вывода через границы инсталляции. В этом случае в алгоритме обратного вывода производится вызов процедуры `get` для фрейма-посредника, который, в свою очередь, производит удаленный вызов соответствующей процедуры на другой инсталляции, передавая в качестве базового фрейма ссылку на текущий фрейм. В этом случае в удаленной инсталляции также создается фрейм-посредник, который связывается с базовым фреймом для данного вывода. Соответственно, процессом вывода управляют динамические правила, расположенные на удаленной ЭВМ, но значения атрибутов присваиваются исходному базовому фрейму в процессе обратного удаленного вызова.

В данном случае важную роль играет тот факт, что фрейм-наследник не должен обладать никакими данными относительно правил, соответствующих родительскому фрейму — достаточно лишь удаленной ссылки. Это свойство мы назовем **прозрачностью удаленного наследования**.

Распределенный прямой вывод реализуется несколько сложнее и накладывает некоторые ограничения, из-за чего практическая его ценность весьма ограничивается. В частности, в посылках правил прямого вывода допускаются только ранние статические ссылки на удаленные фреймы, что обусловлено необходимостью построения сети ссылок до начала вывода. При добавлении правила прямого вывода с удаленным слотом в посылке во фрейм-мир некоторой инсталляции производится запрос к удаленной инсталляции на добавление процедуры-демона к соответствующему удаленному фрейму. При присвоении этому фрейму значения эта процедура

осуществляет удаленный вызов, что приводит к попытке применения правила, осуществляемой обычным образом.

Модифицированный алгоритм прямого вывода оказывается чрезвычайно удобным для применения в распределенном случае, так как в нем не производится построения полной сети Rete , и тем самым не требуется централизованного построения единой сети, включающей в себя ссылки на все используемые в кластере правила вывода. Основным преимуществом алгоритма является возможность распределенного хранения правил прямого вывода, что достигается за счет некоторого снижения производительности. Тем не менее в данном случае ключевой является возможность обеспечения прозрачности удаленного наследования, которая достигается как раз за счет распределенного хранения правил.

6.4.2. Распределенный вывод на основе включения (inclusion)

Удаленный вызов на основе включения сводится к локальному вызову в рамках одной виртуальной машины Java за счет создания копии удаленного фрейм-мира на локальной ЭВМ. Как только в процессе вывода возникает необходимость вычисления мобильной удаленной ссылки, инструментарий, в случае необходимости, загружает соответствующий фрейм-мир с удаленного узла (как правило, в виде внутреннего сериализованного представления) и создает его локальную копию, после чего работа с экземплярами фреймов и слотов этого фрейм-мира ведется в рамках витруальной машины как с локальными объектами. Два и более фрейм-миров могут сосуществовать в одной виртуальной машине Java, каждый имея свой набор управляющих объектов (`World`, `ClusterManager` и др.). При присоединении удаленного фрейм-мира выполняется набор вспомогательных операций, корректирующих ряд внутренних структур соответствующего объекта `World`, в частности, для присоединенного фрейм-мира устанавливается тот же объект журналирования `Log`, интерфейс для диалога с пользователем `Asker` и другие свойства, что и для исходного фрейм-мира.

6.5. Средства обеспечения онтологической прозрачности и инкапсуляции

При создании базы знаний, лежащей в основе некоторой субиерархии, во многих случаях ставят задачу ее повторного использования со стороны различных удаленных наследников. Для обеспечения удобства использования базы знаний необходимо применять в ней принципы, схожие с принципами модульного и объектно-ориентированного программирования. Среди наиболее важных принципов хотелось бы отметить следующие:

Онтологическая прозрачность, позволяющая пользователю получить максимум информации об онтологии, реализуемой рассматриваемой базой знаний.

Онтологическая инкапсуляция, обеспечивающая внешнюю целостность концептов онтологии за счет сокрытия от внешнего по отношению к базе знаний наблюдателя несущественных де-

талей реализации и атрибутов и концентрации его внимания на существенных.

Рутинность, означающая эквивалентность онтологического описания вне зависимости от предыстории, т.е. от конкретного вызова соответствующей этому описанию инсталляции.

Для повышения **онтологической прозрачности** инструментарий содержит средства поддержания более богатого онтологического описания модели. В частности, каждый фрейм, слот, правило или ограничение могут аннотироваться на естественном языке, чтобы при последующем их просмотре было понятно назначение соответствующих элементов модели. Такие текстовые описания вводятся при помощи команды `DESC`, и также могут использоваться для работы механизма *как-* и *почему-*объяснений.

Каждый фрейм или слот фрейма могут быть помечены как `private` в том случае, если создатель модели предполагает, что данный слот/фрейм не представляет интереса как атрибут онтологии, а служит для некоторых внутренних целей. Браузеры онтологий могут учитывать этот флаг для сокрытия несущественных атрибутов модели, тем самым реализуя принцип **онтологической инкапсуляции**.

Рутинность применительно к интеллектуальным системам заслуживает особого внимания, так как одной из характеристик интеллектуальной системы является способность к обучению. Поэтому, с точки зрения таких систем может быть желательным, чтобы некоторая база знаний изменяла свое поведение по мере ее использования, таким образом обучаясь по мере своего существования. В этом случае нельзя гарантировать эквивалентность поведения системы с точки зрения формальной семантики, равно как и эквивалентность содержащихся в базе знаний сведений, так как обучение подразумевает модификацию базы знаний. Однако попрежнему хотелось бы иметь свойство, аналогичное рутинности (которое можно назвать *семиотической рутинностью*), означающее, что база знаний по прежнему *корректно* (с точки зрения некоторого неформального критерия) отражает предметную область.

В инструментарии JULIA при использовании мобильного наследования достигается полная независимость от предыстории, так как каждый раз используется одно и то же внешнее сериализованное представление. Что касается удаленного вызова, то тут возможно нарушение принципа рутинности за счет потенциальной возможности модификации всех открытых слотов. Более того, в некоторых случаях такая функциональность является полезной, так как позволяет упростить реализацию базы знаний на стороне клиента и минимизировать трафик за счет хранения определенных данных в инсталляции на стороне сервера. Тем не менее, в таких случаях хотелось бы изолировать других клиентов от эффекта изменения базы знаний. Обеспечение этого реализуется в инструментарии двумя путями:

- Введением и использованием еще одного уровня защиты атрибутов (`protected`), запрещающего модификацию их значений со стороны внешних запросов. Это позволяет минимизировать расходы памяти на стороне сервера (для всех внешних запросов используется один и тот же фрейм-мир), а также обеспечить полную рутинность, запретив возможность модификации родительской иерархии.

- Созданием для каждого клиента удаленной инсталляции своей копии фрейм-мира на удаленном сервере. Такое создание может быть привязано к команде именования `WORLD ... AS ... SEPARATE`. Такое решение требует хранения на сервере соответствующего числу клиентов количества фрейм-миров.

7. Заключение

Предложенная архитектура построения распределенных интеллектуальных систем на базе продукционно-фреймового представления знаний, реализованная в инструментарии JULIA, обладает следующими преимуществами:

- Фреймовое представление знаний предоставляет естественный способ кластеризации знаний, в особенности динамических правил прямого и обратного вывода, вокруг соответствующих фреймов в виде процедур-демонов и процедур-запросов, что в свою очередь обеспечивает естественное распределение знаний между различными узлами. Представление знаний в виде иерархических структур находит отражение в виде моделей онтологий [1, 2], и модель распределенной фреймовой иерархии представляет собой один из способов реализации таксономических онтологий, совмещенных с традиционной моделью логического вывода и представления знаний. Методология создания распределенных баз знаний на основе распределенной фреймовой модели заслуживает более подробного изучения, в частности, с позиций адаптации существующих объектных методологий проектирования программных систем.
- Фреймовое представление знаний очень похоже на традиционные в настоящее время объектный и компонентный подходы, что позволяет использовать фреймовую модель как некоторый “общий знаменатель” при создании гибридных систем, сочетающих декларативные знания и императивные компоненты. Таким образом, объекты и компоненты открытых систем в форме Java-классов, (Enterprise) JavaBeans, CORBA- и COM-объектов могут быть представлены как фреймы в единой иерархии, и наоборот, любой фрейм иерархии может использоваться как объект, вызываемый из внешней программной системы. С этой точки зрения язык представления знаний может использоваться для интеллектуального скриптинга над императивными компонентами корпоративной бизнес-логики.
- Фреймы могут эффективно использоваться для доступа к реляционным базам данных, а также к другим типам структуризованной информации (например, для анализа сетей веб-страниц [2] и др.).

Таким образом, в инструментарии JULIA реализован целый спектр подходов к созданию гибридных распределенных интеллектуальных систем, что позволяет эффективно использовать его в различных приложениях, ориентированных на распределенное накопление и использование знаний. Безусловно, инструментарий может эффективно использоваться и для создания локальных (функционирующих на одной ЭВМ) эксперт-

ных систем, распределенных систем масштаба предприятия, функционирующих на единой шине CORBA, распределенных систем информационно-интеллектуальной поддержки виртуальных корпораций, гетерогенных Интернет-приложений и др. В частности, имеется положительный опыт использования инструментария для решения задач медицинской диагностики с участием "виртуального консилиума" баз знаний, распределенно создаваемых и поддерживаемых группой специалистов, для создания систем дистанционного компьютерного обучения, основанных на комбинации единой онтологии по методам преподавания с онтологической структурой предметной области и заданием динамики учебного процесса и интеллектуальной навигации по гипертекстовому курсу при помощи продукционной базы знаний, для реализации онтологической системы поиска аннотированных знаниями интернет-ресурсов на основе индексного семейства баз знаний в продукционно-фреймовом представлении, в системах удаленной консультации в среде Интернет, а также в решении ряда других задач. Инструмантарий внедрен в учебный процесс Московского авиационного института (технического университета) в качестве базового учебного средства по курсу "Интеллектуальные и экспертные системы", в также в лечебный процесс отделения урологии Государственной клинической больницы им.С.П.Боткина, где функционирует созданная на базе инструментария интеллектуально-учетная система диагностики и выработки тактики лечения больных заболеваниями предстательной железы.

В то время как агентный подход в настоящее время находится в стадии бурного развития, нам кажется, что альтернативный подход, основанный на классическом синхронном логическом выводе и явном представлении знаний будет дополнять агентные системы, в ряде случаев оказываясь более удобным для применения. Кроме того, создание расширяемой гибридной многоплатформенной библиотеки для построения интеллектуальных систем, основанной на открытых стандартах (XML, CORBA, HTTP, KQML), само по себе является весьма полезной задачей, находящей множество применений на практике.

Литература

[1] Gruber T. R., A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2), 1993. – pp. 199-220.

[2] Гаврилова Т.А., Хорошевский В.Ф. Базы знаний интеллектуальных систем. — СПб.: Питер, 2000.

[3] Soshnikov D. An Approach for Creating Distributed Intelligent Systems. In *Proceedings of the 1st International Workshop on Computer Science and Information Technologies*, Moscow, Mephi Publishing, 1998. pp. 129–134.

[4] Soshnikov D. Software Toolkit for Buiding Embedded and Distributed Knowledge-Based Systems. In *Proceedings of the 2nd International Workshop on Computer Science and Information Technologies, Vol.1*, USATU Publishing, Ufa, 2000. pp. 103–111. (Депонировано в `arXiv:cs.AI/0106054`).

[5] Сошников Д.В. Инструментарий JULIA для построения распределенных интеллектуальных систем на основе продукционно-фреймового представления знаний. Электронный журнал "Труды МАИ" — М.: МАИ, 2002, №7. `http://www.mai.ru/projects/mai_works/index.htm`

[6] Вольфенгаген В.Э. Конструкции языков программирования. Приемы описания. — М.: АО "Центр ЮрИнфоР", 2001.

[7] Forgy C.L. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, V.19, №1, 1982. p.17–37.

[8] Tarski, A. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 1955, Vol.5. pp 285–309.

[9] Kifer M., Lausen G., Wu J. Logical Foundations of Object-Oriented and Frame-Based Languages. Tech. Report 90/14, Department of Computer Science, State University of New York at Stony Brook, June 1990.

[10] Girratano J. , Riley G. Expert Systems: Principles and Programming. — PWS Publishing Company, Boston, 1993. (2nd Ed.)

[11] Fitting M. Fixpoint Semantics for Logic Programming: A Survey. Elsevier Preprint, 1996.

[12] Seng Wai Loke, Adding Logic Programming Behaviour to the World Wide Web, PhD Thesis, Department of Computer Science, The University of Melbourne, Australia, 1998.

[13] Lesser, V.R. An Overview of DAI: Viewing Distributed AI as Distributed Search, *Journal of Japanese Society for AI*, Vol. 5, No. 4, 1990.

[14] Ramakrishnan K., Ullman, J. A Survey of Research on Deductive Database Systems, *Journal of Logic Programming*, 23(2), 1995. – pp. 125–149.

[15] Hanson N.H.,Widom J. An Overview of Production Rules in Database Systems. *Knowledge Engineering Review*, Vol.8, No.2, 1993. pp.121–143.

Сошников Дмитрий Валерьевич

Логический вывод на основе удаленного вызова и включения в системах с распределенной фреймовой иерархией

Научное издание

Под редакцией к.ф.-м.н., доц. *Зайцева В.Е.*
Компьютерная верстка автора

E-mail для связи с автором: `dmitri@soshnikov.com`

Подписано в печать 28.01.2002 г. Формат 60x84 1/16
Печ.л. 3. Тираж 100 экз.

ЗАО «Издательское предприятие «Вузовская книга»
125871, Москва, Волоколамское шоссе, д.4
Тел./факс 158-0235
E-mail: vbook@mai.ru

ISBN 5-9502-0025-X