ПРОГРАММИРОВАНИЕ, 2011, № 3, с. 50–64

ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 681.3.06

МОДЕЛИРОВАНИЕ ОПЕРАЦИОННОЙ СЕМАНТИКИ МАШИННЫХ ИНСТРУКЦИЙ

© **2011 г.** В.А. Падарян, М.А. Соловьев, А.И. Кононов
Институт системного программирования РАН
109004 г. Москва, ул. Солженицына, 25
E-mail: vartan@ispras.ru, eyescream@ispras.ru, extreme@ispras.ru
Поступила в редакцию 03.09.2010 г.

В работе предлагается модель, позволяющая описывать операционную семантику машинных инструкций для широкого класса целевых архитектур. Особенностью модели является то, что она предназначена для обратного по сравнению с классической компиляторной задачей тракта преобразований, но в то же время модель позволяет выполнять над ней различные оптимизирующие преобразования. Для описания целевой машины применяются внешние спецификации. Рассмотрена прототипная подсистема интерпретации модели.

1. ВВЕДЕНИЕ

Анализ бинарного кода, нацеленный на восстановление алгоритмов и повышение уровня их представления, включает в себя большое количество этапов. Часть из используемых этапов анализа применяется последовательно, с целью выделения интересующей части программы. Другие используются итеративно для восстановления отдельных аспектов семантики.

В настоящее время актуальна задача анализа кода, снабженного механизмами, препятствующими как отладке, так и статическому анализу. Примерами подобных механизмов являются различные запутывающие преобразования [1]. Значительная часть преобразований такого рода нацелена на противодействие статическому анализу. Так, преобразование „диспетчер“ [2] не оказывает заметного усложняющего влияния при применении динамического анализа. Кроме того, использование динамического анализа в виде снятия трассы работы целевого процессора (стенда) и последующего ее анализа позволяет анализировать не только пользовательский код, но и код системный, например, драйверов.

В системе динамического анализа кода TrEx [3] применяется подход с использованием полносистемных симуляторов для сбора трасс. Среда поддерживает несколько целевых процессорных архитектур (в том числе Intel 64 и MIPS64) за счёт использования прослойки абстракции архитектуры. На данный момент указанная прослойка обладает возможностями декодирования инструкций в машинно-независимый вид и построения частичных графов зависимостей по данным отдельных инструкций. Указанные особенности позволяют при реализации алгоритмов анализа, основанных на анализе зависимостей, абстрагироваться от целевой машины, что, в свою очередь, значительно упрощает разработку.

Возможность построения декомпозиции на зависимости позволяет проводить более точно различные виды анализа, в том числе слайсинг трассы [4, 5]. Тем не менее, вопрос повышения уровня представления выделенного алгоритма не может быть решен без возможности анализа семантики составляющих алгоритм инструкций по отдельности и в совокупности.

В настоящей работе предлагается модель, пригодная для описания операционной семантики машинных инструкций. Описывается алгоритм приведения бинарной машинной инструкции к модельному виду. Кроме того, рассматривается применение описываемой

модели к задаче интерпретации инструкций.

Прежде чем перейти к изложению предлагаемой модели, необходимо указать требования, исходя из которых принимались решения при ее построении.

Модель должна позволять описывать операционную семантику как пользовательских, так и системных инструкций современных процессорных архитектур. Важно поддержать различные по своей природе процессорные архитектуры, как CISC, так и RISC. В набор архитектур, рассматривавшихся при построении модели, входят Intel 64, MIPS64, PowerPC, ARM v6, SPARC и Motorola m68k. Описание семантики инструкций этих архитектур должно быть достаточным для моделирования всех эффектов выполнения инструкции, в том числе побочных (влияние на слово состояния машины).

Необходимо обеспечить такой уровень модели, который позволял бы без существенных дополнительных затрат применять различные виды компиляторных оптимизаций, таких как удаление общих подвыражений или удаление мертвого кода. Это требование вызвано необходимостью противодействия тем видам запутывающих преобразований, которые намеренно усложняют выражения, входящие в операторы программы, или граф потока управления за счет введения дублирующих или нереализуемых путей. Наличие данных о семантике машинных инструкций, составляющих алгоритм, является необходимым требованием при решении этих задач. Тем самым, модель должна быть приближена к промежуточным представлениям, применяемым в современных компиляторах (SSA, трехадресный код и так далее).

Наконец, необходимо обеспечить возможность интерпретации модельного представления, а именно указать, каким образом из состояния машины перед выполнением моделируемой инструкции можно получить ее состояние после. Тем самым может быть обеспечен режим работы системы анализа, сходный с работой отладчика.

Дальнейшее повествование построено следующим образом. В разделе 2 приводится обзор связанных работ и выделяются те их особенности, которые могут быть использованы при решении задачи моделирования семантики инструкций. В разделе 3 описывается построенная модель. Указываются отдельные объекты модели и их взаимосвязь. В разделе 4 дается краткое описание способа спецификации целевой машины и указывается способ построения модели инструкции при наличии такой спецификации. В разделе 5 кратко описывается прототип интерпретатора модели. В разделе 6 дается заключение и указываются направления дальнейших работ.

2. ОБЗОР РАБОТ

Решаемая задача напрямую связана с двумя сферами компиляторных технологий: собственно построением компиляторов и бинарной трансляцией.

С точки зрения задачи построения компиляторов, пересечение с настоящей работой находится на уровне промежуточного представления. Используя классификацию из [6], можно охарактеризовать это промежуточное представление как представление низкого уровня (LIR). В самом деле, исходными данными для моделирования являются машинные инструкции, оперирующие с конкретными регистрами и адресами в памяти (в противовес виртуальным регистрам или переменным). Несмотря на то, что такое представление может быть механически повышено до представления среднего уровня (MIR) за счёт того, что является его подмножеством, подобное механическое повышение бессмысленно, так как ведет к повышению уровня представления лишь формально.

Одной из широко распространенных форм представления низкого уровня является форма RTL (register transfer list), применяемая, с различными модификациями, в большом количестве компиляторных продуктов, например, в открытом компиляторе GNU GCC [7]. В GCC также используется представление среднего уровня GIMPLE.

Следует учитывать, однако, что поставленная задача, в противовес задаче компиляции, предполагает обратный тракт: от машинных инструкций к более высоким представлениям. Такой тракт встречается в задаче бинарной трансляции, когда машинные инструкции одной

процессорной архитектуры сначала повышаются до некоторого модельного представления, а затем используются для синтеза машинных инструкций другой архитектуры.

Частным случаем можно считать среды для динамического инструментирования, в которых исходная и целевая архитектуры совпадают, например, Valgrind [8] и iDNA [9].

В работах [10, 11] описываются соответственно система статической и динамической бинарной трансляции. Обе системы среди поддерживаемых архитектур указывают и CISC-архитектуру Intel 64. В качестве промежуточного представления используется расширенная форма RTL, называемая авторами H-RTL. Следует отметить, что в данных работах рассматривается вопрос трансляции лишь пользовательских инструкций.

Среда Valgrind [8] использует в качестве промежуточного представления виртуальную RISC-машину с достаточно малым набором операций. При этом регистровый файл и пространство памяти целевой машины являются внешними по отношению к виртуальной машине, то есть трансляции не подвергаются. Valgrind не поддерживает системные инструкции. Кроме того, с точки зрения расширяемости среда Valgrind достаточно неудобна: трансляция в промежуточное представление и последующий синтез закодированы напрямую, а не при помощи внешних спецификаций. Кроме того, этапы декодирования инструкции и формирования ее модели не отделены друг от друга, что вносит дополнительные сложности.

Среди существующих систем бинарной трансляции реальный интерес в контексте данной работы представляют лишь те, которые, во-первых, в том или ином виде используют спецификации для описания машин и, во-вторых, способны проводить трансляцию из и в CISC-архитектуры (в частности, Intel 64). Трансляция в инструкции CISC-машины необходима для обеспечения эффективного решения задачи интерпретации. В то время как первое требование реализовано в достаточном количестве рассмотренных работ, второе значительно ограничивает этот список.

Проведенный обзор работ позволяет сделать вывод о необходимости реализации своей модели операционной семантики машинных инструкций, не обладающей недостатками рассмотренных работ и удовлетворяющей заявленным требованиям. Предлагаемая модель представляет собой комбинацию возможностей рассмотренных систем с дополнением оригинальными особенностями. Без изменений может быть использована модель адресных пространств, используемая в близких видах в [8] и [3]. Во многом удовлетворяет заявленным требованиям модель, используемая в качестве промежуточного представления в [8], однако она должна быть дополнена возможностью подгрузки внешних спецификаций наподобие используемых в [11].

3. МОДЕЛЬ ОПЕРАЦИОННОЙ СЕМАНТИКИ

Перед изложением построенной модели необходимо обозначить некоторые сложности, с которыми приходится сталкиваться при моделировании современных наборов инструкций. В большей части указанные сложности относятся к CISC-архитектурам, в частности к Intel 64.

Первой сложностью является нетривиальный вид регистровых файлов современных процессоров. Так, регистры могут являться частями друг друга (Intel 64), располагаться в нескольких теневых наборах (MIPS64) или иметь организацию в виде регистровых окон (SPARC). Подобные идиомы в регистровом файле приводят к необходимости обеспечения достаточно гибкой его модели. Такой моделью может служить модель адресных пространств [3], используемая в TrEx при декомпозиции на зависимости и более подробно описанная далее. Кроме того, сходный способ описания регистрового файла используется и в среде Valgrind.

Второй сложностью, характерной для CISC-архитектур, является наличие слова состояния машины, изменение которого в качестве побочного эффекта входит в семантику многих инструкций. В соответствии с требованием пригодности модели для проведения различных анализов необходимо исключить неявное обновление слова состояния. С другой стороны, некоторые из статусных флагов Intel 64, например флаг четности PF,

сложно пересчитывать вручную (требуется как минимум 9 операций для обновления этого флага для 32-битного значения без применения таблиц пересчета). Таким образом, необходимо поддержать баланс между минимальным объемом побочных эффектов в операторах модели и объемом (с точки зрения числа примитивных операций) трансляций распространенных инструкций. Указав основные сложности, перейдем теперь к описанию предлагаемой модели. Общая схема модели приведена на рис. 1.

Модель семантики отдельной инструкции представляет собой упорядоченный набор *операторов*, описывающих действия, производимые процессором при выполнении данной инструкции. Отдельные операторы используются для применения *операций*, обращения в *адресные пространства* и ветвления. В качестве параметров и возвращаемых значений операций фигурируют *локальные переменные*. Осуществляется несложный контроль типов на уровне *атомов*.

Операторы читаются (при интерпретации – выполняются) последовательно. Область видимости локальных переменных ограничена единицей трансляции (то есть одной инструкцией). Следует пояснить, что в ходе дальнейших оптимизаций единица трансляции может быть расширена, однако первоначально производится построение модели для одной инструкции. Каждая из указанных сущностей описана подробно в следующих подразделах.

3.1. Атомы

Первой из концепций, вводимых в модели, являются *атомы*. Под атомом будем понимать машинный тип, то есть элементарный блок данных, которым могут оперировать машинные инструкции.

Примерами атомов будут являться 32-битные целые числа, числа с плавающей точкой двойной точности, SIMD-векторы и т. п. Характеристикой атома является размер блока данных, им описываемого.

Некоторая избыточность, вводимая атомами, оказывается полезной для обеспечения корректности построенных моделей. В самом деле, можно было в качестве характеристики блока машинных данных использовать только их размер, не помечая их никаким атомом. Тем не менее это чревато попыткой применения операции над числами с плавающей точкой к, например, целочисленным данным. Такая попытка при использовании атомов будет пресечена на этапе проверки модели. Кроме того, тегирование данных атомами позволяет иметь более читаемый вид промежуточного представления: аналитик может, глядя на атом данных, получить представление о их природе.

Договоримся для дальнейшего изложения под *i8, i16, i32, i64* понимать соответственно 8-, 16-, 32- и 64-битные целочисленные атомы.

3.2. Операции

Как и при любом применении операционного подхода к описанию семантики, необходимо определять примитивные *операции*, через которые будет описываться поведение целевой машины.

В соответствии с требованием о простоте модели возникает естественное требование к отсутствию побочных эффектов у операций. К сожалению, подобный подход приведет к необозримым по объему моделям инструкций Intel 64 из-за обновления большого количества флагов слова состояния. Компромиссом здесь будет являться несколько облегченное требование к побочным эффектам операций: операция не должна иметь побочных эффектов, кроме, возможно, чтения и записи некоторых флагов модельного *слова состояния*.

Следует отметить, что при трансляции достаточно крупных регионов программы (например, суперблоков) с применением оптимизаций, побочные эффекты, указанные явно, могут использоваться, так как весь излишний код будет исключен в результате оптимизации. Тем не менее, поскольку одной из требуемых возможностей является использование модели для работы системы в режиме отладки, от такого подхода приходится отказаться.

Модельное слово состояния представляет собой 16-битное значение, включающее флаги состояния Intel 64: AF, CF, OF, PF, SF, ZF. Как показывает практика, обновления этих флагов достаточно не только для моделирования

Рис. 1. Общая схема модели.

Intel 64, но и RISC-машин, таких как MIPS64 или ARM.

Операции явно специфицируют *i-маску* и *o-маску*, то есть набор флагов слова состояния, которые данная операция просматривает и изменяет. Это позволяет проводить некоторые виды анализа, например, слайсинг, без необходимости различения поведения отдельных операций, лишь за счет точной спецификации зависимостей по данным, возникающих в результате применения этих операций.

Объединяя все вышесказанное, приходим к следующим характеристикам операций: имя, возвращаемый атом, атомы параметров, i-маска и o-маска. Укажем некоторые из стандартных операций модели.

1. Операция *add.i32* представляет собой целочисленное сложение 32-битных целых. Атом результата этой операции – *i32*, она имеет два параметра, каждому из которых также соответствует атом *i32*. I-маска этой операции пуста, а o-маска включает в себя флаги AF (BCD-переполнение), CF (переполнение), OF (знаковое переполнение), PF (четность), SF (знак), ZF (ноль).

2. Операция *smask.c.i16* получает в качестве параметра 16-битное целое значение (*i16*) и возвращает также *i16*. Ее поведение таково: все сброшенные биты параметра копируются в сброшенном виде в результат, а выставленные биты заменяются на значение флага CF из слова состояния. Таким образом, i-маска этой операции состоит из флага CF, а o-маска пуста.

3.3. Адресные пространства

Как было указано выше, для решения проблемы сложно организованного регистрового файла может быть применена модель адресных пространств. С точки зрения этой модели все методы адресации, с которыми может работать машина (регистры, память, порты ввода-вывода), сводятся к единой схеме: каждое адресуемое данное представляется в виде ссылки на одно из адресных пространств и снабжается указателем в это адресное пространство.

Для пространств оперативной памяти такое отображение вводится естественным образом. Несложно расширить естественное представление и на пространство портов ввода-вывода Intel 64.

Регистровый файл отображается в адресное пространство следующим образом. Выбирается размер адреса, позволяющий вместить все адресуемые регистры машины, после чего регистры размещаются в строящемся пространстве в произвольном порядке, но с учетом наложений и пересечений. Рис. 2 иллюстрирует эту идею для небольшой части регистрового файла Intel 64.

При использовании такого подхода не только решается проблема сложной организации регистрового файла, но и вводится унифицированная модель адресуемых данных, что позволяет не рассматривать отдельно каждый из их видов.

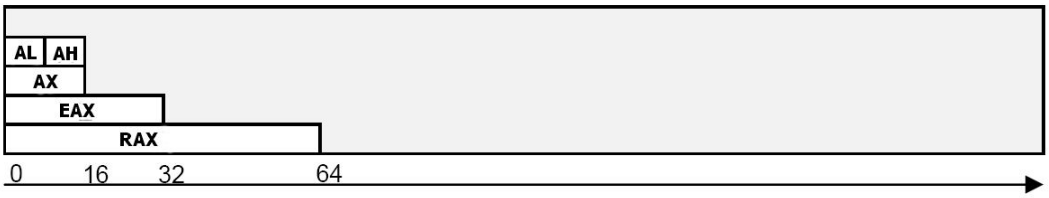

Рис. 2. Фрагмент адресного пространства регистров Intel 64.

Итак, в рамках предлагаемой модели адресные пространства машины включают в себя регистры, память, порты ввода-вывода и прочие подобные сущности и характеризуются атомом адреса (например, i64 для 64-битной памяти). Для упрощения работы с адресными пространствами могут быть введены *псевдонимы*, т. е. имена для некоторых последовательных блоков адресов адресного пространства. Псевдонимы наиболее применимы для обозначения регистров, но могут использоваться и в других адресных пространствах (например, в качестве символов для закрепленных архитектурно адресов в памяти, в частности, адресов обработчиков исключений).

В целях приведения дальнейших примеров будем считать, что задано адресное пространство регистров, называемое r.

3.4. *Локальные переменные*

При построении операторов в описываемой модели могут использоваться локальные переменные. Время жизни локальных переменных ограничена единицей трансляции. Так, при трансляции отдельных инструкций локальные переменные локальны в рамках данной инструкции и используются для хранения результатов промежуточных вычислений. Локальные переменные характеризуются своими номерами. В целях упрощения дальнейшего анализа к локальным переменным предъявляются требования SSA-формы.

Принимая во внимание достаточно простой с точки зрения внутренних передач управления вид машинных инструкций, можно не вводить оператор φ в модель. При определении локальной переменной с ней сопоставляется атом правой части соответствующего оператора.

Локальные переменные будем обозначать через $t.N$, где N – номер переменной.

3.5. *Операторы модели*

Модель предлагает следующие 8 операторов, в которые отображается операционная семантика транслируемых машинных инструкций.

1. Оператор NOP, не имеющий никакого эффекта.

2. Оператор INIT, инициализирующий локальную переменную константным значением. Константа задается в виде битовой последовательности и атома. Пример: INIT $(i16)\ t.3,\ 0x40\ \rightarrow\ (i16)\ t.3\ :=0x40$.

3. Оператор APPLY, применяющий одну из модельных операций. В качестве параметров и результата используются локальные переменные. Может быть указана дополнительная о-маска, ограничивающая влияние на модельное слово состояния. Итоговая о-маска рассматривается как побитовое „или“дополнительной маски и маски применяемой операции.
   Пример: APPLY $(i32)\ t.2, \mathrm{xor}.i32(t.0,\ t.1)\ \rightarrow\ (i32)\ t.2\ :=\mathrm{xor}.i32(t.0,\ t.1)$.
   Пример: APPLY $(i16)\ t.22, \mathrm{smask}.c.i16(t.4)\ \rightarrow\ (i16)\ t.22\ :=\mathrm{smask}.c.i16(t.4)$.

4. Оператор BRANCH, осуществляющий передачу управления. Различаются три вида передачи управления: безусловная, условная по маске и условная по сложному предикату. Во всех случаях указывается знаковое смещение для передачи управления (в качестве единицы адреса используется один модельный оператор). При использовании условной передачи управления по маске указываются два

16-битных числа: *rmask* и *lmask*. Передача осуществляется тогда и только тогда, когда все выставленные биты в *rmask* выставлены и в модельном слове состояния и все выставленные биты в *lmask* не выставлены в модельном слове состояния. Наконец, условная передача управления по сложному предикату проверяет один из предикатов A (беззнаковое „больше“), AE (беззнаковое „больше или равно“), B (беззнаковое „меньше“), BE (беззнаковое „меньше или равно“), G (знаковое „больше“), GE (знаковое „больше или равно“), L (знаковое „меньше“), LE (знаковое „меньше или равно“).
Пример: BRANCH.BE + 6.
Пример: BRANCH.MASKED +13, $0x0010$, $0x0000$.
Пример: BRANCH +7.

5. Оператор LOAD, производящий загрузку значения из одного из адресных пространств (например, значения регистра). Для оператора LOAD указывается адресное пространство, из которого осуществляется загрузка, атом загружаемого данного, локальная переменная с адресом данного и локальная переменная, принимающая результат. При этом атом локальной переменной адреса должен совпадать с атомом адреса пространства.
Пример: LOAD $(i16)$ $t.29$, $r : t.2$

6. Оператор STORE, записывающий значение в одно из адресных пространств. Для оператора STORE указывается адресное пространство для записи и локальные переменные адреса и значения. Атом локальной переменной адреса должен совпадать с атомом адреса пространства.
Пример: STORE $(i8)$ $r : t.0$, $t.21$.

7. Семейство операторов SPECIAL, указывающих на особое моделируемое состояние. Сюда относится оператор HALT (останов до возникновения внешнего события) и оператор TRAP (исключение или прерывание).
Пример: HALT.
Пример: TRAP 0.

8. Операторы ANNOTATION. Эти операторы игнорируются при интерпретации отдельной инструкции и не влияют на модельную семантику. Они используются для задания дополнительных аннотаций, которые могут быть использованы в отдельных видах анализа.

3.6. Пример

В качестве примера рассмотрим инструкцию ADD машины Intel 64, осуществляющую сложение значения регистра с константой и запись результата в тот же регистр. Конкретный вид рассматриваемой инструкции[1]: ADD EAX, 1.

Модель для этой инструкции приведена далее.

1. INIT (i16) t.-1, 0x0000; адрес регистра EAX в r;
2. INIT (i8) t.-2, 0x01; значение операнда-константы;
3. LOAD (i32) t.0, r:t.-1; загрузка значения EAX;
4. APPLY (i32) t.1, sx.i8.i32(t.-2); знаковое расширение[2] константы до $i32$;
5. APPLY (i32) t.2, add.i32(t.0, t.1); сложение;
6. STORE (i32) r:t.-1, t.2; сохранение результата в EAX;
7. INIT (i16) t.3, 0x40; адрес регистра EFLAGS в r;
8. LOAD (i16) t.4, r:t.3; загрузка значения EFLAGS;
9. INIT (i16) t.5, 0x08D5; подготовка маски для EFLAGS;
10. APPLY (i16) t.6, uf(t.4, t.5); обновление EFLAGS с учетом маски[3];
11. STORE (i16) r:t.3, t.6 ; сохранение EFLAGS

[1] Константа 1 в этой инструкции кодируется как 8-битное число со знаком.

[2] Операция знакового расширения 8-битного числа до 32-битного.

[3] Операция обновления слова состояния из модельного слова состояния с учетом параметра-маски.

Можно видеть, что большая часть модели представляет собой обновление слова состояния Intel 64. В RISC-архитектурах трансляция аналогичной инструкции сложения выглядит значительно компактнее: инструкция ADDIU машины MIPS64 при своем применении в виде ADDIU R1, R1, 1 транслируется в следующую модель.

1. INIT (i16) t.-1, 0x0010; адрес регистра R1 в r, первый операнд;
2. INIT (i16) t.-2, 0x0010; адрес регистра R1 в r, второй операнд;
3. INIT (i16) t.-3, 0x0001; значение операнда-константы;
4. LOAD (i64) t.0, r:t.-2; загрузка значения R1;
5. APPLY (i32) t.1, trunc.i64.i32(t.0); ограничение значения 32 битами;
6. APPLY (i32) t.2, sx.i16.i32(t.-3); знаковое расширение операнда-константы;
7. APPLY (i32) t.3, add.i32(t.1, t.2); сложение R1 и константы;
8. APPLY (i32) t.4, sx.i32.i64(t.3); расширение до размера регистра;
9. STORE (i64) r:t.-1, t.4; сохранение результата в R1.

4. СПЕЦИФИКАЦИЯ ЦЕЛЕВОЙ МАШИНЫ

Для обеспечения построения моделей конкретных инструкций целевой машины необходимо предоставить ее спецификацию.

Спецификация включает в себя описание атомов, операций, адресных пространств и подстановок (см. далее), используемых в данной машине. Поскольку подобное примеру из раздела 4.6 низкоуровневое описание модели неудобно с точки зрения поддержки, был разработан язык спецификаций, позволяющий записывать некоторые распространенные идиомы более компактно.

Процесс работы со спецификациями устроен следующим образом.

1. Разработчик предоставляет спецификацию машины в виде текстового файла на разработанном в рамках данной работы языке Pivot.
2. Спецификация проверяется на корректность и строится ее низкоуровневый вид в виде метамодели, позволяющей впоследствии генерировать модели семантики конкретных инструкций. Эта операция выполняется инструментом PivotC.
3. Результат работы PivotC – бинарный файл, называемый *архивом машины*. Для чтения этого архива разработана библиотека PivotPcma, использующаяся в дальнейших анализах и при интерпретации.

4.1. Структура спецификации

Спецификация представляет собой набор текстовых файлов на языке Pivot. Краткое описание этого языка дается в последующих подразделах.

Каждый файл представляет собой последовательность деклараций, описывающих один из аспектов специфицируемой машины. Сюда относятся декларации атомов, операций, адресных пространств и подстановок. Кроме того, файлы спецификаций могут включать (include) друг друга, тем самым позволяя использовать общие для нескольких машин декларации без дублирования их текста.

4.2. Атомы, операции и адресные пространства

Для спецификации атомов, операций и адресных пространств введен простой синтаксис, иллюстрируемый следующими примерами (все примеры взяты из спецификации машины 8086).

- **atom** i8 /.8
  Описывает атом $i8$ размером 8 бит (число до точки – байты, число после точки – биты, любое из них может быть опущено).
- **operation** i16 cons.i16(i8, i8)
  Описывает операцию cons.$i8$ с пустыми i- и o-масками.

- **operation** # and.#(#, #) **omask** „COPSZ“**with** i8, i16, i32, i64
  Описывает семейство операций *and.i8, and.i16, and.i32, and.i64* с пустой i-маской и o-маской, включающей CF, OF, PF, SF и ZF. Символ „#“ – подстановочный, вместо него подставляются все атомы из блока **with**.

- **space** io(i16)
  begin
  end
  Описывает адресное пространство с атомом адреса *i16*.

- **space** r(i16)
  begin
  alias ax 0 /.16
  alias cx +8 /.16
  ...
  alias ah ax + .8 /.8
  end
  Описывает адресное пространство с атомом адреса *i16* и задает в нем псевдонимы регистров. Адрес псевдонима может быть указан явно (ax), относительно предыдущего псевдонима (cx) или относительно одного из ранее определенных псевдонимов (ah).

4.3. Подстановки

Для сопоставления моделей операционной семантики с машинными инструкциями применяются *подстановки*. С одной стороны, подстановка указывает, каким инструкциям целевой машины она отвечает, то есть модели каких инструкций могут быть построены при помощи этой подстановки. С другой стороны, подстановка специфицирует сами эти модели. Одна подстановка может описывать семейство сходных инструкций за счет достаточно мощного синтаксиса описания с возможностью применения шаблонов и автоматического вывода атомов.

Подстановка состоит из двух частей: заголовка и тела.

Заголовок подстановки включает в себя сопоставляемую мнемонику (как строку) и условия на операнды машинной инструкции.

Изучение современных процессорных архитектур показывает, что операнды инструкций можно разбить на четыре класса:

1. константы, т. е. прямо закодированные значения;
2. простые выражения над регистрами, например, значение регистра со сдвигом в ARM (например, $R1 << 4$);
3. регистры и прямо закодированные адреса в памяти;
4. вычисляемые адреса в памяти.

Первый и второй классы, таким образом, объединяют в себе операнды, используемые только по значению. Такие операнды характеризуются значением (возможно, вычисляемым) и типом данных. Это означает, что с точки зрения вводимой модели можно рассматривать такие операнды как значения, записанные в некоторые локальные переменные. Тип данных при этом определяется атомом этой переменной.

Третий и четвертый классы с точки зрения модели могут быть представлены как ссылки в адресные пространства. При этом в одной из локальных переменных будет храниться адрес. Как и в случае с константами и выражениями над регистрами, тип адресуемых данных указывается в виде атома.

Условия на операнды в подстановках выписываются в соответствии с указанной классификацией. Операнд помечается как *val*, если он принадлежит к классу 1 или 2, и как *ref*, если он принадлежит к классу 3 или 4. Кроме того, для каждого операнда сообщается его атом.

При поиске подстановки для конкретной инструкции выбирается та, мнемоника которой совпадает с мнемоникой инструкции и условия на операнды которой выполнены для этой инструкции.

Тело подстановки представляет собой последовательность операторов языка спецификации, которые отображаются на операторы модели (NOP, INIT, APPLY, BRANCH, LOAD, STORE, SPECIAL, ANNOTATION). Перед любым из операторов может быть поставлена метка (синтаксис **label** *метка*).

Определены следующие операторы.

1. Оператор **nop**, отображаемый в NOP.

2. Оператор присваивания общего вида *lvalue* = *rvalue*. Части представляют собой одно из следующих выражений:

a. Ссылка на операнд $\$N$, например \$1. Если N-й операнд помечен как *val*, то выражение раскрывается в значение операнда (то есть в соответствующую локальную переменную). Если N-й операнд помечен как *ref*, то выражение раскрывается в чтение значения операнда (то есть в LOAD в правой части и STORE в левой части). Может встречаться только в правой части для *val*-операндов и в обеих частях для *ref*-операндов.

b. Ссылка на псевдоним *пространство:псевдоним*, например *r:eax*. В левой части раскрывается в соответствующий STORE, а в правой – в LOAD.

c. Константа (*атом*) *значение*, например (*i16*) 0x08D5. Может встречаться только в правой части.

d. Применение операции *операция*(*параметры...*) {**omask** маска}[4], например *add.i32*(\$1, \$2). Раскрывается в оператор APPLY. В качестве *параметров* могут использовать произвольные выражения из данного списка. Применение операции возможно только в правой части. Дополнительно может быть указана *маска*, ограничивающая изменение модельного слова состояния.

e. Локальная переменная **local** *имя*. Может встречаться в каждой из частей, но требуется единственность определения.

f. Обращение к адресному пространству *пространство* [*адрес*], где *адрес* – произвольное выражение. Требуется, чтобы атом выражения *адрес* совпадал с атомом адреса пространства.

3. Вычисление побочного эффекта выражения **discard** *rvalue*. Оператор вычисляет правую часть, но не сохраняет результат, лишь обновляя слово состояния.

4. Операторы **branch** *метка*, **branch**.*условие метка* и **branch**(*rmask, lmask*) *метка*. Раскрываются в соответствующие операторы BRANCH.

5. Операторы **halt** и **trap** *номер*, отображаемые в соответствующие SPECIAL-операторы.

Обращения к операндам выполняются следующим образом: операнду с номером N сопоставляется локальная переменная *t.-N*, в которой при построении модели конкретной инструкции для *val*-операндов содержится значение операнда, а для *ref*-операндов – значение адреса. Эти локальные переменные видны на уровне модели, но скрыты от разработчика спецификаций за счет выражений $\$N$.

Транслятор PivotC содержит подсистему вывода атомов, что позволяет указывать атомы лишь при константах и некоторых чтениях из адресных пространств.

Кроме того, транслятор позволяет применять шаблоны при описании подстановок: как и в случае с операциями в качестве атома операнда можно указать символ „#“. В этом случае вместо него будут последовательно подставлены все атомы, перечисленные в блоке **with**. Пример заголовка такой подстановки: **match** „IN“ **ref** #, **val** i8 **with** i8, i16.

Символ „#“может быть также указан и при применении операции в операторах тела подстановки. В этом случае PivotC выберет ту операцию соответствующего семейства, которая подходит по сигнатуре. В качестве еще одного упрощения помимо классов *val* и *ref* для операторов водится псевдо-класс *auto*, который подставляется и как *val*, и как *ref*. При этом если подстановка в качестве *val* невозможна (из-за наличия ссылки на операнд в левой части оператора присваивания), будет подставлен только *ref*-вариант.

Описанный язык позволяет достаточно быстро составлять спецификации машинных инструкций. Встроенные средства вывода атомов избавляют от необходимости указывать их вручную. Возможность шаблонного задания подстановок значительно облегчает их спецификацию для CISC-архитектур.

[4] Фигурные скобки обозначают необязательную часть.

4.4. *Примеры*

В качестве первого примера рассмотрим спецификацию подстановки для инструкции NOR машины MIPS64, выполняющую операцию «не-или» над значениями второго и третьего операндов-регистров и сохраняющую результат в первый операнд-регистр.

match „NOR“ **ref** i64, **ref** i64, **ref** i64
begin
$1 = not.i64(or.i64($2, $3))[5]
end

Легко видеть, что построенная спецификация достаточно проста и практически «слово в слово» повторяет описание поведения инструкции из документации по процессору MIPS64. Такая ситуация характерна для RISC-архитектур в целом.

В качестве примера набора подстановок для CISC-инструкции рассмотрим инструкцию сложения ADD машины Intel 64. Документация на Intel 64 указывает 19 различных кодировок этой инструкции. Использование шаблонов и ключевого слова *auto* позволяет свести эти 19 случаев к трем подстановкам.

match „ADD“ **ref** #, **auto** # **with** i8, i16, i32, i64
begin
$1 = add.#($1, $2); r:flags = uf(r:flags, (i16) 0x08D5)
end
match „ADD“ **ref** #, **val** i8 **with** i16, i32, i64
begin
$1 = add.#($1, sx.i8.#($2)); r:flags = uf(r:flags, i16) 0x08D5)
end
match „ADD“ **ref** i64, **val** i32
begin
$1 = add.i64($1, sx.i32.i64($2)); r:flags = uf(r:flags, (i16) 0x08D5)
end

В результате подстановки атомов из блоков **with** и за счет использования ключевого слова auto этими тремя подстановками будут покрыты все 19 случаев кодировки. Вторая подстановка для ADD при замене „#“ на *i32* будет раскрыта транслятором PivotC в модель, приведенную в 3.6, за исключением первых двух операторов. Эти операторы будут генерироваться отдельно для каждого конкретного набора операндов.

[5]Здесь и далее в примерах подстановок с целью упрощения читаемости опускается часть, связанная с продвижением указателя на текущую инструкцию (PC, EIP и т. п.).

5. ИНТЕРПРЕТАЦИЯ МОДЕЛИ

В рамках данной работы реализована прототипная система PivotI интерпретации модели операционной семантики инструкции. Поставленная перед системой интерпретации задача может быть формализована следующим образом.

Пусть задано полное состояние машины в некоторой точке времени (под полным состоянием машины понимаются значения всех ее регистров, в том числе регистра счетчика инструкций, а также содержимое памяти и иных пространств). Требуется получить полное состояние машины после выполнения *одной* инструкции по указателю счетчика инструкций (т.е. текущей инструкции).

Задачу будем решать в несколько этапов.

1. Необходимо провести декодирование текущей инструкции. Тем самым будет получена мнемоника этой инструкции (в виде строки), а также список ее операндов.

2. Среди подстановок для данной машины требуется найти ту, которая принимает данную инструкцию. При этом необходимо отобразить машинные типы на стандартные атомы (атомов *i8, i16, i32, i64* достаточно для подавляющего большинства пользовательских инструкций, за исключением FPU- и SIMD-инструкций, для которых определяются свои атомы, такие как *f64* и *v128*).

3. Сгенерировать код инициализации значений операндов-констант и адресов операндов-указателей (локальные переменные t.-N). В простых случаях этот код будет представлять собой набор операторов INIT; в более сложных (например, при использовании регистровых окон или косвенной адресации памяти) будет включать в себя соответствующую адресную арифметику.

4. Дополнить сгенерированный код операторами из подстановки, тем самым получив полную модель данной инструкции.

5. Выполнить модель.

Поскольку из перечисленных этапов лишь последний является неочевидным, рассмотрим его более подробно.

Обозначим сначала требования, которые интерпретатор будет предъявлять к знаниям о машине. Во-первых, для интерпретации необходимо знать смысл каждой из используемых в машине операций. Во-вторых, необходимо понимать природу используемых адресных пространств (запись в память и запись в порт ввода-вывода принципиально отличны с точки зрения влияния на состояние машины).

Первое из требований приводит к классическому замкнутому кругу при использовании операционного подхода к описанию семантики, так как фактически требует задания операционной семантики тех операций, которые, в свою очередь, используются для описания семантики машинных инструкций. Эту проблему можно разрешить, отобразив эти операции на набор инструкций машины, на которой выполняется сам интерпретатор, то есть синтезировав аналогичный native-код. Фактически, таким образом, интерпретатор производит бинарную трансляцию отдельных инструкций.

На данный момент прототип интерпретатора может использоваться на машине Intel 64; требуется доступность расширенных 64-битных регистров и машинных команд.

Второе требование предусматривает установку обработчиков чтения и записи на каждое из адресных пространств. Такие обработчики для некоторых адресных пространств будут тривиальными (например, для адресного пространства регистров общего назначения), а для некоторых (например, для пространства портов ввода-вывода) потребуют эмуляции набора оборудования машины. Впрочем, для интерпретации пользовательского кода эмуляция оборудования, как правило, не требуется.

Выполнение модели производится в рамках *контекста интерпретации*. Контекст интерпретации включает в себя следующие части.

1. Область локальных переменных. В этой области хранятся значения локальных переменных и их атомы.

2. Информация об операциях в виде списка соответствующих им встраиваемых функций.

3. Информация об адресных пространствах в виде списка соответствующих им обработчиков.

4. Область сгенерированного кода.

5. Информация о распределении локальных переменных по регистрам машины, на которой производится интерпретация.

Для решения подзадачи отображения операций модели на машинный код инструментального компьютера используются встраиваемые функции. К ним предъявляется перечень требований, закрепляющих, в частности, отсутствие пролога и эпилога, запрет использования стека и соглашение о распределении машинных регистров, через которые функция принимает значения параметров интерпретируемой операции и возвращает результат. Поскольку особенности реализации интерпретатора не являются основной темой данной работы, этот перечень требований опускается. Приведем лишь пример одной такой функции для операции сложения с обновлением слова состояния *add.i32*.

```
arithmetic_add_i32 PROC
  ;; Сложение.
  MOV R15D, R8D
  ADD R15D, R9D

  ;; Пересчет слова состояния.
  LAHF
  SETO AL
  SHL AL, 3
  AND BH, 0FCh
  OR BH, AL
  MOV BL, AH
arithmetic_add_i32 ENDP
```

Особенностью такого подхода является возможность простого расширения набора поддерживаемых операций за счет расширения множества функций. Это, в свою очередь, позволяет определять операции, упрощающие интерпретацию для конкретной машины.

Важным моментом является то, что функции не вызываются, а подставляются в генерируемый блок кода полностью, что позволяет избежать накладных расходов на передачу параметров и сброс конвейера.

Интерпретатор включает в себя набор встраиваемых функций для „стандартных“операций (целочисленная арифметика, битовая логика и т. д.). Похожим образом решается и вторая подзадача. Соответствующие блоки ассемблерного кода описывают процедуры чтения из адресных пространств и записи в адресные пространства. В интерпретатор включены так называемые „нулевой механизм“и „механизм по умолчанию“.

„Нулевой механизм“используется при отладке. Он игнорирует все записи в адресное пространство, к которому привязан, а при чтениях возвращает нули.

„Механизм по умолчанию“выделяет блок памяти размером со все адресное пространство и реализует чтения и записи естественным образом. Механизм применим для регистровых адресных пространств, но не применяется для пространств памяти из-за их относительно большого объема. Для реализации механизма работы с моделируемой оперативной памятью требуется более сложная логика, в частности из-за необходимости трансляции виртуальных адресов в физические.

Генерация машинного кода для интерпретируемой инструкции происходит в два этапа: этап генерации кода и этап расстановки переходов.

Первый этап последовательно просматривает операторы модели и генерирует машинный код по следующим правилам.

1. NOP: подставляется инструкция NOP.
2. INIT: подставляется инструкция MOV (или XOR для инициализации нулем).
3. MOVE: подставляется соответствующая функция.
4. BRANCH: подставляется инструкция перехода по данному условию, однако адрес перехода не заполняется, т. к. еще неизвестен.
5. LOAD: подставляется обработчик-механизм загрузки из соответствующего адресного пространства.
6. STORE: подставляется обработчик-механизм выгрузки в соответствующее адресное пространство.
7. SPECIAL: подставляется код, выставляющий специальное состояние интерпретатора.
8. ANNOTATION: игнорируются.

На втором этапе заполняются адреса всех переходов, сгенерированных в результате синтеза кода для операторов BRANCH.

При генерации кода используется область локальных переменных, где хранятся значения и атомы, связанные с ними. Производится несложное распределение регистров, сокращающее обращения к памяти этой области.

Такой подход в совокупности с минимальным количеством инструкций передачи управления в сгенерированном коде позволяет максимально использовать специфику Intel 64 (out-of-order issue и неявный параллелизм).

Сгенерированный код снабжается прологом и эпилогом и может быть выполнен для моделирования эффекта инструкции, что решает поставленную задачу. Особенностью является возможность одновременного существования произвольного количества контекстов интерпретации, содержащих трансляции отдельных инструкций, что позволяет, например, реализовывать пошаговую отладку многопоточных приложений.

К преимуществам применяемого в прототипной системе подхода следует отнести его простоту и гибкость. Очевидный недостаток – неэффективный генерируемый код. Эта проблема может быть решена при использовании вместо такой „наивной“генерации кода одного из алгоритмов динамического программирования для генерации оптимального кода (tree rewriting) в сочетании с набором оптимизаций. Поддержка

такого способа интерпретации планируется в дальнейшем.

Тем не менее, отказываться от предложенной схемы полностью также неразумно, т. к. в случаях, когда инструкция выполняется лишь один раз, данная схема будет показывать большую производительность, чем полноценный синтез за счет отсутствия расходов на анализ и оптимальную кодогенерацию. Окончательным вариантом интерпретации видится сочетание двух подходов (например, применение полновесного синтеза только для расширенных базовых блоков, лежащих на горячих путях).

6. ЗАКЛЮЧЕНИЕ

В настоящей работе выделены требования к модели операционной семантики машинных инструкций, выполнение которых необходимо для проведения широкого класса анализов, в совокупности нацеленных на восстановление алгоритмов. Предложена удовлетворяющая этим требованиям модель, опробованная в контексте широко распространенных CISC- и RISC-архитектур. Указаны способы упрощения построения спецификаций машин, описывающих приведение семантики инструкций к модельному виду. Эти способы нашли свою реализацию в средстве PivotC и библиотеке PivotPcma. С использованием указанной библиотеки разработан и опробован прототип системы интерпретации инструкций PivotI, общая схема функционирования которого также описана в работе.

В дальнейшем планируется полностью специфицировать распространенные машины (Intel 64, MIPS64, ARM, PowerPC) и на основании этих спецификаций сделать выводы о достаточном наборе модельных операций.

Планируется адаптировать алгоритм слайсинга трассы для использования с моделями операционной семантики инструкций трассы, что позволит отбирать в слайс не только машинные инструкции целиком, но и их части, выраженные в виде операторов модели.

Улучшение интерпретации за счет усложнения схемы кодогенерации также является важной задачей. Кроме того, планируется провести исследование возможности введения «границ интерпретации» на входе и выходе из известных библиотечных функций путем замены самих этих функций спецификациями влияния на состояние машины. Это позволит значительно ускорить интерпретацию и во многих случаях отказаться от необходимости эмуляции оборудования.

Введение данной модели позволяет вплотную подойти к решению важной и интересной задачи интерпретации нереализованных путей в трассе, которая может играть решающую роль при динамическом анализе. Неформально задача может быть поставлена следующим образом. Пусть дан набор трасс некоторой программы и известно, каким образом эти трассы связаны друг с другом (например, они снимались с одного и того же начального состояния, но с различным пользовательским вводом). Пусть в программе существует некоторое ветвление, одна из ветвей которого не реализовалась ни в одной из трасс. Выдвигается предположение о реализации этой ветви. Система, решающая задачу, должна предоставить условия на состояние машины для реализации ветви (в идеале эти условия должны быть распространены до пользовательского ввода), выбрать конкретный вид переменных, входящих в условие и в данных условиях провести интерпретацию ветви, тем самым получив дополнительную трассу выполнения, в которой интересующий путь уже будет реализован.

Кроме того, в список дальнейших работ включено также исследование возможности повышения уровня представления семантики, в частности, трансляция в LLVM [12].

СПИСОК ЛИТЕРАТУРЫ

1 *Collberg C., Thromborson C., Low D.* A taxonomy of obfuscating transformations // Technical report, University of Auckland, New Zealand.

2 *Wang C., Hill J., Knight J., Davidson J.* Software tamper resistance: obstructing static analysis of programs. // Technical report, University of Virginia, US. 2000.

3 *Падарян В.А., Гетьман А.И., Соловьев М.А.* Программная среда для динамического анализа бинарного кода. // Труды Института системного программирования, Т. 16, С. 51–72. 2009.

4 *Korel B., Laski J.* Dynamic program slicing. // Information Processing Letters, V. 29, I. 3, P. 155-163. 1988.

5 *Korel B., Smith R.* Slicing event traces of large software systems. // Automated and algorithmic debugging. 2000.

6 *Muchnick S.* Advanced compiler design and implementation. // Morgan Kaufmann Publishers Inc. 1997.

7 GNU Compiler Collection. http://gcc.gnu.org/.

8 *Nethercote N., Seward J.* Valgrind: a framework for heavyweight dynamic binary instrumentation. // Proceedings of ACM SIGPLAN 2007 conference on programming language design and implementation. San Diego, California, US. 2007.

9 *Bhansali S., Chen W., Jong S., Edwards A., Murray R., Drinic M., Mihocka D., Chau J.* Framework for Instruction-Level Tracing and Analysis of Program Executions. Microsoft Corporation. 2006.

10 *Cifuentes C., Emmerik M., Ramsey N., Lewis B.* Experience in the design, implementation and use of a retargetable static binary translation framework. 2002.

11 *Cifuentes C., Lewis B., Ung D.* Walkabout – a retargetable dynamic binary translation framework. 2002.

12 Low Level Virtual Machine. http://www.llvm.org/.