УДК 004.43 DOI: 10.17587/prin.8.345-353

В. А. Васенин, д-р физ.-мат. наук, проф., **М. А. Кривчиков**, канд. физ.-мат. наук, ст. науч. сотр., e-mail: maxim.krivchikov@gmail.com, МГУ имени М. В. Ломоносова

Методы промежуточного представления программ

Представлена классификация промежуточных представлений программ, написанных на различных языках программирования. Такие представления используются на практике в задачах трансляции программ, разработанных на нескольких языках. Основными критериями классификации являются синтаксические характеристики представления (являются ли частью представления нетривиальные синтаксические конструкции), способ описания потока исполнения (синтаксические конструкции, графы или байт-код), а также типизация элементов представления (сохраняются ли типы выражений исходного языка программирования в промежуточном представлении; какова степень соответствия типов в промежуточном представлении и типов в исходном языке программирования). С позиций предлагаемой классификации представлено современное состояние исследований в области промежуточных представлений программ.

Ключевые слова: *языки программирования, промежуточное представление, трансляция программ, формальная семантика программ, классификация, обзор*

Введение

Понятие промежуточного представления программы используется, в первую очередь, в рамках теории трансляции программ. При этом известны примеры применения промежуточных представлений как в задачах совместной трансляции программ, написанных с использованием нескольких языков программирования, так и для более удобного представления программ, написанных на одном языке, на различных стадиях их трансляции.

В настоящей статье представлена классификация и дана характеристика современному состоянию исследований в области промежуточных представлений программ на высоком уровне их абстракции (независимости) от аппаратной платформы, на которой они могут исполняться. Представленный в статье краткий обзор выполнен в контексте решения задачи разработки промежуточного представления для описания высокоуровневой формальной семантики и верификации функциональных свойств программ, написанных на нескольких языках программирования, в первую очередь, предметно-ориентированных. Основными предметами настоящего обзора и анализа являются подходы к формальному (математическому) описанию промежуточных представлений программ, а также к описанию в терминах представлений различных особенностей семантики языков программирования. В статье не рассмотрены промежуточные представления с узкой областью применения, в том числе ориентированные исключительно на оптимизацию кода программ, распараллеливание и на низкоуровневое описание асинхронных систем. Кроме того, не ставится цель получения исчерпывающего обзора, поскольку в настоящее время в той или иной степени промежуточные представления используются практически в любой реализации языка программирования. Целью настоящей статьи является систематизация и анализ уже существующих решений на рассматриваемом направлении и выделение особенностей промежуточного представления для различных категорий языков программирования.

Определения основных терминов, которые встречаются в тексте настоящей статьи, даны далее.

Язык промежуточного представления программ — частный случай языка программирования, который используется в качестве интерфейса взаимодействия между определенными стадиями трансляции программы. Промежуточное представление программ не предназначено для использования человеком. Как правило, синтаксическое описание промежуточного представления ограничивается абстрактным синтаксическим деревом или байт-кодом. Далее для краткости вместо термина "язык программирования" будем использовать "язык", а в качестве сокращения для термина "язык промежуточного представления программ" — "представление".

Байт-код — форма промежуточного представления программ, в которой код программы задан в виде линейной последовательности инструкций. Такая линейная последовательность кодируется в двоичном виде, аналогично машинному коду.

Базовый блок — последовательность инструкций программы, которая имеет одну точку входа, одну точку выхода и не содержит инструкций передачи управления ранее, чем в точке выхода.

Симуляция — рефлексивное транзитивное отношение на парах моделей формальной семантики языка программирования, согласно которому каждый шаг вычислений в первой модели соответствует некоторому шагу вычислений во второй модели с сохранением отношения следования.

Изложение материала в статье построено следующим образом. В последующих разделах, соответству-

ющих предлагаемой классификации, перечислены основные особенности промежуточных представлений, которые, на взгляд авторов, наиболее полно характеризуют современное состояние исследований в этой области. Заключение содержит обобщение и анализ результатов, которые приведены в основной части статьи.

Низкоуровневые промежуточные представления, используемые в компиляторах в машинный код

Промежуточные представления программы на низком уровне абстракции от аппаратной платформы, на которой предполагается ее использование, характеризуются линейной структурой кода функций, возможностью описания типов данных, преимущественно с позиций их хранения в памяти, а также близостью набора инструкций к инструкциям машинного кода. Основным представителем данного класса является байт-код LLVM (*Low Level Virtual Machine*), основанный на абстрактном представлении SSA (*Static Single Assignment Form*). Как правило, подобные промежуточные представления слишком близки к машинному коду для адекватного описания формальной семантики. Однако три примера, приведенные далее, показывают, что принципиальных ограничений по описанию формальной семантики таких языков нет.

В работе [1] в рамках проекта Vellvm определена в терминах среды Coq формальная спецификация подмножества инструкций LLVM. Определяется статическая семантика байт-кода и следующие пять моделей операционной семантики: недетерминированная мелкошаговая; детерминированная мелкошаговая (уточняет ряд аспектов семантики, которые не зафиксированы в спецификации байт-кода); крупношаговая с выполнением вызовов функций за один шаг; крупношаговая с выполнением базовых блоков за один шаг; семантика в терминах интерпретатора.

Типизированный язык ассемблера (TAL) [2] предназначен для использования в виде промежуточного представления нижнего уровня абстракции, из которого возможна непосредственная генерация машинного кода. Блоки кода задаются в представлении TAL с помощью тройки (H, R, I), где H — спецификация кучи (частичное отображение из меток в значения на куче; среди значений на куче могут находиться блоки кода), R — спецификация регистров (типы значений, которые находятся в каждом из регистров на момент начала выполнения блока) и I — последовательность инструкций. Переходы между блоками могут происходить только с использованием меток (а не адресов в явном виде), что гарантирует безопасность типов.

Понятие фундаментального кода, несущего доказательство свойств программы (*foundational proof-carrying code*) было предложено А. Аппелем в работе [3]. В рамках демонстрации реализации этого подхода логика второго порядка с аксиомами арифметики в рамках среды Twelf используется для описания операционной семантики машинного кода. Затем в терминах этого представления определяются требуемые свойства кода и выполняется построение доказательства свойств для заданного кода программы. Следует отметить, что аппарат логики второго порядка по сравнению с исчислением конструкций имеет ограниченные выразительные средства. В частности, описание свойств, зависящих от значений переменных, выполняется в ней с помощью предикатов. Кроме того, низкий уровень абстракции машинного кода от оборудования требует формулировать свойства программы в терминах участков памяти.

Виртуальные машины уровня приложений на основе байт-кода, ориентированные на императивные языки программирования

Отличительной особенностью представлений, составляющих основу виртуальных машин уровня приложений для императивных языков программирования (в том числе объектно-ориентированных языков), является более высокий, по сравнению с низкоуровневыми представлениями, уровень абстракции от аппаратной платформы. Эта особенность выражается в наличии инструкций, которые не имеют прямых аналогов в машинном коде распространенных архитектур. В отличие от низкоуровневых промежуточных представлений, более типичным является использование стековых, а не регистровых машин. В таких промежуточных представлениях допускается описание типов данных и функций, поддерживающих параметрический полиморфизм первого порядка. Промежуточные представления, ориентированные на объектно-ориентированные императивные языки, как правило, позволяют в том или ином виде определять типы данных с позиций иерархии наследования.

Стандарт *Common Language Infrastructure* (CLI) [4] определяет систему типов, формат метаданных, виртуальную машину и систему команд для нее, а также набор библиотек, которые в совокупности составляют инфраструктуру взаимодействия произвольных языков программирования, удовлетворяющих этому стандарту. Отличительной особенностью представлений CLI и JVM (*Java Virtual Machine*, в настоящий обзор не входит, так как в основных особенностях аналогична представлению CLI, которое представляет больший интерес с позиций настоящей статьи) является хранение информации об используемых в программах типах данных на высоком уровне абстракции от аппаратной платформы. Информация о типах в представлении CLI хранится в реляционной форме — в виде набора таблиц, в которых содержится информация о составе и типах полей классов, о функциях, дополнительных атрибутах и т. д. Код программы представлен в CLI в виде байт-кода — линейной последовательности инструкций виртуальной машины с переходами по смещению. Набор инструкций виртуальной машины поддерживает:

- низкоуровневые инструкции, аналогичные инструкциям распространенных архитектур про-

цессоров (арифметические инструкции, операции сравнения, условные и безусловные переходы, инструкции вызова);

- распространенные инструкции, предоставляющие абстракцию от аппаратной платформы (чтение и запись заданного поля структуры данных);
- специальный набор инструкций для поддержки объектно-ориентированного программирования (проверка наличия типа в иерархии наследования для заданного значения, виртуальный вызов).

Следует отметить, что система типов CLI поддерживает параметрический полиморфизм — обобщенные типы, при определении которых может быть задан ряд типов-параметров с ограничениями. Ограничения могут иметь вид "данный тип должен наследоваться от заданного типа", "данный тип должен предоставлять реализацию заданного интерфейса" или "данный тип должен иметь конструктор по умолчанию". Еще одним важным аспектом представления CLI (с позиций настоящей статьи) является возможность присваивать записям в таблицах типов произвольные типизированные данные в форме пользовательских атрибутов. Такая расширяемость позволяет хранить в стандартном формате дополнительную информацию, структура которой не определена в стандарте.

Необходимо обратить внимание на то обстоятельство, что пользовательские атрибуты CLI имеют ограничения. В реализации средства Code Contracts вместо пользовательских атрибутов данные записываются непосредственно в код верифицируемой функции в виде инструкций виртуальной машины. Это может объясняться тем фактом, что механизм пользовательских атрибутов не позволяет сохранять данные с требуемой для средств такого рода гранулярностью, а именно не поддерживает присваивание атрибутов базовым блокам. Набор инструкций и модель исполнения виртуальной машины допускают определение обобщенных функций без подстановки параметров типов, а также анализ типов выражений. Работы по описанию формальной статической семантики CLI выполнялись, в частности, и авторами настоящей статьи [5].

Промежуточное представление Vortex IL [6] предназначено для компилятора программ, написанных на разных объектно-ориентированных языках программирования (таких, например, как C++, Java и Modula-3). Для единообразного представления статических процедур, функций-членов классов и мультиметодов в Vortex IL используется понятие обобщенной функции, под которым понимается множество функций с атрибутами, ассоциирующими их с классами — узлами в дереве наследования. Статическая перегрузка имен функций не поддерживается, предполагается ее реализация в интерфейсе компилятора с помощью техники *name mangling*, которая кодирует во внутреннем идентификаторе функции типы ее аргументов. В представлении Vortex IL выделены инструкции для отправки сообщений (вызова методов-членов для экземпляра некоторого класса), доступа к переменным-членам, выделения памяти для объектов, динамической проверки типов объектов (как в форме логических выражений, так и в форме предположений *assertions*). Для поддержки более широкого набора оптимизирующих преобразований авторы отказались от единообразной реализации исключений и остановились на параллельном использовании двух реализаций — высокоуровневой (для языков C++, Java) со специальным соглашением вызова функций и низкоуровневой на основе команд дальнего перехода Setjmp, Longjmp.

Виртуальные машины уровня приложений на основе байт-кода, ориентированные на функциональные языки программирования

В отличие от виртуальных машин уровня приложений, ориентированных на императивные языки программирования, машины, ориентированные на функциональные языки программирования, как правило, поддерживают на уровне байт-кода создание замыканий — структур, которые описывают функции со значениями свободных переменных из внешнего контекста. Следует отметить, что в настоящий раздел входит также описание абстрактной машины Уоррена (WAM), предназначенной для языка Prolog. Строго говоря, язык Prolog относится к логическим языкам программирования. Однако определенные особенности, а именно описание замыканий с использованием перманентных переменных, наличие инструкций, сходных с языком сборки для языка Рефал, а также влияние этого представления на машину BEAM языка Erlang, позволяют рассматривать представление WAM в качестве промежуточного представления для функциональных языков программирования.

В работе [7] описана реализация ZINC для языков программирования ML-семейства. Представление поддерживает такую особенность языка ML, как модули. Абстрактное синтаксическое дерево преобразуется в промежуточное представление на основе безтипового лямбда-исчисления, расширенного константами и следующими конструкциями: локальные объявления (обычные и рекурсивные); примитивные операции (например, арифметические); конструкция множественного выбора switch (с отдельной формой для констант примитивных типов); конструкции обработки исключений, последовательного выполнения операций и цикла while. На следующем этапе это представление преобразуется в граф потока исполнения, узлами которого являются инструкции абстрактной машины ZINC с продолжениями. Листьями графа могут являться инструкции stop (завершение предложения программы верхнего уровня), return (конец функции), termapply (хвостовой вызов функции). Проверка типов была отложена автором [7] в качестве дальшейшей работы, поэтому промежуточные представления не хранят информацию о типах.

Примером промышленного промежуточного представления для программ, написанных на динамическом языке программирования, является байт-код эталонной реализации CPython языка Python (официальной опубликованной документации реа-

лизации CPython не существует, основная информация представлена в работе [8], перечень инструкций с описаниями доступен в работе [9]). В качестве особенности байт-кода CPython следует выделить наличие инструкций для импорта функций и переменных из внешних программных модулей, а также расширенных операций со стеком исполнения (копирование двух верхних элементов стека, подъем второго и третьего элементов стека), инструкции для описания генераторов.

С позиций строгого формального описания представлений для функциональных языков программирования, в работе [10] описан процесс получения интерпретатора абстрактной машины KAM на языке Agda для лямбда-исчисления с простыми типами путем логического вывода из набора правил исчисления с явными подстановками. Интерпретатор является корректным по построению, т. е., во-первых, на его вход может быть подан только корректно типизированный терм, и, во-вторых, для любого корректного входного терма процедура его нормализации (вычисление значения) завершается значением, которое имеет такой же тип. Доказательства корректности проходят проверку средствами языка Agda. Автор статьи [10] отмечает, что аналогичным образом может быть получен корректный по построению интерпретатор и для других разновидностей абстрактных машин, выполняющих бета-редукцию.

В работе [11] рассмотрено промежуточное представление (язык сборки) для языка Рефал. Это представление ориентировано на сопоставление с образцом и переписывание термов. Язык сборки представляет собой линейный байт-код, без нетривиального синтаксиса и нетривиального описания потока исполнения. Поток исполнения задается с использованием стека откатов и инструкций, модифицирующих этот стек. Стек откатов представляет собой последовательность обработчиков исключений.

Промежуточное представление BEAM [12] разработано для языка Erlang на основе абстрактной машины Уоррена и машины Janus Virtual Machine. Особенностями виртуальной машины BEAM являются инструкции управления памятью, работы со списками и кортежами, а также, учитывая специфику языка Erlang, инструкции межпроцессного взаимодействия. Инструкции содержат указатели на начало тела той функции, частью которой они являются. Так как Erlang является языком программирования с динамической типизацией, информация о типах хранится в значениях в процессе выполнения программы, а в наборе инструкций содержатся инструкции динамической проверки типов.

Абстрактная машина Уоррена [13], предназначенная для языка Prolog, существенно отличается как от промежуточных представлений императивных и объектно-ориентированных языков программирования, так и от представлений функциональных языков программирования. Основная часть инструкций машины связана с записью и чтением из памяти структур данных языка Prolog (констант, формул логики первого порядка, переменных и списков), унификацией структур и определением процедур поиска с откатами. Представление имеет вид последовательности инструкций, поток исполнения задается в форме откатов, не имеет абстрактного синтаксиса. Как и для других промежуточных представлений языков с динамической типизацией, предусмотрены инструкции для динамической проверки типов.

Внутренние промежуточные представления на основе графа потока исполнения, сохраняющие информацию о типах значений

Представления на основе байт-кодов, описанные в предыдущих двух разделах, в общем случае, без дополнительного динамического контроля, допускают с помощью инструкций перехода по меткам описание потока исполнения, выделение структуры которого является непростой для решения задачей. Представления, описанные в настоящем разделе, определяют поток исполнения в структурированном виде, в форме графа или, в частном случае, дерева.

Представление на основе продолжений (*continuation-passing style, CPS*) является классическим промежуточным представлением, используемым для трансляции программ, написанных с использованием функциональных языков программирования, в машинные коды. В этом представлении поток исполнения описывается путем передачи в функцию дополнительного аргумента-продолжения, который можно считать функциональным аналогом оператора return из императивных языков программирования. Таким образом, функция в представлении CPS возвращает не само значение, а результат применения этого значения к функции-продолжению. В работе [14] Кеннеди сравнивает продолжения с A-нормальной формой, которая является распространенной альтернативой представления CPS, и с языками на основе монад, которые могут считаться развитием A-нормальной формы. A-нормальная форма представляет собой вложенную последовательность определений именованных значений с помощью конструкции вида let <имя> = <значение> in <выражение>. В результате сравнения делается вывод, что с позиций оптимизирующих преобразований программы продолжения значительно удобнее в использовании. Следует отметить, что представление CPS можно считать частным случаем термов лямбда-исчисления, что позволяет сохранять типизацию. Для поддержки обработки исключений авторы работы [14] предлагают использовать типизированные термы с двумя функциями продолжения: для штатного возврата из функции и для обработки исключительных ситуаций. Такая схема может быть адаптирована и для общего случая эффектов.

В качестве примера использования A-нормальных форм на практике следует отметить компилятор TIL языка ML [15]. В этом компиляторе используется последовательность типизированных промежуточных представлений, начиная от представления Lmli на

основе полиморфного лямба-исчисления на верхнем уровне, через представление Bform, которое представляет собой А-нормальную форму, и заканчивая представлением Lmli-Closure, которое содержит конструкции для явного выделения замыканий. На последующих этапах используется бестиповое промежуточное представление.

Промежуточное представление Thorin [16] предназначено для описания программ, в которых используются функции высшего порядка. По сравнению с представлением SSA, Thorin допускает более компактное представление кода таких программ, а по сравнению с представлением CPS — дополнительный класс оптимизаций под названием *lambda mangling* (декорирование лямбда-термов). Как и в CPS, последовательность вычислений описывается в виде вызовов функций-продолжений. Имена функций и переменных используются в качестве вершин ориентированного графа, направленными ребрами которого, в свою очередь, связаны место использования и определение имени. Такая структура, как и в случае с индексами де Брёйна, позволяет не хранить имена в явном виде и исключить тем самым необходимость коррекции термов при подстановке. При этом, в отличие от индексов де Брёйна, графовое представление допускает ссылки как на ранее определенные имена, так и на имена, определяемые далее по коду, что позволяет задавать взаимно-рекурсивные функции в явном виде.

В недавно вышедшей работе [17] предложено промежуточное представление на основе исчисления секвенций, дополненного лямбда-термами. Одним из итогов исследования, представленного в этой статье, является установление непосредственной связи между промежуточным представлением CPS и предлагаемым авторами работы [17] промежуточным представлением. При этом секвенции описывают продолжения в явном виде, отделенном от обычных лямбда-термов. Такое разделение целесообразно, поскольку позволяет отделить места "служебного" использования продолжений для описания потока исполнения от использования продолжений непосредственно в коде программы. В программах на функциональных языках программирования такая ситуация возникает достаточно часто. Частным случаем последнего являются функции обратного вызова при асинхронном потоке исполнения.

Промежуточное представление среднего уровня абстракции (MIR) в эталонной реализации языка программирования Rust было предложено в работе [18]. Одной из задач, для решения которой было разработано представление, было упрощение реализации проверки типов с возможностью последующего математического обоснования безопасности типов. Представление является графовым, направленным на описание тела отдельной функции. Граф потока исполнения составлен из базовых блоков, которые завершаются одной из специальных инструкций-терминаторов (goto, panic, if, switch, call, diverge, return). Кроме того, имеются "инструкции"-конструкторы сложных выражений (структур, кортежей и массивов). Информация о типах сохраняется, однако в настоящее время не используется при проверке типов. Существенным отличием от SSA-подобных представлений является наличие типов-ссылок. Статический контроль за корректностью передачи и использования ссылок является основной особенностью языка Rust. В явном виде (в виде инструкций-терминаторов или выделенного вида предложений) описываются порождение исключений и момент выхода выражения за пределы области видимости (*drop*). Авторы спецификации отмечают, что некоторые аспекты представления (например, проверка на выход за границы массива по месту обращения) в настоящее время не подлежат статической проверке типов, что ограничивает набор свойств, которые может гарантировать система типов на основе такого представления. После обработки на среднем уровне абстракции, представление MIR транслируется в набор инструкций LLVM.

Высокоуровневые, специфичные для языка программирования промежуточные представления на основе канонических форм синтаксического дерева

Последний класс промежуточных представлений в классификации, предлагаемой авторами настоящей статьи, в значительной степени пересекается с понятием языка программирования и отличается от него единственным аспектом, который относится скорее к прагматике языка. А именно, ряд синтаксических конструкций языка, семантика которых может быть выражена через более простые синтаксические конструкции, заменяются на такие более простые конструкции. Для обозначения удаляемых таким образом синтаксических конструкций используются термины "синтаксический сахар" (*syntactic sugar*) или, в других случаях, "макрос", а удаление таких конструкций называется "раскрытием сахара" (*desugaring*) или "раскрытием макросов" соответственно.

Классическая работа [19] представляет машину STG (*Spineless Tagless G-machine*), которая предназначена для описания программ, использующих нестрогий порядок вычислений (ленивые вычисления). Это промежуточное представление было разработано для реализации языка программирования Haskell. Такое представление можно отнести к классу промежуточных, имеющих нетривиальный абстрактный синтаксис, который в целом аналогичен подмножеству исходного языка Haskell. Следует выделить следующие его особенности:

- спецификация замыкания (перечисление используемых свободных переменных) в определении функции;
- операторы последовательного и рекурсивного определения (let, letrec);
- оператор разбора альтернатив case (среди литералов или по конструкторам алгебраических типов данных);
- работа как с упакованными (находящимися в динамической памяти), так и с неупакованными (передаваемыми по значению) данными.

Промежуточное представление CIL [20] предназначено для анализа, преобразования и последующего восстановления исходного кода на языке программирования C. Оно относится к высокоуровневым промежуточным предоставлениям с нетривиальным абстрактным синтаксисом. В значительной степени представление CIL повторяет основные конструкции языка C (и даже поддерживает такие расширения компиляторов, как атрибуты). Авторы уделяют внимание возможности отображения конструкций промежуточного представления на строки исходной программы. Однако ряд конструкций ("синтаксический сахар") раскрывается в более простые команды. К таким конструкциям относятся циклы (типично для промежуточных представлений императивных языков) и левые части операторов присваивания (*lvalue*).

Язык промежуточного представления данных IDL (*Interface Descriptor Language*) предназначен для описания структур данных и их свойств в форме базовой спецификации структур данных. Эта спецификация задает: состав полей данных и общую схему их взаимодействия; последовательности уточнений (*refinements*), которые определяют конкретные типы данных и их свойства. На представлении IDL основано представление DIANA (*Descriptive Intermediate Attributed Notation for Ada*) программ на языке Ada [21]. Это представление описывает результаты лексического анализа, синтаксического анализа и анализа статической семантики. Основной абстрактной моделью, которой оперирует представление DIANA, являются абстрактные синтаксические деревья и атрибутные деревья, дополняющие узлы синтаксического дерева результатами синтаксического и семантического анализа. Атрибуты могут содержать ссылки на другие узлы атрибутного дерева. Представление DIANA может быть транслировано обратно в исходный текст программы на языке Ada. Однако в отличие от остальных промежуточных представлений, рассматриваемых в настоящем разделе, оно не имеет конкретного синтаксиса.

Представление Henk [22] основано на термах чистых систем типов (PTS), описанных Хэнком Барендрегтом в виде лямбда-куба. Такой выбор базовой модели допускает гибкую настройку выразительных возможностей систем типов с помощью включения тех или иных правил лямбда-куба. Представление Henk основано на лямбда-исчислении второго порядка. Авторы предпочли не вводить правило, допускающее зависимые типы, так как это существенно осложнило бы доказательство завершимости проверки типов. В рамках постановки задачи о разработке промежуточного представления авторы упоминают необходимость сохранения информации о типах на всех этапах трансляции, в частности, в промежуточном представлении. Представление Henk имеет нетривиальный синтаксис. Его операционная семантика в явном виде не задана. Объявления типов отделены от кода функций. Авторы отмечают практическую необходимость в конкретной синтаксической записи представления (с сокращениями и "сахаром") и в многоместных абстракциях и перечислениях (суммах). Верхний уровень представления взаимно рекурсивен, т. е. возможно описание рекурсивных функций в явном виде.

В верифицированном компиляторе языка C CompCert используется последовательность промежуточных представлений [23], каждое из которых имеет заданную формальную семантику, описанную в терминах исчисления конструкций в среде Coq. Входным для компилятора является представление Clight, аналогичное абстрактному синтаксису языка C с раскрытием сложных синтаксических конструкций ("синтаксического сахара"), без побочных эффектов внутри приложений и без объявлений переменных на уровне блоков. Первые две стадии компиляции переводят представление Clight в представление C#minor. В последнем удаляется перегрузка арифметических операторов, а также приводятся к единому виду циклы. Затем выполняется трансляция в представление Cminor, удаляющее оператор получения указателя (&) и заменяющее локальные переменные, адреса которых используются в теле функции, на области памяти, выделенные на стеке. На следующем шаге выполняется специфичная для целевой аппаратной платформы подстановка специализированных арифметических инструкций процессора (представление CminorSel). После этого код транслируется в граф потока управления, узлы которого содержат наборы инструкций в представлении RTL (*Register transfer language*). На нескольких последующих стадиях на основе графа потока исполнения происходит генерация машинного кода, однако эти вопросы выходят за пределы настоящего обзора. Следует отметить, что для каждой стадии компиляции в среде Coq получено доказательство сохранения семантики на этой стадии с использованием формальной семантики исходного и целевого промежуточных представлений.

Исходным языком для средства формальной верификации Boogie [24] является язык Spec# — расширение языка C# конструкциями для описания инвариантов, пред- и постусловий, а также их динамическими проверками. Средство Boogie на уровне промежуточного представления CIL распознает такие динамические проверки и выполняет статическую верификацию на предмет выполнения таких условий. Представление CIL транслируется во внутреннее представление BoogiePL, которое для верификации преобразуется в последовательность предложений логики первого порядка. Представление BoogiePL имеет нетривиальный синтаксис и состоит из "теории" — набора объявлений типов, имен объектов и логических аксиом, которые в совокупности определяют семантику исходного языка, — и императивной части, в которой содержатся объявления переменных, объявления процедур и реализации процедур. По сравнению с обычным императивным языком программирования представление BoogiePL содержит специальные команды assert (условие, которое должно быть верифицировано, при невыполнении порождается ошибка), assume (условие, которое

подразумевается в качестве данного для процесса верификации, а при невыполнении которого считается, что дальнейшее доказательство корректности процедуры можно получить по принципу ex falso) и команды havoc (присвоить переменной произвольное корректно типизированное значение, удовлетворяющее ограничениям, которые задаются опционально при объявлении переменной). Контроль потока исполнения в представлении ограничивается командой недетерминированного перехода на один из базовых блоков, перечисленных в аргументах команды. Это позволяет, с одной стороны, генерировать условия верификации для всех возможных путей исполнения тела процедуры, но с другой стороны, ограничивает возможность по выводу проверок по потоку исполнения в ходе выполнения процедуры (вместо этого используется команда assume). Набор базовых блоков представления CIL преобразуется в плоское абстрактное синтаксическое дерево, в котором побочные эффекты вычисления аргументов задаются как отдельные слоты стека вычислений. Стек вычисления является обобщением понятия стека исполнения CIL: на стек допускается помещение выражений (а не только значений).

Заключение

В настоящей статье представлена классификация методов промежуточного представления программ с позиций их формальной спецификации. Для каждого из классов приведены примеры, в общей сложности 22 промежуточных представления, которые позволяют охарактеризовать современное состояние исследований на этом направлении.

Если упорядочить рассмотренные промежуточные представления в хронологическом порядке (заметим, что в настоящей статье могут использоваться ссылки на более поздние источники), можно выделить следующие изменения в задачах, которые ставятся перед промежуточным представлением.

1. Ранние промежуточные представления ([4, 6, 7, 11—14, 19, 21], а также классические представления — SSA и трехадресный код) используются, в первую очередь, в качестве фиксированной структуры данных, другими словами, интерфейса между различными стадиями трансляции программы. Интерфейс промежуточных представлений оказался удобен также для декомпозиции трансляторов и при создании внешних средств анализа и трансформации кода (например, оптимизаторов): подобные этапы являются последовательностью отображений промежуточного представления в себя, в общем случае частичных отображений (например, стадии анализа могут прерывать процесс трансляции с ошибкой). Кроме того, естественным образом такие отображения использовались при реализации взаимодействия между фрагментами кода программы, написанными с использованием нескольких разных языков программирования ([4, 6], в настоящее время — [12] в реализации языка Elixir).

2. В более поздних промежуточных представлениях ([2, 15—18, 20, 22, 23], в меньшей степени — в упомянутом в предыдущем пункте [4]) большее внимание уделяется системе типов. В дополнение к роли интерфейса, такие промежуточные представления используются для упрощения реализации процедур проверки типов.

3. Отдельно следует отметить промежуточные представления, на которых реализуются процедуры формальной верификации программ (в настоящем обзоре к таким представлениям относятся представления, описанные в работах [1, 10, 24] и упомянутые ранее [2, 23]).

Промежуточные представления, направленные на решение задач практического характера, по сравнению с результатами академических исследований, как правило, имеют большое количество примитивов (инструкций, команд), многие из которых специфичны для задач, на решение которых нацелено то или иное промежуточное представление. Достаточно большой вклад в число команд вносят константы примитивных типов, операции над ними и определяющие их отношения, такие как, например, равенство и порядок на типе целочисленной арифметики аппаратной платформы.

Следует отметить также определенное сходство между промежуточными представлениями объектно-ориентированных языков программирования (например, [4, 6]) и промежуточными представлениями языков с динамической типизацией [8, 12]: и в том и в другом случае в набор инструкций представления входят инструкции динамической (в процессе выполнения) проверки типов.

С позиций математического описания промежуточных представлений следует обратить внимание на тот факт, что исполнители проекта Vellvm разработали несколько крупношаговых (менее детализированных) моделей семантики для того, чтобы на этих моделях можно было обосновать ряд преобразований, проводимых компилятором в рамках оптимизации. Этот факт показывает, что для адекватного описания функциональных свойств программ их промежуточное представление должно иметь более высокий уровень абстракции от среды исполнения, допускающий определенную свободу реализации. Мелкошаговая операционная семантика существенно ограничивает набор преобразований, сохраняющих семантику.

Систематический обзор работ по построению промежуточных представлений программ представлен в статье [25]. Отчет [26] содержит более детальное по сравнению с настоящей работой сравнение типизированных промежуточных представлений, но содержит только три представления, а именно: виртуальную машину Java, во многом эквивалентную представлению CLI, вошедшему в настоящий обзор; типизированный язык ассемблера TAL, упоминаемый и в настоящем обзоре; код, несущий доказательство, развитием которого является упомянутый выше фундаментальный код, несущий доказательство.

Языки LISP и Forth, которые также традиционно используются в качестве промежуточных представлений, выходят за рамки настоящего обзора, однако

их можно было бы охарактеризовать как представление класса "высокоуровневое представление с нетривиальным синтаксисом на основе канонических форм синтаксического дерева" (LISP) и виртуальную машину уровня байт-кода, ориентированные на императивные языки программирования (Forth).

Представленные в настоящей статье результаты анализа различных технических решений, использованных в известных промежуточных представлениях, могут быть применены при разработке новых языков программирования и промежуточных представлений.

Работа выполнена при поддержке РФФИ, проект № 16-07-01178а.

Список литературы

1. **Zhao J., Nagarakatte S., Martin M. M. K., Zdancewic S.** Formalizing the LLVM Intermediate Representation for Verified Program Transformations // Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2012. P. 427—440.

2. **Morrisett G., Walker D., Crary K., Glew N.** From System F to Typed Assembly Language // ACM Trans. Program. Lang. Syst. 1999. Vol. 21, N 3. P. 527—568.

3. **Appel A.** Foundational proof-carrying code // Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. IEEE Comput. Soc., 2001. P. 247—256.

4. **ISO** ISO/IEC 23271:2003: Information technology — Common Language Infrastructure. Geneva, Switzerland: International Organization for Standardization, 2005. xi + 99 (Part. I), ix + 164 (Part. II), vi + 125 (Part. III), iii + 16 (Part. IV), iv + 79 (Part. V) c.

5. **Васенин В. А., Кривчиков М. А.** Статическая семантика стандарта ECMA-335 // Программирование. 2012. № 4. С. 3—16.

6. **Dean J., DeFouw G., Grove D., Litvinov V., Chambers C.** Vortex: An Optimizing Compiler for Object-oriented Languages // Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM, 1996. P. 83—100.

7. **Leroy X.** The ZINC experiment: an economical implementation of the ML language: Technical report. INRIA, 1990.

8. **Portnoy A., Santiago A.-R.** Reverse Engineering Python Applications // Proceedings of the 2Nd Conference on USENIX Workshop on Offensive Technologies. Berkeley, CA, USA: USENIX Association, 2008. P. 6:1—6:5.

9. **Van der Laan W. J.** Python Bytecode Archeology. 2011. URL: https://laanwj.github.io/2011/5/4/python-bytecode-archeology (дата обращения 01.02.2017).

10. **Swierstra W.** From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine // Electronic Proceedings in Theoretical Computer Science. 2012. Vol. 76. P. 163—177.

11. **Романенко С. А**. Машинно независимый компилятор с языка рекурсивных функций: дис. ... канд. физ.-мат. наук. М.: ИПМ АН СССР, 1978. 148 с.

12. **Hausman B.** Turbo Erlang: Approaching the Speed of C // Implementations of Logic Programming Systems / Eds. E. Tick, G. Succi. Springer US, 1994. P. 119—135.

13. **Aït-Kaci H.** Warren's Abstract Machine: A Tutorial Reconstruction. Cambridge, Mass: The MIT Press, 1991. 114 p.

14. **Kennedy A.** Compiling with continuations, continued // ACM SIGPLAN Notices. 2007. Vol. 42, N 9. P. 177—190.

15. **Tarditi D., Morrisett G., Cheng P. et al**. TIL: A Type-directed Optimizing Compiler for ML // Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, 1996. P. 181—192.

16. **Leißa R., Köster M., Hack S.** A graph-based higher-order intermediate representation // 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2015. P. 202—212.

17. **Downen P., Maurer L., Ariola Z. M., Jones S. P.** Sequent Calculus As a Compiler Intermediate Language // Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, 2016. P. 74—88.

18. **Matsakis N.** *rust-lang/rfcs/1211-mir.md. GitHub. 2016. URL: https://github.com/rust-lang/rfcs

19. **Jones S. P.** Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine // Journal of Functional Programming. 1992. Vol. 2, P. 127—202.

20. **Necula G. C., McPeak S., Rahul S. P., Weimer W** CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs // Compiler Construction. Berlin, Heidelberg, Springer, 2002. P. 213—228.

21. **DIANA** An Intermediate Language for Ada / Eds. G. Goos et al. Berlin, Heidelberg: Springer, 1983.

22. **Jones S. P., Meijer E.** Henk: A Typed Intermediate Language // In Proc. First Int'l Workshop on Types in Compilation. 1997. URL: http://rnyingma.synrc.com/publications/cat/Lambda%20Calculus/Henk.pdf

23. **Leroy X**. Formal Verification of a Realistic Compiler // Communications of the ACM. 2009. Vol. 52, N 7. P. 107—115.

24. **Barnett M., Chang B. E., DeLine R. et al.** Boogie: A Modular Reusable Verifier for Object-Oriented Programs // Formal Methods for Components and Objects. Berlin, Heidelberg, Springer, 2005. P. 364—387.

25. **Diehl S., Hartel P., Sestoft P.** Abstract machines for programming language implementation // Future Generation Computer Systems. 2000. Vol. 16, N 7. P. 739—751.

26. **Tse S.** Typed Intermediate Languages: Technical Report MS-CIS-04-17. University of Pennsylvania Department of Computer and Informational Scinece, 2004.

Program Intermediate Representation Techniques

V. A. Vasenin, vasenin@msu.ru, **M. A. Krivchikov,** maxim.krivchikov@gmail.com, Lomonosov Moscow State University, Moscow, 119234, Russian Federation

Corresponding author:

Krivchikov Maxim A., Senior Researcher, Lomonosov Moscow State University, Moscow, 119234, Russian Federation,
E-mail: maxim.krivchikov@gmail.com

Received on April 21, 2017
Accepted on May 30, 2017

The concept of an intermediate representation of a program originates from software translation theory. In the present paper the classification of high-level intermediate representations is proposed, together with the review of current state-of-the-art research in intermediate representations. The paper presents a part of a research project aimed at development of intermediate representation for high-level specification of formal semantics for domain-

specific programming languages. The review is mostly concerned with mathematical representation of intermediate representations. The five proposed intermediate representation classes are:

- *low-level intermediate representations used mostly in compilers;*
- *bytecode-based application virtual machines for imperative programming languages;*
- *bytecode-based application virtual machines for functional programming langauges;*
- *internal graph-based intermediate representations;*
- *high-level, programming language-specific intermediate representations based on canonical forms for the AST.*

The review contains 22 intermediate representations. Looking at them in chronological order we can observe the following shift in problem statements. Earlier intermediate represntations ([4, 6, 7, 11—14, 19, 21] and classical examples of SSA, RTL and three-address code) are used primarily as a fixed data structure, in other words, an interface between different stages of compilation pipeline. Recent intermediate representations ([2, 15—18, 20, 22, 23] and, to lesser degree, [4], mentioned previously) are more concerned with type system. These representations are used to simplify implementation of the typechecker. In addition, we shall mention intermediate representations aimed at formal verification procedures ([1, 10, 24] and [22, 23] from previous group).

Industrial intermediate representations (in comparison with academic research) usually have more primitive instructions or commands. The increase is mostly due to primitive type constants, operations and relations. We shall also note the similarity between intermediate representations for the object-oriented programming languages (e.g. [4, 6]) and dynamically-typed programming languages ([8, 12]): in both cases instructions of the dynamical (run-time) type checking are present.

Related work includes the article [25], which contains systematic review of intermediate representations, and report [26] with rather detailed comparison of type system expressiveness for three typed intermediate representations.

Keywords: *programming languages, intermediate representation, program translation, formal semantics, software engineering, review, classification*

For citation:

Vasenin V. A., Krivchikov M. A. Program Intermediate Representation Techniques, *Programmnaya Ingeneria*, 2017, vol. 8, no. 8, pp. 345—353.

DOI: 10.17587/prin.8.345-353

References

1. **Zhao J., Nagarakatte S., Martin M. M. K., Zdancewic S.** Formalizing the LLVM Intermediate Representation for Verified Program Transformations, *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012, pp. 427—440, doi:10.1145/2103656.2103709.

2. **Morrisett G., Walker D., Crary K., Glew N.** From System F to Typed Assembly Language, *ACM Trans. Program. Lang. Syst.,* 1999, vol. 21, pp. 527—568.

3. **Appel A. W.** Foundational proof-carrying code, *In Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 2001, pp. 247—256, doi:10.1109/LICS.2001.932501.

4. **ISO.** ISO/IEC 23271:2003: Information technology — Common Language Infrastructure, 2005.

5. **Vasenin V. A., Krivchikov M. A.** Staticheskaja semantika standarta ECMA-335 (ECMA-335 Static Formal Semantics), *Programming and Computer Software*, 2012, vol. 38, no. 4, pp. 183—188 (in Russian).

6. **Dean J., DeFouw G., Grove D., Litvinov V., Chambers C.** Vortex: An Optimizing Compiler for Object-oriented Languages, *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1996, pp. 83—100, doi:10.1145/236337.236344.

7. **Leroy X.** The ZINC experiment: an economical implementation of the ML language. Technical report, INRIA, 1990.

8. **Portnoy A., Santiago A.-R.** Reverse Engineering Python Applications, *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies,* 2000, vol. 6, pp. 1—5.

9. **Van der Laan W. J.** *Python Bytecode Archeology*, 2011, available at: https://laanwj.github.io/2011/5/4/python-bytecode-archeology.

10. **Swierstra W.** From Mathematics to Abstract Machine: A formal derivation of an executable Krivine machine, *Electronic Proceedings in Theoretical Computer Science*, 2012, vol. 76, pp. 163—177.

11. **Romanenko S. A.** *Mashinno nezavisimii kompilyator s yazika rekursivnikh funkcii* (Machine-independent compiler for recursive functions language), PhD thesis (Keldysh Institute of Applied Mathematics), 1978, 148 p. (in Russian).

12. **Hausman B.** Turbo Erlang: Approaching the Speed of C, *Implementations of Logic Programming Systems* / Eds. E. Tick, G. Succi. Springer US, 1994, pp. 119—135.

13. **Aït-Kaci H.** *Warren's Abstract Machine: A Tutorial Reconstruction,* Cambridge, Mass, The MIT Press, 1991, 114 p.

14. **Kennedy A.** Compiling with continuations, continued, *ACM SIGPLAN Notices*, 2007, vol. 42, no. 9, pp. 177—190.

15. **Tarditi D., Morrisett G., Cheng P., Stone C., Harper R., Lee P.** TIL: A Type-Directed Optimizing Compiler for ML. *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996, pp. 181—192, doi:10.1145/231379.231414.

16. **Leissa R., Koster M., Hack S.** A graph-based higher-order intermediate representation, *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO),* 2015, pp. 202—212.

17. **Downen P., Maurer L., Ariola Z. M., Jones S. P.** Sequent Calculus As a Compiler Intermediate Language, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 74—88, doi:10.1145/2951913.2951931.

18. **Matsakis N.** *rust-lang/rfcs/1211-mir.md, 2016, available at: https://github.com/rust-lang/rfcs

19. **Jones S. P.** Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine, *Journal of Functional Programming,* 1992, vol. 2, pp. 127—202.

20. **Necula G. C., McPeak S., Rahul S. P., Weimer W.** CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, *Compiler Construction,* 2002, pp. 213—228, doi:10.1007/3-540-45937-5_16.

21. **DIANA** An Intermediate Language for Ada. / Eds. **G. Goos,** Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

22. **Jones S. P., Meijer E.** Henk: A Typed Intermediate Language, *Proc. First Int'l Workshop on Types in Compilation,* 1997, available at: http://rnyingma.synrc.com/publications/cat/Lambda%20Calculus/Henk.pdf

23. **Leroy X.** Formal Verification of a Realistic Compiler, *Communications of the ACM,* 2009, vol. 52, no. 7, pp. 107—115.

24. **Barnett M., Chang B. E., DeLine R., Jacobs B., Leino K. R. M.** Boogie: A Modular Reusable Verifier for Object-Oriented Programs, *Formal Methods for Components and Objects,* 2005, pp. 364—387, doi:10.1007/11804192_17.

25. **Diehl S., Hartel P., Sestoft P.** Abstract machines for programming language implementation, *Future Generation Computer Systems,* 2000, vol. 16, no. 7, pp. 739—751.

26. **Tse S.** Typed Intermediate Languages: Technical Report MS-CIS-04-17, University of Pennsylvania Department of Computer and Informational Science, 2004.