Микаэль Франц

Динамическая кодогенерация: ключ к разработке переносимого программного обеспечения

Michael Franz (1994) «Code-Generation On-the-Fly: A Key for Portable Software» // ETH Zurich, Diss. No.10497.
Р. Богатырев, А. Китаев, перевод с англ.

Об авторе. Микаэль Франц (Michael Franz) — ныне адъюнкт-профессор компьютерных наук в университете Калифорнии в Ирвине (University of California at Irvine). Ранее работал ведущим исследователем (Senior Research Associate) в группе Никлауса Вирта в ETH (Цюрих). Будучи одним из первых коллег Н. Вирта по проекту Oberon, внес значительный вклад в развитие этой технологии. М. Франц является автором ETH-реализации системы Oberon для Apple Macintosh.

Аннотация. В настоящей диссертации предлагается метод представления программ в виде абстрактном и независящим от возможной целевой архитектуры, который позволяет получить представление файлов в два раза более компактное, нежели машинный код для CISC-процессора. Этот метод лег в основу реализованной автором системы, в которой процесс кодогенерации откладывается до этапа загрузки. В этот момент загрузчик с динамической кодогенерацией и порождает чистый машинный (native) код.

При динамической кодогенерации процесс загрузки протекает настолько быстро, что на него уходит лишь чуть больше времени, нежели на ввод эквивалентного чистого кода из дискового запоминающего устройства. Это объясняется главным образом компактностью абстрактного представления программы: дополнительное время, потраченное на кодогенерацию, компенсируется за счет сокращения времени ввода. Поскольку мощность процессоров растет в настоящее время быстрее, нежели сокращаются время доступа к диску и скорость передачи данных, есть основания полагать, что по мере дальнейшего развития технологии производства аппаратных средств предлагаемый метод станет еще более конкурентоспособным.

Пользователи реализованной системы могут теперь работать с модулями в абстрактном представлении с той же легкостью, что и с обычными объектными файлами. В этой системе представления файлов обоих типов сосуществуют; они полностью взаимозаменяемы, и модули в любом представлении могут импортировать данные из модуля другого типа. Реализована полная поддержка раздельной компиляции (separate compilation) программных модулей с интерфейсами, обеспечивающими контроль типов, а также помодульная динамическая загрузка (dynamic loading).

Задержка кодогенерации до этапа загрузки открывает ряд новых возможностей, особенно в тех случаях, когда промежуточное представление программы не зависит от используемой машины и является таким образом переносимым. В диссертации показано, что сочетание переносимости и практичности представляет собой важный шаг на пути к индустрии программных компонент (software-component industry). Среди других преимуществ нового метода — возможность сокращения числа повторных компиляций при внесении изменений в исходный текст, а также механизм, позволяющий определять, следует ли динамически проверять на целостность библиотечный модуль (это избавляет от необходимости различать конфигурации библиотек, предназначенных для разработки программных продуктов). Есть все основания полагать, что эти новые возможности приведут к сокращению издержек по проектированию и эксплуатации программного обеспечения.

Не исключено, что в перспективе переносимость программного обеспечения на различные процессоры, реализующие одну и ту же архитектуру, будет достигаться уже не за счет бинарной совместимости, а с помощью быстрой динамической кодогенерации. Ведь уже сегодня различные модели семейства процессоров все сильнее отличаются друг от друга, так что одинаково успешно работать с ними, применяя лишь одну версию машинного кода, становится все труднее. Если же код генерируется лишь во время загрузки, его всегда можно «подогнать» под конкретный процессор, на котором он в конечном счете и будет работать.

Ключевые слова. Динамическая кодогенерация, SDE-технология, семантический словарь, абстрактные машины, абстрактное синтаксическое дерево.

Благодарности. Я глубоко признателен профессору Никлаусу Вирту (Niklaus K. Wirth) за выпавшее на мою долю счастье работать рядом с ним. На протяжении многих лет он одаривает меня сокровищами мудрости, которые помогают и будут помогать мне в течение всей моей профессиональной жизни, да и после ее окончания. Я искренне преклоняюсь перед его редким даром мгновенно выделять существенную сторону явлений и перед тем мужеством, с которым он отказывается от всего лишнего и наносного, даже когда другие считают отброшенные им вещи просто незаменимыми. Довольно часто я пытаюсь следовать в своей работе его принципам суждений — и результаты таковы, что ими нельзя не восхищаться. Я сердечно признателен ему за внимательную опеку моей диссертации и за те компетентные замечания к проводимому проекту, что мне довелось от него слышать. Я благодарен профессору Юргу Гуткнехту (Juerg Guetknecht) за его согласие быть еще одним научным консультантом (co-examiner) моей диссертации и за помощь в ее подготовке. Он поистине великий педагог, и если содержание моей работы покажется вам четким и понятным, то это целиком и полностью его заслуга, это результат его критических замечаний на ранней стадии проекта, после которых диссертация унаследовала его непревзойденный стиль изложения. Я выражаю признательность моим коллегам из Института компьютерных систем (Institute for Computer Systems) в Цюрихе за их готовность к восприятию новых идей и за тот энтузиазм, с которым они участвовали в их обсуждении, даже хотя наши дискуссии нередко заканчивались далеко за полночь. Благодаря всем им создалась настоящая рабочая атмосфера дружбы и интеллектуального творчества. В заключение мне хочется выразить свою благодарность Куно Пфистеру (C.Pfister) и Стефану Людвигу (S.Ludwig) за их внимательное прочтение моей диссертации.

СОДЕРЖАНИЕ

1. Введение

Быстрое развитие аппаратной технологии оказывает постоянное влияние на разработку программного обеспечения, и это ведет как к положительным, так и к отрицательным последствиям. Случается, что быстродействие аппаратных средств маскирует изъяны и стоимость неудачно спроектированных программ. Мартин Рейзер [Rei89] не так уж далек от истины, когда замечает, что иногда «программы замедляют свою работу быстрее, чем ускоряется быстродействие аппаратуры». С другой стороны, усовершенствования в аппаратных средствах, такие, как увеличение объема памяти и повышение быстродействия процессоров, обеспечивают возможность создания более совершенных программ.

Достаточно часто появление аппаратуры более высокого качества имеет своим косвенным следствием принципиально новые методические решения. Взять хотя бы такие приемы программной инженерии, как инкапсуляция информации (information hiding) и абстрактные типы данных (abstract data types). Их просто нельзя было разработать до того, как компьютеры стали настолько мощными, что смогли поддерживать компиляторы для модульных языков программирования. Механизмы, подобные раздельной компиляции, также предъявляют определенные требования к используемому оборудованию, и они тоже смогли получить распространение лишь после того, как повсюду стали применяться достаточно мощные компьютеры.

В диссертации представлен пример еще одного новаторского решения в области систематической технологии, реализация которого стала возможной благодаря усовершенствованию аппаратных средств. В ней описывается метод представления программ в абстрактном виде — в формате, в два раза более компактном, нежели объектный код для CISC-процессора. Применение промежуточного представления программ с помощью нового метода в сочетании с быстродействием современных процессоров и большим объемом основной памяти позволяет столь существенно ускорить процесс кодогенерации, что он может совершаться «на лету» в ходе

загрузки программы в память обычного настольного компьютера, причем с получением высококачественного кода.

Автором реализована система, позволяющая оперировать программными модулями в промежуточном машинно-независимом представлении «со всеми удобствами» — как если бы они были транслированы в машинный код. В диссертации утверждается, что такая система открывает несколько абсолютно новых возможностей, в числе которых — перспектива создания целой индустрии, производящей компоненты программного обеспечения непосредственно для пользователей, причем компоненты эти можно применять непосредственно на нескольких типах целевой архитектуры.

Среди тем, к которым диссертант многократно обращается в данной работе, — методы представления программ, кодогенерация, модульное программирование (modular programming) и переносимость программного обеспечения. Этот достаточно широкий для диссертации круг проблем нашел свое отражение в объемном списке литературы, включающем самые разнообразные источники.

2. Представление программ

Для каждой вычислимой функции (calculable function) существует алгоритм ее вычисления [Chu36]. А программа — это закодированное описание такого алгоритма, дающее универсальной вычислительной машине указание на вычисление соответствующей функции. Машина именуется универсальной в том случае, если она позволяет вычислить любую функцию, которую можно представить с помощью алгоритма; возможность существования таких универсальных машин показана еще Тьюрингом [Tur37]. Языки программирования соответствуют абстрактным и относительно сложным универсальным машинам, тогда как цифровые компьютеры представляют собой физическую реализацию универсальных машин, которые можно программировать с помощью характерных для этих компьютеров машинных языков.

Алгоритмы могут транслироваться из одного представления программы в другое; само существование некоторого алгоритма свидетельствует о возможности создания соответствующей программы для любой универсальной машины. В этой главе обсуждается применение промежуточных языков в качестве переходных шагов в таких трансформациях между представлениями программ, в особенности — решений, основанных на абстрактных машинах. Затем в ней вводится новый метод представления программ, именуемый кодированием на основе семантического словаря (semantic-dictionary encoding). Этот метод позволяет добиться высокой информационной плотности и одновременно облегчает дальнейший эффективный перевод на различные машинные языки.

Декомпозиция компиляторов

Часто (и по самым разным причинам) компиляторы разделяют на более мелкие элементы. Это может быть статическое разделение на модули, либо динамическое разделение на две и более фазы компиляции (или прохода), выполняющиеся друг за другом. На такое разделение часто приходится идти из-за физических ограничений инструментального компьютера, не позволяющих создать монолитный компилятор. Однако декомпозиция компилятора на более мелкие элементы дает и другие преимущества, которые делают этот подход привлекательным, даже если он и не диктуется внешними ограничениями. Привлекательность его сохраняется даже тогда, когда упомянутые органичения в конце концов устраняются с развитием аппаратной технологии.

Наиболее важное обстоятельство состоит в том, что разделение компилятора обычно приводит к его упрощению. Ведь отдельные элементы меньше и, следовательно, проще, нежели целое, а все части, взятые вместе, по сложности зачастую уступают соответствующему монолитному компилятору. Объясняется это тем, что различные части, как правило, довольно легко отделяются друг от друга, и в результате между ними образуются узкие интерфейсы. С другой стороны, сложность концентрируется, главным образом, в нелокальных зависимостях.

Еще один довод в пользу структурирования компилятора — это та легкость, с которой его можно перенацеливать на другую архитектуру. Отделив кодогенерирующие функции от остальной части

компилятора, мы можем существенно упростить создание семейства компиляторов для различных целевых архитектур. И тогда для адаптации компилятора к новой целевой машине достаточно будет создать новый кодогенератор, а в остальных частях компилятора ничего менять не придется. К слову, этот метод стал настолько распространенным, что мы сегодня часто говорим о «кодогенераторах», как если бы они сами по себе были полноценными программами, совершенно забывая о том, что в первых компиляторах синтаксический анализ и генерация кода не были отделены друг от друга.

Промежуточные языки

Компилятор преобразует программу из одного представления в другое. Когда такой компилятор состоит из нескольких фаз, то это подразумевает наличие еще нескольких промежуточных представлений (intermediate representations), переносящих состояние компиляции из предыдущей фазы в последующую. Как правило, эти промежуточные представления — всего лишь переходные структуры данных в памяти, но в отдельных компиляторах они имеют линейную форму, которая может быть отражена в файле данных. В последующем изложении такие особые линейные представления именуются промежуточными языками (intermediate languages).

Промежуточные языки представляют интерес с точки зрения переносимости программного обеспечения на различные компьютеры. Если промежуточный язык достаточно прост, можно без труда создать несколько интерпретаторов, способных непосредственно работать с этим языком на ряде различных целевых машин. Этот подход представляет интерес, ибо вообще говоря, интерпретаторы проектировать легче, нежели кодогенераторы. Правда, за простоту реализации здесь приходится платить ухудшением рабочих характеристик. В числе систем, реализованных на основе интерпретируемых таким образом промежуточных языков, можно назвать BCPL [Ric71, RiW79], SNOBOL4 [Gri72] и Pascal [NAJ81].

Кроме того, промежуточные языки часто используются в процессе первоначальной загрузки компиляторов на новые машины [Hal65]. Допустим, унас есть переносимый компилятор, который транслирует входной язык SL (Source Language) на промежуточный язык IL (Intermediate Language) и что этот компилятор сам написан на SL, а закодирован как на SL, так и на IL. Допустим далее, что мы располагаем интерпретатором для языка IL, который выполняется на компьютере с целевым машинным языком TL (Target Language). Тогда мы сможем, модифицировав переносимый компилятор (написанный на SL), получить компилятор в машиный код (native compiler), транслирующий прямо с SL на TL. Скомпилировав его с интерпретируемой версией первоначального компилятора, мы получим компилятор в машинный код, написанный на языке IL и, таким образом, выполнимый для данного интерпретатора. Теперь с помощью этого компилятора можно создавать компилятор в машинный код, способный выполняться непосредственно на целевой машине (см. рис. 2.1).

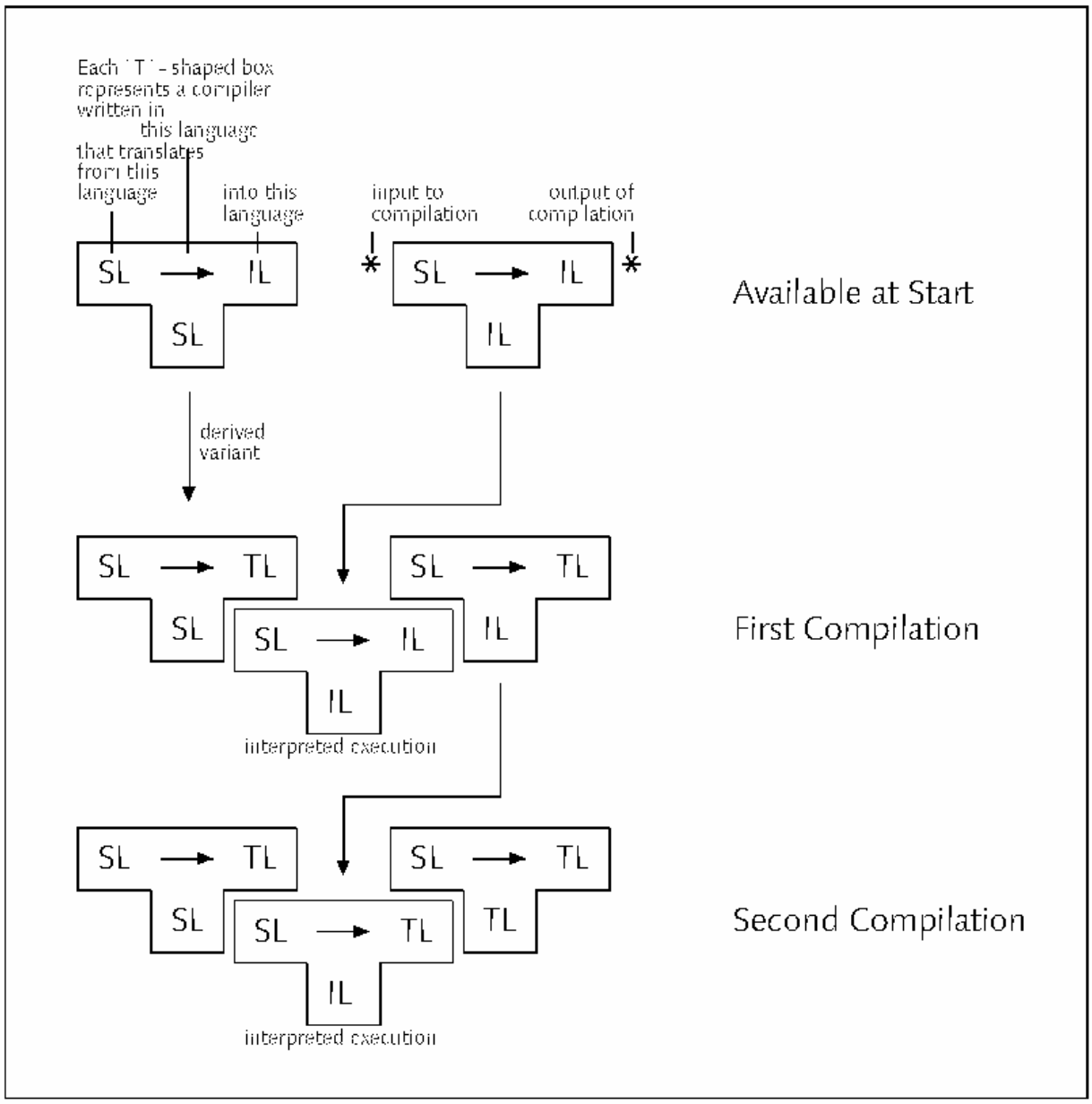

Рис. 2.1. Раскрутка компилятора (bootstrapping)

Универсальный язык UNCOL

В 1950-х годах стало очевидно, что тенденция к росту числа языков программирования и типов аппаратных архитектур еще какое-то время сохранится. А это, соответственно, вызовет большой спрос на различные компиляторы, поскольку для каждой комбинации входного языка и целевой архитектуры потребуется отдельный компилятор.

Иными словами, можно было ожидать настоящего комбинаторного взрыва, и в этой обстановке родилась идея «языкового коммутатора» (linguistic switchbox), которая и была материализована в 1958 г. [SWT58, Con58, Ste60, Ste61a]. Замысел состоял в том, чтобы разработать некий универсальный промежуточный язык, на который с помощью соответствующего преобразователя (front-end, компилятора переднего плана, если пользоваться нынешней терминологией) можно было бы транслировать программы, написанные на любом проблемно-ориентированном языке и с которого при помощи подходящего генератора (back-end) можно было бы генерировать код для любой акрхитектуры процессора. В этом случае (если принять число языков за «m», а число машин, на которых компиляторы должны работать, — за «n»), для компиляторов уже не пришлось бы писать m*n программ. Можно было бы обойтись гораздо меньшим их числом —

всего m+n, то есть m-программ для front-end частей системы, и n-программ — для back-end частей (см. рис. 2.2). Для этого промежуточного языка было выбрано имя UNCOL, что означает «универсальный машинно-ориентированный язык» (universal computer-oriented language).

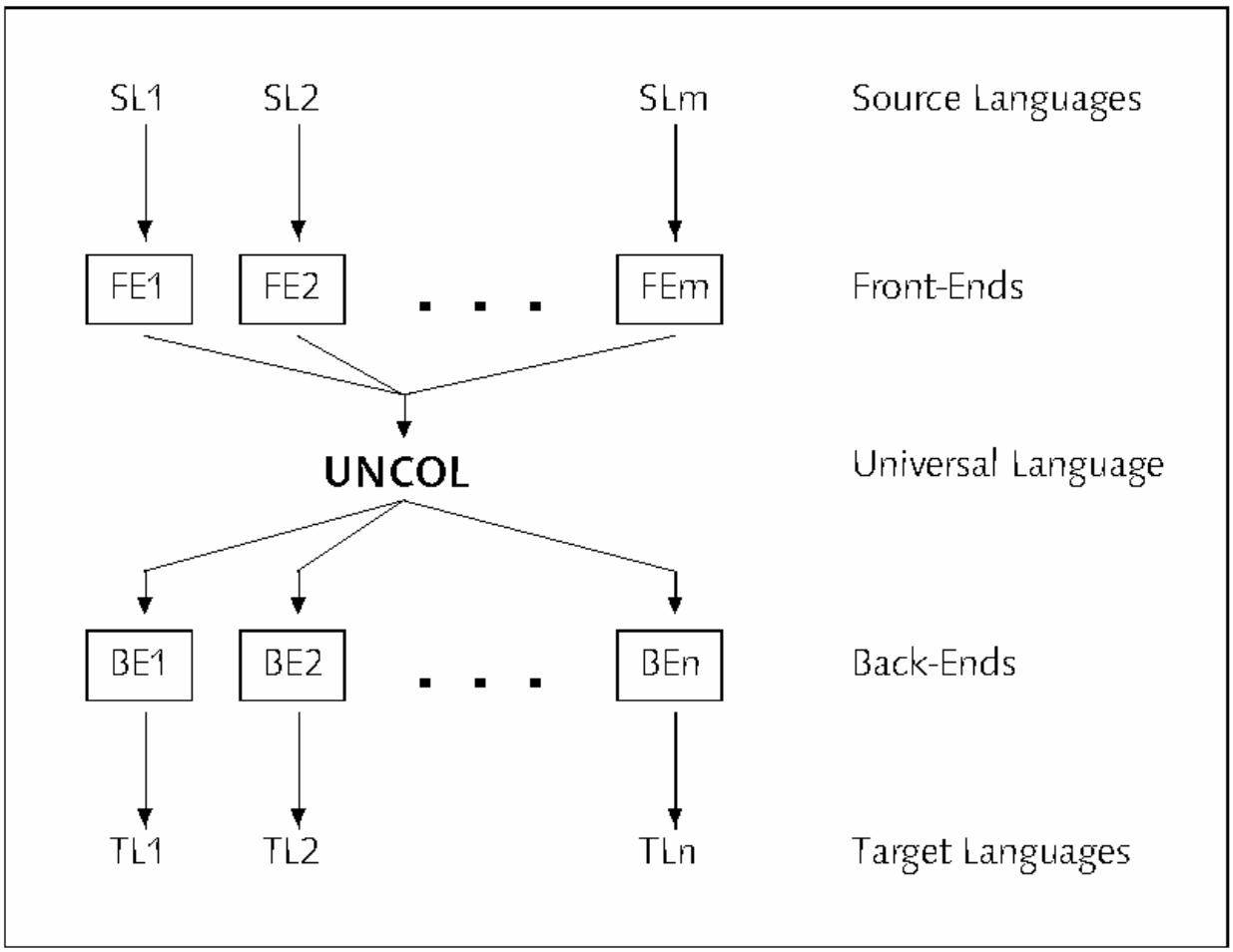

Рис. 2.2. Язык UNCOL в качестве языкового коммутатора

Разумеется, может возникнуть вопрос, почему подобный промежуточный язык надо обязательно проектировать специально для этой цели. Ведь если алгоритмы могут транслироваться из одного представления программы в другое, тогда в принципе в качестве языка UNCOL может выступать машинный язык любого достаточно мощного компьютера. К сожалению, однако, трансляции с языка на язык в большинстве случаев настолько трудны, что на практике этим методом пользоваться невозможно. То есть, кандидаты на звание языка UNCOL должны быть легко и эффективно реализуемыми. Они должны также допускать дальнейшее развитие как входных, так и целевых языков. Стил (Steel) [Ste61b] приводит пример гипотетического языка UNCOL, созданного до изобретения индексных регистров. Он оказался очень неудобным при работе с теми программами, которые оперировали массивами.

По сей день ни один из кандидатов на звание универсального машинно-ориентированного языка так и не получил всеобщего признания. Однако дух языка UNCOL по-прежнему живет во многих семействах компиляторов, где посредством общих промежуточных представлений реализованы различные комбинации front-end частей и back-end частей. Более того, определенные языки программирования реализованы на столь многочисленных различных архитектурах, что они практически достигли статуса универсального языка. Так, почти повсеместное распространение получил язык программирования Си [KeR78]. Аткинсон (Atkinson) и его коллеги [ADH89] описывают компилятор языка Cedar, переносимость которого достигается за счет высокоуровневой трансляции «исходник-исходник» (source-to-source), дающей на выходе код на языке Си. Получающиеся таким образом программы затем непосредственно передаются на компилятор языка Си без какого-либо дальнейшего редактирования. В данном контексте эти программы выступают лишь как промежуточные представления.

Абстрактные машины

Один из наиболее простых способов представления программы на промежуточном языке состоит в том, чтобы представить ее как последовательность команд для некоторого воображаемого компьютера, иначе именуемого абстрактной машиной (abstract machine) [PoW69, NPW72, KKM80]. Именно такая схема реализована в большинстве промежуточных языков, включая самый первый из предложенных универсальных машинно-ориентированных языков [Con58].

Помимо абстрактных машин, в качестве основы для промежуточных языков выступают также линеаризованные деревья синтаксического разбора [GaF84,DRA93a]. К тому же, существуют еще и компиляторы, которые работают через обычные языки программирования высокого уровня уже упоминавшимся методом трансляции «исходник-исходник» [ADH89].

Пик популярности абстрактных машин пришелся на начало 1970-х годов. Десять лет они широко использовались повсюду, обеспечивая переносимость программ на ряд целевых архитектур и потребляя при этом весьма скромные ресурсы. Некоторые типы архитектуры появлялись на свет в виде абстрактных машин и лишь позднее благодаря своей популярности были реализованы физически или в микрокоде. Среди наиболее известных абстрактных машин этого периода можно назвать следующие.

- **BCPL.** BCPL [Ric69] представляет собой компактный, состоящий из блоков язык программирования, предназначенный, главным образом, для системного программирования. В нем предусмотрен лишь один тип данных, Rvalue, которому язык придает различные интерпретации в зависимости от совершаемой операции. Каждый экземпляр типа Rvalue занимает единицу памяти идеального объектного компьютера (idealized object computer), видимую на уровне входного языка. Данная реализация BCPL [Ric71, RiW79] основана на абстрактной стековой машине. Семантическая модель BCPL без типов на идеальном компьютере допускает переносимую адресную арифметику в исходных программах и не зависимое от целевой машины управление стеками в компиляторе.
- **SNOBOL4.** SNOBOL4 входит в семейство обрабатывающих знаковые строки языков SNOBOL [Gri81]. Первоначально был реализован [Gri72] на безрегистровой абстрактной машине, предусматривающей только команды «память-память». Основная единица данных, обрабатываемых такой машиной, называется дескриптор (descriptor). Внутри дескрипторов кодируются все типы данных языка SNOBOL4. Однако внутренний формат дескрипторов зависит от типа целевой архитектуры, что позволяет эффективно реализовывать данный интерпретатор на различных типах аппаратных средств, но требует внесения изменений в компилятор SNOBOL4 всякий раз, когда система переносится на другую аппаратуру.
- **Pascal-P.** Универсальный язык программирования Паскаль [Wir71] широко используется и по сей день. Pascal-P, одна из его ранних реализаций, основан на гипотетическом стековом компьютере, подобном абстрактной машине, которая использовалась при реализации BCPL. Но в отличие от BCPL язык Паскаль различает несколько типов данных. В реализации Pascal-P представления этих типов на абстрактной машине (в плане их требований к памяти и выравниванию на границу блока памяти (alignment) адаптируются применительно к целевой архитектуре. Как и в случае с реализацией языка SNOBOL4, эта зависимость от целевой архитектуры препятствует прямой переносимости промежуточного языка, но зато обеспечивает эффективное распределение памяти и работу интерпретатора.
- **Janus.** В отличие от трех приведенных выше примеров, где для реализации одного входного языка на ряде целевых архитектур используется абстрактная машина, Janus определяет главные параметры промежуточного языка независимо от входного языка или целевой архитектуры, и тем самым он ближе подходит к первоначальной идее универсального машинно-ориентированного языка UNCOL. Janus не опирается на одну абстрактную машину, а описывает основные черты общей структуры (стековая машина с индексным регистром) целого семейства абстрактных машин, отличающихся друг от друга наборами команд, которые применяются для обработки конкретных конструкций входных языков. Janus успешно использовался при проектировании компиляторов для языков программирования Паскаль [Wir71] и Algol-68 [WMP69].

Появление персональных компьютеров привело в конце концов к закату эры абстрактных машин, поскольку оно означало фактическую стандартизацию машинного языка на компьютерах с низкой производительностью, которые стали доступны широким слоям пользователей. В свете квазистандартного машинного языка, больше не было нужды применять промежуточные языки для достижения переносимости программного обеспечения, тем более что интерпретированное выполнение абстрактных машин ассоциировалось с неэффективностью.

И только сейчас ситуация вновь начинает меняться. Хотя процесс стандартизации аппаратного обеспечения продолжается вот уже два последних десятилетия (так что теперь остается лишь десяток важных типов архитектур, на которые нужно переносить программное обеспечение), популярность традиционной бинарной совместимости идет на спад. Объясняется это тем, что различные реализации одной и той же архитектуры становятся настолько непохожими друг на друга, что генерировать чистый машинный код, хорошо выполняемый на всех процессорах внутри одного семейства, становится все труднее и труднее. В диссертации предлагается альтернатива: динамическая кодогенерация в период загрузки (речь об этом пойдет в одной из следующих глав). Таким образом, каждый процессор получает оптимизированные последовательности команд, а работать при этом так же удобно, как и при использовании метода бинарной совместимости.

(Продолжение следует)

Литература

[ADH89] Atkinson R., Demers A., Hauser C., Jacobi Ch., Kessler P., Weiser M. (1989) «Experiences Creating a Portable Cedar» // Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation // SIGPLAN Notices, Vol.24, No.7, p.322-329.

[BCF92] Brandis M., Crelier R., Franz M., Templ J. (1992) «The Oberon System Family» // ETH Zurich, Department of Informatics, Report No.174.

[Chu36] Church A. (1936) «An Unsolvable Problem of Elementary Number Theory» // American Journal of Mathematics, Vol.58, p.345-363.

[Con58] Conway M.E. (1958) «Proposal for an UNCOL» // Communications of the ACM, Vol.1, No.10, p.5-8.

[Cre91] Crelier R. (1991) «OP2: A Portable Oberon-2 Compiler» // Proceedings of the 2nd International Modula-2 Conference // Loughborough, England, p.58-67.

[DRA93a] (1993) «United Kingdom Defence Research Agency: A Guide to the TDF Specification» // Issue 2.1, June 1993.

[Fra90a] Franz M. (1990) «The Implementation of MacOberon» // ETH Zurich, Department of Informatics, Report No.141.

[Fra90b] Franz M. (1990) «MacOberon Reference Manual» // ETH Zurich, Department of Informatics, Report No.142.

[Fra91] Franz M. (1991) «The Rewards of Generating True 32-bit Code» // SIGPLAN Notices, Vol.26, No.1, p.121-123.

[Fra93a] Franz M. (1993) «Emulating an Operating System on Top of Another» // Software — Practice and Experience, Vol.23, No.6, p.677-692.

[Fra93b] Franz M. (1993) «The Case for Universal Symbol Files» // Structured Programming, Vol.14, No.3, p.136-147.

[GaF84] Ganapathi M., Fischer C.N. (1984) «Attributed Linear Intermediate Representations for Retargetable Code Generators» // Software — Practice and Experience, Vol.14, No.4, p.347-364.

[Gri72] Griswold R.E. (1972) «The Macro Implementation of SNOBOL4: A Case Study in Machine-Dependent Software Development» // Freeman.

[Gri81] Griswold R.E. (1981) «A History of the SNOBOLProgramming Languages» // In «History of Programming Languages» ed. by R.L.Wexelblat // Proceedings of the History of Programming Languages Conference // Academic Press, p.601-645.

[Hal65] Halpern M.I. (1965) «Machine Independence: Its Technology and Economics» // Communications of the ACM, Vol.8, No.12, p.782-785.

[KeR78] Kernighan B.W., Ritchie D.M. (1978) «The C Programming Language» // Prentice-Hall.

[KKM80] Kornerup P., Kristensen B.B., Madsen O.L. (1980) «Interpretation and Code Generation Based on Intermediate Languages» // Software — Practice and Experience, Vol.10, No.8, p.635-658.

[Mot87] (1987) «MC68030 Enhanced 32-bit Microprocessor User's Manual» // Motorola Inc., 1987.

[NAJ81] Nori K.V., Amman U., Jensen K., Nageli H.H., Jacobi Ch. (1981) «Pascal-P Implementation Notes» // In «Pascal: The Language and Its Implementation» ed. By D.W.Barron // John Wiley & Sons.

[NPW72] Newey M.C., Poole P.C., Waite W.M. (1972) «Abstract Machine Modelling to Produce Portable Software: A Review and Evaluation» // Software — Practice and Experience, Vol.2, No.2, p.107-136.

[Nat84] (1984) «Series 32000: Instruction Set Reference Manual» // National Semiconductor Corp.

[PoW69] Poole P.C., Waite W.M. (1969) «Machine Independent Software» // Proceedings of the ACM 2nd Symposium on Operating System Principles // Princeton.

[Rei89] Reiser M. (1989) <личная переписка>.

[Ric69] Richards M. (1969) «BCPL: A Tool for Compiler Writing and System Programming» // Proceedings of the 1969 Spring Joint Computer Conference // Published as «Proceedings of the AFIPS Conference Proceedings», No.34 // AFIPS Press.

[Ric71] Richards M. (1971) «The Portability of the BCPLCompiler» // Software — Practice and Experience, Vol.1, No.2, p.135-146.

[RiW79] Richards M., Whitby-Strevens C. (1979) «BCPL: The Language and its Compiler» // Cambridge University Press.

[Ste60] Steel T.B. (1960) «UNCOL: Universal Computer Oriented Language Revisited» // datamation, Vol.6, No.1, p.18-20.

[Ste61a] Steel T.B., Jr. (1961) «A First Version of UNCOL» // Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference, Los Angeles, p.371-378.

[Ste61b] Steel T.B., Jr. (1961) «UNCOL: The Myth and the Fact» // Annual Review in Automatic Programming, Vol.2, p.325-344.

[SWT58] Strong J., Wegstein J., Tritter A., Olsztyn J., Mock O., Steel T.B. (1958) «The Problem of Programming Communication with Changing Machines: A Proposed Solution» // Communications of the ACM, Vol.1, No.8, p.12-18; Vol.1, No.9, p.9-15.

[Tur37] Turing A.M. (1937) «On Computable Numbers, with an Application to the Entscheidungsproblem» // Proceedings of the London Mathematical Society, 2nd Series, Vol.42, p.230-265.

[Wel84] Welch T.A. (1984) «A Technique for High-Performance Data Compression» // IEEE Computer, Vol17, No.6, p.8-19.

[Wir71] Wirth N. (1971) «The Programming Language Pascal» // Acta Informatica, Vol.1, No.1, p.35-63.

[Wir88] Wirth N. (1988) «The Programming Language Oberon» // Software — Practice and Experience, Vol.18, No.7, p.671-690.

[WMP69] Van Wijngaarden A., Mailloux B.J., Peck J.E.L., Koster C.H.A (1969) «Report on the Algorithmic Language Algol-68» // Numerische Mathematik, Vol.14, p.79-218.