Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Средства динамической генерации кода и смешанных вычислений для языка Objective Caml

Дипломная работа студента 58 группы
Ломова Дмитрия Степановича

Научный руководитель:	.... Булычев Д.Ю.
Рецензент:	.... к.ф.-м.н. Бульонков М.А.
“Допустить к защите”, заведующий кафедрой, д.ф.-м.н., профессор	.... Терехов А.Н.

Санкт-Петербург
2001

Аннотация

Разработана иерархия средств для использования динамической генерации кода и смешанных вычислений в функциональном языке Objective Caml. Основу иерархии составляет *Dynamic Caml* — высокоуровневая библиотека динамической генерации кода с полным статическим контролем типов. Ее использование упрощают разработанные синтаксические расширения Caml.

На основе данной библиотеки построен генератор генерирующих расширений для программ на Objective Caml *AutoGen*.

Оглавление

1 Введение

Динамическая генерация кода имеет более чем тридцатилетнюю историю. Еще в 1960-х годах Кен Томпсон использовал эту технику в реализации программы поиска в текстовых файлах `grep`.

`grep` — программа поиска строк в текстовых файлах, удовлетворяющих данному регулярному выражению — являтся типичным примером программы, где использование динамической генерации кода чрезвычайно актуально. В `grep` по регулярному выражению строится функция, проверяющая, удовлетворяет ли ее аргумент этому регулярному выражению, а затем эта функция применяется ко всем строкам входного файла. Таким образом, код программы адаптируется к ее входным данным.

Если мы будем рассматривать `grep` как функцию от двух аргументов – регулярного выражения и входного файла – то процесс построения функции по регулярному выражению можно рассматривать как подстановку его в "тело" `grep` или, иначе говоря, *специалиацию* `grep` по первому аргументу.

Аналогичная ситуация возникает в задачах из самых разнообразных областей – от построения трансляторов до машинной графики. Адаптация кода программы к ее входным данным может дать выигрыш по скорости исполнения в десятки раз. Практически каждому программисту-профессионалу хоть раз приходилось писать самомодифицирующийся код.

Однако широкому распостранению динамической генерации кода мешает высокая сложность ее реализации. Написание самомодифицирующейся программы является сложным и трудоемким процессом. К тому же самомодифицирующиеся программы плохо переносимы.

Естественный путь преодоления указанных трудностей аналогичен тому пути, который прошло программирование в целом — формализация и структуризация приемов динамической генерации кода.

Для этого есть две возможности:

1. *Явное указание динамически генерируемого кода.* Программист сам строит код, котрый он хочет динамически генерировать. Задача средств динамической генерации в этом случае состоит в том, чтобы упростить эту задачу, предоставив удобный и безопасный интерфейс и обеспечив переносимость полученной программы.

2. *Автоматическая определение динамически генерируемого кода.* В этом случае адаптацию кода программы к входным данным прозрачно для программиста производит транслятор. Специализацию функций можно здесь рассматривать как одну из оптимизаций, проделываемую транслятором.

При всей привлекательности второго подхода, он обладает тем недостатком, что задача построения оптимальной специализации функции по ее аргументу алгоритмически неразрешима. Поэтому представляют интерес обе возможности. В главе 2 мы рассмотрим несколько примеров как первого, так и второго подходов.

Целью настоящей работы является построение средств динамической генерации кода для языка Objective Caml [12].

Objective Caml является функциональным языком с наиболее эффективной из существующих функциональных языков реализацией, и широко применяется в промышленной разработке программного обеспечения. Поэтому интересной представляется попытка разработать средства динамической генерации кода для него, не меняя транслятор и реализацию языка, что позволит немедленно использовать эти средства в существующих проектах.

В настоящей работе автор предлагает два таких средства.

Библиотека *Dynamic Caml* представляет собой высокоуровневую, переносимую, простую в использовании и эффективную библиотеку для конструирования и генерации кода во время выполнения. Удалось практически полностью "уложить" конструирование и генерацию кода в систему типов Objective Caml, сделав код полноценным значением языка, что позволяет транстору Objective Caml провести полный статический типовой анализ генерируемого кода. Эта библиотека являет собой яркий пример явного указания динамически генерируемого кода. Программист фактически сам пишет программу для конструирования кода, который нужно сгенерировать.

Генератор генерирующих расширений *AutoGen* позволяет построить *генерирующее расширение* функции Objective Caml, то есть для данной функции построить другую функцию, строящую ее специализацию по указанным аргументам. Получаемое генерирующее расширение генерирует код с помощью библиотеки Dynamic Caml; таким образом, генерирующее расширение выражается на том же языке, что и исходная функция.

Dynamic Caml представляет собой готовую, отлаженную систему. AutoGen, в существующей реализации, является, скорее, прототипом полноценного смешанного вычислителя, развитие которого может послужить темой дальнейших исследований.

2 Обзор существующих подходов к динамической генерации кода

Работы в области динамической кодогенерации и специализации программ ведутся в двух направлениях, очерченных в главе 1: “ручная” и автоматическая спецификация динамически генерируемого кода. Последняя тесно связана с проблематикой смешанных вычислений и частичных вычислителей.

Рассмотрим сначала несколько средств ручной спецификации кода, а затем перейдем к обзору работ в области смешанных вачислений.

2.1 Средства динамической генерации кода

Видимо, самым первым средством динамической генерации кода можно считать примитив `EVAL` в языке LISP. Подход к динамической генерации тут очень прост: так как программа на LISP является списком, примитив `EVAL` просто вычисляет список, являющийся его аргументом.

В общем случае, все средства динамической генерации кода позволяют *конструировать* код во время выполнения, а затем *вычислять* его при помощи аналогов примитива `EVAL`.

Рассмотрим две реализации средств динамической генерации кода, разработанные в последнее время.

Язык `'C` (tick C) [5] является расширением языка `C`. Ядром расширения служит библиотека, поддерживающая динамическую генерацию и содержащая набор примитивов для создания при выполнении программы некоторого промежуточного представления кода, которое может быть скомпилировано, а затем использовано. В синтаксисе языка имеются конструкции, упрощающие доступ к этим примитивам, основная из которых – оператор `'`. Например, конструкция:

```
cspec code = ' printf("%d", 27 ); ;
compile(code, TC_V)();
```

создает и исполняет код, печатающий 27.

Программа на `'C` транслируется в ANSI C и может быть скомпилирована любым компилятором `C`. При генерации кода используется высокопроизводительная система VCODE [6]. Однако данному подходу присущи некоторые недостатки, основным из которых следует признать недостаточную надежность системы. Фактически, примитивам ядра необходимо при вызове передавать типовую информацию (в приведенном выше примере это константа `TC_V`, отвечающая типу `void` результата сгенерированного кода). Эта типовая информация не может быть проверена статически, что является источником потенциальных ошибок.

Язык $\mathtt{ML}^{\Box}$(modal ML) [10] – ML-подобный язык, в котором введен типовой оператор $\Box\tau$, означающий “код, который, будучи вычисленным, выдает значение типа τ”. Значения типа $\Box\tau$ создаются оператором <u>`let`</u> <u>`code`</u> `x =`

M **in**, где M – значение типа τ. Здесь не возникает проблемы с типизацией: благодаря полной статической типизации и автоматическому выводу типов в $\mathtt{ML}^{\Box}$(как диалекте ML) все типы в динамически генерируемом коде выводятся и проверяются статически.

Однако $\mathtt{ML}^{\Box}$является лишь подмножеством ML, специально ориентированным на динамическую кодогенерацию, что, безусловно, мешает его использованию в реальных задачах.

Представляемая в данной работе система Dynamic Caml совмещает достоинства двух вышеописанных подходов: статической типизации генерируемого кода на этапе компиляции, что позволяет достигнуть полной безопасности ее использования без дополнительных типовых проверок на этапе выполнения, и оформления средств динамической генерации кода в виде библиотеки для "живого" функционального языка с эффективной реализацией, что позволяет использовать ее в реальных задачах.

2.2 Смешанные вычисления

Хорошим введением в проблематику смешанных вычислений могут служить работы [8, 7].

Общая постановка задачи смешанных вычислений звучит так:

Пусть дана программа p от двух аргументов и значение ее первого аргумента x. Необходимо построить такую программу p_x, что для любых значений второго аргумента y выполняется:

$$p(x, y) = p_x(y)$$

Специализированная версия программы p при известном значении x называется также *остаточной программой (residual program)*.

Автоматические смешанные вычисления осуществляет программа mix, определяемая следующим уравнением:

$$mix(p, x) = p_x$$

Тогда

$$(mix(p, x))(y) = p(x, y)$$

Из S^m_n-теоремы Клини [1] следует существование решения этого уравнения.

Функция mix называется *частичным вычслителем (partial evaluator)*. Простейшим смешанным вычислителем (который, собственно и предъявляется в доказательстве теоремы Клини) является

$$mix'(p, x) = \lambda y.p(x, y)$$

т.е. просто реализация карринга (curring). Однако mix', безусловно, не является оптимальным смешанным вычислителем. Следует отметить что, как показано в главе 5, оптимальный смешанный вычислитель построить невозможно, поэтому все реальные смешанные вычислители строят некоторые приближения к оптимально специализированной программе.

Как все вышеизложенное связано с динамической генерацией кода? В задачах динамической генерации кода нам нужно во время исполнения специализировать известную на этапе компиляции программу p на аргумент x, который доступен только во время исполнения. Иными словами нам надо построить *генерирующее расширение* $gen(p)$, определяемое следующим уравнением:

$$(gen(p))(x) = p_x$$

Нетрудно видеть, что

$$gen(p) = mix(mix, p)$$

то есть $gen(p)$ является специализацией mix на программу p.

В настоящее время разработано немало частичных вычислителей для различных языков программирования.В качестве примеров можно привести C-Mix [11] для языка `C`, смешанный вычислитель для функционально-декларативного языка Curry, основанный на абстрактном представлении [9], смешанный вычислитель для языка Standard ML, использующий анализ, основанный на множествах [4].

Какие требования предъявляются к реализации смешанного вычислителю, чтобы его можно было использовать для генерации генерирующих расширений?

Во-первых, он должен быть *самоприменим*: быть написанным на своем входном языке.

Во-вторых, он должен порождать для своего тела достаточно хорошую специализацию.

В-третьих, он должен порождать специалиацию программы в виде объектного кода: тогда $mix_p(x)$ будет действительно динамически генерировать код.

Так как ни один смешанный вычислитель не оптимален, второе условие достаточно трудно достижимо, и требование самоприменимости накладывает ограничения на реализацию смешанного вычислителя. Третье условие же говорит, что при получении $gen(p)$ как $mix(mix, p)$, первое всегда будет выражено на объектном языке. Поэтому, хотя генерирующее расширение и неоптимально, программисту на исходном языке недоступен его код, и он не может непосредственно оценить его качество.

Таким образом, с практической точки зрения интерес представляет непосредственная (а не через самоприменение смешанного вычислителя) реализация gen.

Автору известна только одна работа такого рода: транслятор Fabius [3]. Fabius является транслятром с ML-подобного языка, использующим так называемую *отложенную компиляцию* – для некоторых участков кода в результирующем коде порождаются динамические генераторы кода, т.е., по существу, генерирующие расширения.

Решение о том, какие данные в этих генераторах доступны, а какие задержаны, принимаются транслятором независимо от программиста. Так как транслятор порождает объектный код, текст получаемых генерирующих расширений программисту недоступен.

Разработанный автором генератор генерирующих расширений AutoGen строит генерирующее расширение для функций Caml, которые динамически генерируют код с помощью библиотеки Dynamic Caml. Получаемые генерирующие расширения выражены на Caml.

В существующей реализации AutoGen выполняет достаточно простой анализ периода связывания, основанный, фактически, на несложной абстрактной интерпретации исходной программы, не принимающей во внимание побочные эффекты выполнения и возможность появления частично доступных структур данных. Данная реализация служит лишь экспериментальным прототипом полноценного генератора генерирующих расширений.

3 Dynamic Caml и AutoGen: интерфейс пользователя

Как Dynamic Caml, так и AutoGen используют синтаксические расширения, построенные с помощью стандартного для Obective Caml средства – препроцессора CamlP4 (Caml Pre-Processor and Pretty Printer) [13]. CamlP4 являеся расширяемым $LL(1)$-распознавателем, что позволяет строить весьма сложные расширения языка.

3.1 Dynamic Caml

При построении Dynamic Caml упор был сделан на следующие характеристики:

- *Безопасность.* Прежде всего, необходимо обеспечить возможность статической проверки типов в создаваемом коде.
- *Переносимость.* Программы, использющие библитеку, должны быть легко переносимы. Интерфейс, предоставляемый библиотекой, не должен меняться от платформы к платформе.
- *Простота использования.* Необходима простая и удобная форма записи динамически генерируемого кода. Любой программист на Caml не должен испытывать трудностей в использовании библиотеки.
- *Эффективность.* Генератор кода должен быть быстрым.

Библиотка состоит из двух модулей – `Rtcg` и `Eval`, а также набора синтаксических расширений, упрощающих и дополняющих интерфейс этих модулей.

Модуль `Rtcg` содержит набор примитивов для создания кода. Модуль `Eval` предоставляет функции для генерации кода. Реализация первого платформонезависима, в то время как реализация второго меняется от платформы к платформе.

3.1.1 Конструирование кода

Библиотека построена как реализация абстрактного полиморфного типа `code`:

```
type 'a code
```

Семантика значений этого типа проста: значение типа `'a code`, будучи вычисленным, дает значение типа `'a`.

Для вычисления этих значений используется функция `eval` из модуля `Eval`, которая и генерирует код:

```
val eval : ('a -> 'b) code -> ('a -> 'b)
```

Заметим, что вычислить можно только код функции.

Таким образом, порядок использования библиотеки таков: пользователь сначала конструирует код (значение типа `'a code`), а затем вычисляет его, пользуясь функцией `eval`.

Для конструирования кода служат примитивы из модуля `Rtcg`, к рассмотрению которых мы сейчас и перейдем.

Ядро

Базовые примитивы конструирования кода таковы:

```
val quote : 'a code -> 'a
val apply : ('a -> 'b) code -> 'a code -> 'b code
val abstr : ('a code -> 'b code) -> ('a -> 'b) code
val fix : (('a -> 'b) code -> ('a -> 'b) code)
                    -> ('a -> 'b) code
val if' : bool code -> 'a code -> 'a code -> 'a code
val let' : 'a code -> ('a code -> 'b code) -> 'b code
```

`quote` Возвращает код, выдающий данное значение, т.е. код константы ("закавычивает" свой аргумент). Например, `eval (quote 2)` равно 2.

`apply` По коду функции и ее аргумента получает код применения функции к аргументу. Например:

```
eval(abstr(fun x -> apply (quote print_int x)))
```

создает функцию, печатающую свой аргумент.

`abstr` По функции от кода строит код функции, т.е. "поднимает" данную функцию над доменом `code`. Например:

```
eval (abstr (fun x -> x))
```

дает тождественную функцию. Стоит обратить внимание на отличие этого от

```
eval (quote (fun x -> x))
```

В первом случае код тождественной функции создается динамически, во втором случае – статически.

`fix` Оператор фиксированной точки. Например, код зацикливающейся функции можно получить так:

```
eval(fix(fun f -> abstr(fun x -> apply f x)))
```

`if'` Условное выражение. По коду условия и then- и else-частей строит аналог конструкции `if ... then ... else ...`.

`let'` Аналог `let` *var* `= ... in ...`.

Ограничения на использование функций ядра

Полиморфные функции. Алгоритм автоматической типизации Caml имеет одно, существенное для наших целей, ограничение. Именно, полиморфными значениями в Caml могут быть только функциональные значения. Если значение имеет полиморфный тип, отличный от функционального, его тип определяется по первому использованию.

Рассмотрим следующий пример:

```
let fcode = abstr (fun x -> x) in
            (* казалось бы, fcode - полиморфный код...*)
let f1,f2 = eval fcode, eval fcode in
            (* ...a f1,f2 - полиморфные функции *)
let i = f1 1 in (* => f1 : (int -> int) *)
               (* => fcode : (int -> int) code *)
(f1 "abc"), (f2 "abc") (* ти́повая ошибка *)
```

Причина такого ограничения в алгоритме статической типизации – трудности с выводом типов для изменяемых значений, в частности значений типа `'a ref`[1].

"Закавычивание" аргументов. Рассмотрим следующий пример:

```
let f = eval (abstr (fun x -> quote x)) in
let g = eval (abstr (fun y -> f 27)) in ...
```

Здесь `f` имеет тип `int -> int code`, поэтому данное выражение пройдет типовой контроль. Однако задумаемся о его семантике.

Функция `f` выдает код кода своего аргумента, `x`. Семантика `x` такова: это код, который, будучи вычисленным, выдает текущий аргумент `f`. Однако "текущий аргумент `f`" не существует вне тела `f`, как, например, в `g`! Будем называть такой эффект *"закавычиванием аргументов"* (*arguquoting*).

Dynamic Caml обнаруживает закавычивание аргументов при генерации кода. В приведенном примере произойдет исключение при вычислении `g`.

Операции над типами данных

Все операции над типами данных реализованы как функции, аргументами которых являются кодовые значения, вычисляющие операнды, и которые строят по ним кодовое значение, вычисляющее данную операцию.

Например:

```
val equal : 'a code -> 'a code -> bool code
```

– полиморфное структурное равенство, или

```
type int_bin_op =
  IOpSUB | IOpADD | IOpMUL | IOpDIV | IOpMOD |
  IOpAND | IOpOR | IOpXOR |
  IOpLSL | IOpLSR | IOpASR
val int_bin : int code -> int_bin_op -> int code -> int code
```

– различные бинарные операции над целыми.

Стоит обратить внимание на то, что типы в получаемом коде полностью проверяются на этапе компиляции. При генерации кода во время исполнения никаких проверок типов не требуется.

Полный интерфейс библиотеки приведен в разделе A.1. Этот примитивы библиотеки практически полностью покрывают синтаксические кон-

[1] Стоит отметить, что единственным препятствием к полиморфности кода в данном случае является ограничение алгоритма типизации. Получаемый код в самом деле полиморфен, в чем можно убедится, "обманув" алгоритм типизации с помощью `Obj.magic` (см. раздел 4.1):

```
Printf.printf "%s" ((Obj.magic f1) "abc")
```

действительно печатает `abc`.

струкции указанным способом, обеспечивающим статический контроль типов. Однако есть несколько исключений, которые мы сейчас и рассмотрим.

Кортежи

Естественный набор операций с кортежами включает две операции: *создание* кортежа длины n из n значений, и *выборка* определенного значения (известного на этапе компиляции, иначе статическая проверка типов невозможна) из данного кортежа.

В рамках системы типов Caml невозможно создать унифицированные функции для работы с кортежами, которые обеспечивали бы тот же уровень статической проверки типов, как и остальные функции библиотеки.

Однако благодаря тому, что все значения в реализации Caml имеют одинаковый размер (одно машинное слово), в Caml имеется “заглушка” статического анализатора типов — функция `Obj.magic`:

```
val magic : 'a -> 'b
```

Фактически, это приведение значение любого типа к любому типу.

Этим можно воспользваться для унифицирования операций над кортежами. Функции

```
val unsafe_tuple : 'a -> 'b
val unsafe_slice : 'a -> int -> 'b
```

предоствляют такой интерфейс. Параметром `unsafe_tuple` служит кортеж из кодов элементов кортежа, а резльтатом ее будет код кортежа. Функция `unsafe_slice` по коду кортежа и индексу строит код выборки элемента из кортежа.

Вот как могут выглядеть операции над парами:

```
let pair : 'a1 code -> 'a2 code -> ('a1 * 'a2) code =
  fun a b -> Obj.magic (unsafe_tuple (a,b))
let fst : ('a1 * 'a2) code -> 'a1 code =
  fun x -> Obj.magic (unsafe_slice x 0)
let snd : ('a1 * 'a2) code -> 'a1 code =
  fun x -> Obj.magic (unsafe_slice x 1)
```

Интерфейс библиотеки содержит полный набор функций для пар и троек. Синтаксические расширения (см. раздел 3.1.2) предоставлют полностью безопасный интерфейс к кортежам любой длины, расставляя нужным образом аннотации типов.

3.1.2 Синтаксические расширения

Синтаксические расширения, упрощающие использование Dynamic Caml, написаны с помощью CamlP4 – Caml Preprocessor and Pretty-Printer.

CamlP4 позволяет строить распознователи атрибутных $LL(1)$-грамматик. Определение грамматики можно расширять динмически. Существует полное описание грамматики Caml для CamlP4, при этом вычисляемым атрибутом при выводе является синтаксическое дерево Caml.

Расширяя это описание новыми правилами, можно вводить в язык новые конструкции, выражая их в терминах старых.

Общий синтаксис расширний

Синтаксическое расширение, вводимое Dynamic Caml, выглядит следующим образом: вводится дополнительный вид выражений:

```
expr ::= {< cdexpr >}
```

Здесь *cdexpr* – кодовое выражение. Его синтаксис в целом следует синтаксису выражений Caml, однако есть несколько расширений, специфичных для кодовых выражений.

Полный синтаксис *cdexpr* приведен в разделе A.2.

Препроцессор практически один-в-один переводит синтаксические конструции *cdexpr* в вызовы примитивов построения кода. Например:

```
{< fun x -> x * 2 >}
```

переходит в

```
abstr (fun x -> int_bin x IOpMUL (quote 2))
```

Имеются также специальные синтаксические конструкции для "закавычивания" констант и подстановки кода. Например:

```
let plus x = eval {< fun y -> y + ‘x >}
let codeplus x =
  let f x = {< x + 27 >}in
  eval {< fun y -> y + @(f x) >}
```

переходит в:

```
let plus x = eval (abstr (fun y -> int_bin y IOpADD (quote x)))
let codeplus x =
  let f x = int_bin x IOpADD (quote 27) in
  eval (abstr (fun y -> int_bin y IOpADD (f x)))
```

Кортежи

Синтаксические расширения предоставляют безопасный интерфейс к примитивам создания кортежей любой длины, расставляя нужным образом ограничения типов.

Создание кортежей происходит так: конструкция

```
{< (e1,e2,...,en) >}
```

переходит в

```
(let f : 'a1 code -> ...-> 'an code -> ('a1*...*'an) code =
  fun x1 ...xn -> unsafe_tuple (x1,...,xn)) inf)
  g1 ...gn
```

где *gi* – результат препроцессирования *ei*.

Выборка из кортежа производится с помощью сопоставления с образцом:

```
{< let (v1,...,vn) = e in e' >}
```

переходит в

```
(let' (g : ('a1 * ...* 'an) code)
  (fun temp ->
    let' (unsafe_slice temp 1) (fun v1 -> ...)))
```

3.1.3 Ограничения Dynamic Caml

На данный момент Dynamic Caml не поддерживает некоторые элементы языка, наиболее важным из которых являются взаимно рекурсиные функции.

3.1.4 Пример

В качестве примера использования динамической кодогенерации рассмотрим вычисление арифметического выражения. Пусть тип `expr` определен следующим образом:

```
type expr =
  Const of int
| Var
| Add of expr * expr
| Sub of expr * expr
| Mul of expr * expr
| Div of expr * expr
```

Тогда значение этого выражения в данной точке может быть вычислено так:

```
let rec calc e x =
  match e with
    Const i -> i
  | Var -> x
  | Add(left,right) -> (calc left x) + (calc right x)
  | Sub(left,right) -> (calc left x) - (calc right x)
  | Mul(left,right) -> (calc left x) * (calc right x)
  | Div(left,right) -> (calc left x) / (calc right x)
```

Однако если нужно вычислять одно и то же значение во многих точках, разумно сгенерировать по выражению функцию, его вычисляющую:

```
let dmlcalc e =
  let rec gen e x =
    match e with
      Const i -> quote i
    | Var -> x
    | Add(left,right) ->
        int_bin (gen left x) IOpADD (gen right x)
    | Sub(left,right) ->
        int_bin (gen left x) IOpSUB (gen right x)
    | Mul(left,right) ->
        int_bin (gen left x) IOpMUL (gen right x)
    | Div(left,right) ->
        int_bin (gen left x) IOpDIV (gen right x) in
  eval (abstr (fun x -> gen e x))
```

С использованием синтаксических расширений то же самое можно записать так:

```
let dmlcalc' e =
  let rec gen e x =
    match e with
      Const i -> {< `i >}
    | Var -> x
    | Add(left,right) -> {< @(gen left x) + @(gen right x) >}
    | Sub(left,right) -> {< @(gen left x) - @(gen right x) >}
    | Mul(left,right) -> {< @(gen left x) * @(gen right x) >}
    | Div(left,right) -> {< @(gen left x) / @(gen right x) >}
  eval {< fun x -> @(gen e x) >}
```

На самом деле, `dmlcalc` – в точности результат препроцессрования `dmlcalc'`.

3.2 AutoGen

Легко видеть, что функция `dmlcalc` из раздела 3.1.4 – это ни что иное как $mix_{\texttt{calc}}$. Таким образом, `dmlcalc` является *генерирующим расширением* для `calc`.

Программа *AutoGen* предназначена для того, чтобы автоматически строить генерирующие расширения для функций.

AutoGen реализован в виде набора синтаксических расширений. Введена специальная конструкция let codegen для определения функций, для которых надо сгенерировать генерирующее расширение. Например, для функции `calc` это определение выглядит так:

```
let rec codegen calc e `x =
  match e with
    Const i -> i
  | Var -> x
  | Add(left,right) -> (calc left x) + (calc right x)
  | Sub(left,right) -> (calc left x) - (calc right x)
  | Mul(left,right) -> (calc left x) * (calc right x)
  | Div(left,right) -> (calc left x) / (calc right x)
```

Символ ` перед аргументом `x` указывает, что данный аргумент *задержан*, т.е. что он не известен генерирующему расширению и должен стать аргументом генерируемой этим расширением функции.

3.2.1 Корректность генерирующего расширения

При построении смешанных вычислителей важную роль играет принятое определение корректности генерирующего расширения. Слишком сильные условия корректности мешают получению хороших генерирующих расширений; слишком слабые неприемлимы по вполне понятным причинам.

В AutoGen принято следующее определение корректности:

Пусть $f(x, y)$ - исходная функция, а $f_x(y)$ - генерирующее расширение для f при доступном x и задержанном y. Тогда f_x корректно, *тогда и только тогда, когда при любых x, y, при которых f останавливается, $f'(x, y) = f_x(y)$ останавливается и дает тот же результат.*

При этом поведение f' вне области определения f не специфицируется.

Данное определение корректности представляется наиболее применимым на практике.

$\mathcal{NB}$: *Следует отметить, что несмотря на то, что данное определение достаточно сильное, оно не самое сильное из возможных. Можно было бы еще потребовать, чтобы $f_{mixed}(x) = f_x$ была определена во всей области определеня $f_{curried}(x) = \lambda y.f(x, y)$, т.е. всюду. Это, однако, как мы увидим, сильно "связывает руки" смешанному вычислителю.*

3.2.2 Ограничения реализации

В существующей релизации AutoGen является, скорее, экспериментальным прототипом смешанного вычислителя, чем средством, которое можно использовать в реальных задачах.

Поэтому его реализация гарантирует корректность генерирующего расширения только для чисто функциональных программ, без побочных эффектов и присваиваний.

Также, на данный момент поддержаны не все конструкции Caml. Наиболее важной из неподдерживаемых конструкций являются взаимно рекурсивные функции, что связано с отсутствием поддержки для них в Dynamic Caml.

4 Динамическая генерация кода

В этой главе рассмотрим реализацию Dynamic Caml.

Вначале кратко рассмотрим реализацию Objective Caml, а далее перейдем к описанию реализации самой библиотеки.

Язык Objective Caml разрабатывается в Institut National de Recherche en Informatique et en Automatique (INRIA)[1], Франция. Существует компилятор язык для специально рзработанной виртуальной машины, а также практически для всех совеменных платформ.

На данный момент Dynamic Caml поддерживает только генерацию в код виртуальной машины.

В следующем разделе мы вкратце ознакомимся с реализацией Objective Caml на базе виртуальной машины.

4.1 Обзор реализации Objective Caml

4.1.1 Представление данных

Все данные в Caml делятся на два типа: прямые (“unboxed”) и косвенные (“boxed”). Прямые значения — те, которые помещаются в машинное слово (32 бита), косвенные — все остальные. К прямым значениям относятся значения типов `unit`, `int` и `char`, а также 0-арные конструкторы алгебраических типов данных. К косвенным значениям относятся кортежи, записи, значения типа `double`, массивы, фнкциональные значения (замыкания функций) и прочее.

Косвенные значения размещаются в куче. Они всегда выровнены на длину машинного слова, и их размер всегда кратен машинному слову.

Любое значение в Caml может быть представлено машинным словом. Для косвенных таким представлением является указатель в кучу, а для прямых значений — само это значение. Определить, что именно хранится в данном слове, можно по его младшему биту. Если он равен 0, это косвенное значение. Если он равен 1, то это прямое значение, и чтобы его получить, нужно знаково сдвинуть данное слово на один бит вправо[2].

Косвенные значения предстવляются блоком в куче. Первое слово блока — заголовок, содержащий 8-битовый тег блока, его размер в словах, и некоторая информация, используемая сборщиком мусора (“цвет” блока). Тег блока, представляющего значение алгебраического типа данных, определяется конструктором этого значения (каждый контруктор имеет уникальный номер внутри своего типа). Есть также несколько зарезервированных тегов: для замыканий, массивов, вещесвенных чисел, массивов вещественных и других.

[1] `http://www.inria.fr`

[2] Таким образом, диапазон целых чисел — от -2^{30} до $2^{30}-1$.

В зависимости от тега, остальные слова блока ("тело" блока) должны состоять либо из правильно построенных значений Caml, либо из произвольных данных.

Функциональное замыкание представлено блоком, первое слово которого — ссылка на код функции, а остальные слова содержат значения, составляющие замыкание.

Имеется одно исключение из общих првил организации памяти: массивы вещественных значений. В соответствии с вышеприведенными правилами, массив вещественных ссылок должен представлятся блоком, содержащим указатели на блоки, каждый из которых содержит вещественное значение. Однако в целях повышения производительности массивы вещественных чисел представляются просто как массив вещественных чисел в `C`.

4.1.2 Виртуальная машина

Виртуальная машина, используемая в Caml, является стековой машиной с аккумулятором, использующей технику шитого кода. Машина оперирует с пятью регистрами:

`accu` аккумулятор

`env` указатель на текущее замыкание

`pc` счетчик команд

`sp` указатель стека

`global_data` глобальные данные

`extra_args` используется в реализации карринга.

Набор комад машины включает команды выборки из стека, выборки из текущего замыкания, арифметические и логические команды, переходы, условные переходы, вызовы, команды выборки из блоков в памяти, создания блоков в куче, команды операций с массивами и записями и другие. Все команды берут первый внешний операнд из аккумулятора, а остальные — со стека, и помещают результат в аккумулятор.

Интерес представляет реализация карринга. Ее обеспечивает пара инструкций `RESTART`/`GRAB` *n*. Функция с несколькими аргументами всегда начинается с этих двух инструкций. Началом кода функции является адрес инструкции `GRAB`. При вызове функции команда `APPLY` *n* устанавливает регистр `extra_args` в количество переданных аргументов. Команда `GRAB` проверяет, достаточное ли количество аргументов передано. Если да, она ничего не делает. Если нет, она создает замыкание, в котором содержится указатель на текущее замыкание, значение `extra_args` и переданные аргументы. В качестве точки входа в это замыкание помещается адрес предшествующей `GRAB` инструкции, которой всегда является `RESTART`. После этого управление передается по адресу возврата. Команд `RESTART`, в свою очередь, восстанавливает параметры из переданного ей замыкания, восстанавливает прежнее замыкание и увеличивает значение `extra_args` на количество восстановленных аргументов. Затем она передает управление следующей команде, `GRAB`.

4.1.3 Сборка мусора

Куча в реализации Caml разделена на два “поколения”: “молодое” и “старое”. При заполнении молодого поколения запускается копирующий сборщик мусора, который переносит живые блоки из молодого поколения в старое. Старое поколение обрабатывается инкрементально, сборщиком мусора типа “выкрасить и выбросить” (mark-and-sweep).

При сборке мусора сборщик просматривает все объекты, теги которых меньше определенной константы `No_scan_tag`. Существуют также финализиремые блоки, при уничтожении которых запускается определенная функция (в качестве примера можно привсти значения типов `in_channel` и `out_channel`, при уничтожении которых происходит закрытие файла). Такие блоки всегда размещаются в старом поколении, так как копирующий сборщик мусора не посещает мертвые блоки.

Из кода виртуальной машины не бывает ссылок в кучу, поэтому просматривать их при сборке мусора не надо. Все косвенные константы, используемые в коде, помещаются в отдельной области памяти, на которую указывает регистр `global_data`.

4.2 Организация библиотеки

Библиотека состоит из трех частей:

- *Платформо-независимый уровень*: отвечает за создание кода. Сюда относятся все рассмотренные выше примитивы создания кода.
- *Платформо-зависимый уровень*
  - *синтез*: отвечает за предварительную генерацию кода
  - *генерация*: отвечает за окончательную генерацию кода и покадровые оптимизации
- *Распределение памяти*: отвечает за создание и удаление блоков кода, интерфейс со сбощиком мусора.

4.2.1 Платформо-независимый уровень

Тип `'a code` “внутри” библиотеки представляет собой дерево, по оторому впоследствии генерируется код. Приведем фрагмент его описания:

```
type 'a code =
   Quote of 'a
|  Var of id
|  Abstr of (any code -> any code)
|  Apply of (any -> 'a code) * any code
|  Fix of ('a code -> 'a code)
|  If of bool code * 'a code * 'a code
...
```

К сожалению, из-за отсутствия экзистенцальных типов [2] в Caml, невозможно аккуратно выразить тип `code` в системе типов Caml[3].

[3] На самом деле, узлы `Abstr` и `Fix` должны были бы выглядеть так:

Поэтому в определении типа стоят "типовые заглушки" `any code`, где `any` может быть определен любым образом (в реализации – `int`). Примитивы создания кода просто создают различные значения этого алгебраического типа, пользуясь, при надобности, `Obj.magic`:

```
let abstr f = Abstr (Obj.magic f)
let fix f = Fix f
let quote x = Quote x
...
```

4.2.2 Синтез

Общие замечания

Синтез получеат на вход дерево кода, созданное платформо-независимым уровнем, и выдает код на промежуточном языке, очень близком к коду виртуальной машины. Основное отличие промежуточного языка от машинного кода в том, что у виртуальной машины может быть нсколько разновиностей одной и той же по существу команды, а в промежуточном языке — только одна. Оптимальную разновидность выбирает уровень генерации кода.

Например, у виртуальной машины есть команда `ENVACC` n, выбирающая в аккумулятор n-ое слово текущего замыания, а также команды `ENVACC1`, `ENVACC2` и `ENVACC3`, выбирающие соответственно первое, второе и третье слова. В промежуточном языке есть только одна команда `Ienvacc of int`.

Синтез не создает полного перевода дерева, а, породив очередную команду промежуточного языка, передает ее генератору кода.

Упор при генерации кода делается на скорость генерации. Поэтому никаких "машинно-независимых" оптимизаций, таких как чистка циклов, выделение общих подвыражений или константные вычисления, не проводится — они остаются на совести программиста, создающего код.

Также не проводится анализ использования переменных: при создании функционального значения (узел `Abstr`) в замыкание попадают все определенные в данной точке переменные, а при генерации каждого узла `Let` заводится новая временная переменная на стеке. Анализ использования переменных по временным затратам сравним с общим временем генерации кода, а выигрыш от него в типичных применениях небольшой, так как создание замыкания в динамически генерируемом коде требуется нечасто.

Синтез генерирует код, обходя дерево снизу вверх. Код генерируется так, что результат генерации каждого поддерева помещается в аккумулятор.

При обходе дерева синтез поддерживает список введенных переменных с размещениями.

Далее рассмотрим наиболее интересные моменты в генерации кода.

```
type 'a code =
...
|  Abstr of ('b code -> 'c code) where 'a = 'b -> 'c
|  Fix of (('b -> 'c) code -> ('b -> 'c) code)
        where 'a = 'b -> 'c
...
```

Узел Abstr

При генерации узла `Abstr` делается следующее:

1. Определяется количество аргументов функции (для оптимизации карринга).
2. Заводятся новые переменные по числу аргументов функции.
3. Заводится новый блок кода (см. раздел 4.2.3)
4. Производится "снятие λ":
   Пусть `id` - идентификатор переменной – аргумента функции, а `f` – функция, содержащаяся в узле `Abstr`. Код этой функции можно получить так: `f (Var id)`.
5. Все известные переменные помещаются в замыкание
6. По коду функции в отведенном блоке генерируется машинный код.
7. Генерируются команды виртуальной машины для создания замыкания и регистрации его в системе распределения памяти.
8. При регистрации замыкания в него помещается список внешних значений.

Узел Quote

Если значение в узле `Quote` прямое, генерируется команда виртуальной машины `CONSTINT`.

Сложнее дело обстоит с косвенными значениями. Так как блоки кода не сканируются сборщиком мусора, в них нельзя хранить ссылки в кучу. Поэтому вместе с каждым блоком кода (создающимся при генерации кода функции) связан *список внешних значений*: кортеж из внешних ссылок данной функции. Ссылка на него помещается в первое поле замыкания, а выборка внешнего значения происходит в две команды:

```
ENVACC1
GETFIELD k
```

где k – номер значения в списке внешних ссылок.

Узел Fix

Генерация `Fix` аналогична генерации `Abstr`, за исключением того, что во второе поле замыкания при регистрации помещается ссылка на себя, и заводится еще одна переменная, с размещением во втором поле замыкания. После этого происходит "снятие FIX", аналогично "снятию λ".

Массивы

При генерации узлов доступа к массивам главной трудностью является то, что доступ к элементам массивов вещественных должен производится совсем не так, как к прочим массивам. Фактически, синтез не имеет доступа к типовой информации, и существуют ситуации, когда определить тип значения на этапе синтеза невозможно.

Однако в некоторых случаях можно частично восстановить типовую информацию по тегам значений. Например, если выборка из массива оказывается операндом целочисленной операции, то массив не может быть массивом вещественных.

Синтез отлавливает такие ситуации следующим образом: при генерации поддеревьев узел передает вниз ожидаемый тип значения, если он известен. После того, как поддерево сгенерировано, наверх, в свою очередь передается тип значения, генерируемого данным поддеревом. Для идентификации типов используются теги блоков, дополненные двумя значениями: `unboxed_tag` и `unknown_tag` – для прямых значений и неизвестных типов.

Генерация, например, выборки из массива происходит так:

```
let rec gen cd ((cg,gs) as state) expedtedtag lastresult =
  match cd with
  ...
  |  ArrayGet(array,index)->
        (* Генерация индекса массива. *)
        (* Он целый, поэтому ожидается unboxed_tag *)
      let cg,gs = gen index stat unboxed_tag false in
        (* Генерация массива. Тег неизвестен *)
      let cg,gs = gen (Obj.magic array)
            ((instr Ipush cg),gs) unknown_tag false in
      let cg, tag =
        if gs.last_tag == Obj.double_array_tag
          or expected_tag == Obj.double_tag
         (* если мы знаем что массив вещественный
          * генерируем команду выборки из вещественного *)
        then (instr IVgetfloatitem cg), Obj.double_tag
        else if gs.last_tag == unknown_tag &&
          expected_tag == unknown_tag
         (* Ничего не знаем про тип - генерируем общую
          * инструкцию с проверкой тегов *)
        then (instr IVgetvectitem cg), (unknown_tag)
         (* Что-то знаем про тип, и знаем, что он не
          * вещественный - генерируем инструкцию для выборки
          * из невещественного массива *)
        else (instr Igetvectitem cg), expected_tag in
      cg, (set_tag gs tag)
```

Подобный типовой анализ, не оказывая существенного влияния на скорость генерации, существенно повышает качество получаемого кода.

4.2.3 Распределение памяти и сборка мусора

Для Dynamic Caml пришлось написать собственную систему распределения памяти и состыковать ее со сборщиком мусора в реализации Objectiva Caml.

Блоки сгенерированного кода размещаются в пуле кода. В начале генерации кода функции генератор запрашивает в пуле блок кода, расширяемый по мере необходимости. По окончании генерации неиспользованная часть блока возвращается в пул, а самому блоку присваивается определенный индекс (handle). Здесь же определяется и точка входа в блок.

Когда возникает необходимость поместить в замыкание указатель на точку входа, вызывается специальный примитив системы распределения памяти. Ему передается ссылка на замыкание и индекс блока. Он помещает указатель на точку входа в замыкание и заносит замыкание в список ссылок на блоки кода.

Список ссылок представляет собой массив слабых ссылок[4] на блоки в куче.

Когда пул кода переполняется, запускается сборщик мусора, который проходит по списку ссылок, помечая живые блоки кода.

4.2.4 Генерация кода

На этом этапе промежуточный язык, генерируемый синтезом, преобразуется в язык виртуальной машины и производится "прошивка" кода. Здесь также выполняются несколько несложных покадровых оптимизаций.

4.3 Результаты тестрования

Проведена оценка эффективности Dynamic Caml на двух программах: табулирование функции и умножение разреженных матриц.

Тестирование проводилось на машине с процессором AMD K6-2 500MHz с 64MB памяти под управлением Linux 2.2.14 в однопользовательском режиме. Использовался Objective Caml v.3.0.

4.3.1 Табулирование функции

При использовании целочисленной арифметики динамически сгенерированный код выигрывает по времени исполнения в 5 раз, учитывая время генерации кода, а при использовании вещественной арифметики – в 3 раза уже при втором использовании динамически сгернерированного кода.

4.3.2 Умножение разреженных матриц

При умножении разреженных матриц (использовались вещественные матрицы 100×100 с пятью ненулевыми элементами на строке) динамически сгенерированный код при втором использовании выигрывает на 3%. На 10-ом переиспользовании выигрыш составляет 30%.

[4]слабая ссылка - это ссылка, которая не учитывается при поиске живых блоков в куче; если блок, на который ссылка указывает, умирает, слабая ссылка обнуляется

5 Алгоритм генерации генерирующих расширений

AutoGen является поливариантным смешанным вычислителем с вынесенным комбинированным статико-динамическим двухуровневым анализом периода связывания.

В разделе 5.1 обсуждаются общие проблемы, возникающие при построении смешанных вычислителей в контексте чисто функциональных программ и способы их решения в AutoGen.

В разделе 5.2 рассматривается алгоритм анализа периода связывания, используемый в AutoGen.

В разделе 5.3 обсуждается генерация генерирующих расширений.

5.1 Введение

5.1.1 Оптимальность генерирующего расширения

Обратившись к определению смешанного вычислителя, легко заметить, что карринг представляет собой тривиальный вид генерирующего расширения. Поэтому для смешанного вычислителя важнейшей характеристикой является качество получаемых генерирующих расширений.

В качестве неформального определения оптимального генерирующего расширения можно принять следующее:

оптимальным генерирующим расширением для данной функции будем считать такое генерирующее расширение, которое выполняет (с сохранением корректности, разумеется) все операции в теле функции, которые возможно выполнить, зная только значения доступных аргументов.

Оптимальное генерирующее расширение невычислимо. Рассмотрим пример:

```
let codegen f g ‘x =
  if g x = 1
  then true
  else false
```

Если `g x = 1` для любого `x`, то оптимальным генерирующим расширением для `f` будет `true`. Но данная проблема неразрешима, поэтому и построить оптимальное генерирующее расширение для `f` нельзя.

Таким образом, в различных алгоритмах построения генерирующего расширения мы можем искать только приближения к оптимальному генерирующему рсширению.

5.1.2 Общая схема алгоритма

Алгоритм генерации генерирующих расширений работает в два этапа: *анализ периода связывания* (АПС) и собственно генерацию. Анализ периода связывания определяет, какие значения в программе доступны, а какие задержаны. Фактически, точность проведения АПС определяет качество получаемого генерирующего расширения.

После проведения АПС становится ясно, какие выражения в исходной функции доступны, а какие задержаны. При построении генерирующего расширения вместо задержанных выражений генерируются выражения, строящие их код.

Доступные и задержанные значения

В функциях Objective Caml новые значения могут появиться:

- как параметры функций
- как связываемая переменная в `let ... in ...`, а также в цикле `for ... do ... done`.

Для связываемых в `let` и `for` переменных анализ прост — если выражение, определяющее переменную, задержано, то и переменная задержана. Для конструкции `for` выражениями, определяющими параметр цикла, служат верхняя и нижняя его граница.

Наиболее сложным представляется анализ первого случая. Для каждой подфункции исходной функции генератор генерирующего расширения должен принять два решения: a) будет ли для данной подфункции строиться генерирующее расширение или ее абстрация войдет в результирующий код; и b) если для функции будет строиться генерирующее расширение, то какие ее параметры будут доступны, а какие задержаны.

Ответ на первый вопрос мы оставим до следующего раздела, так как он тесно связан с проблемой задержанного управления, а сейчас рассмотрим второй вопрос.

Чтобы ответить на него, генератору надо просмотреть все вызовы данной подфункции и определить для каждого формального параметра, какие фактические параметры подставляются при вызове. Есть несколько очевидных вариантов:

1. все фактические параметры задержаны
2. все фактические параметры доступны
3. есть и задержанные, и доступные фактические параметры

В первом и во втором случае формальный параметр объявляется задержанным или доступным соответственно.

В третьем случае все не так просто. Генератор не может объявить параметр доступным, так как непонятно, как тогда обрабатывать задержанные фактические параметры. Генератор может объявить параметр задержанным, и передавать доступные фактические параметры через `quote`. Однако это — источник неоптимальности генерирующего расширения всей функции.

Возможны два, по своей сути одинаковых, решения. Одно из них заключается в том, чтобы сгенерировать по версии генерирующего расширения на каждый вызов (точнее, на каждую комбинацию доступных/задержанных параметров). Второе подразумевает, что генерирующее расширение будет одно, но вместе с каждым проблемным параметром будет каким-то образом передаваться флаг доступности параметра. Первое решение представляется наиболее отвечающим сути дела (это так называемый *поливариантный АПС*).

Проблема задержанного управления

Основной источник неоптимальности генерируемых расширений в различных алгоритмах смешанных вычислений — проблема задержанного управления.

Рассмотрим две функции:

```
let codegen rec f1 ‘x y =
  if x > 0
  then f1 (x-1) y
  else y
```

и

```
let codegen rec f2 ‘x y =
  if y > 0
  then f1 x (y-1)
  else y
```

Генерирующее расширение для первой из них будет:

```
let f1’ x y =
  let rec gen x y =
    if x > 0
    then gen (x-1) y
    else y in
  eval {< fun y -> @(gen x y) >}
```

Однако что произойдет для второй функции? Попробуем поступить по аналогии:

```
let f2’ x y =
  let rec gen x y =
    {< if y > 0
      then @(gen x {< y-1 >})
      else y >} in
  eval {< fun y -> @(gen x y) >}
```

Однако, в то время как `f2` останавливается при любых значениях аргументов, `f2’` никогда не останавливается!

Проблема здесь в *задержанном управлении.* За остановку функции `f2` “отвечает” задержанный аргумент `y`. Рекурсивный вызов `gen` находился в *задержанном контексте.*

Мы видим, что при рекурсивном вызове генерирующего расширения из задержанного контекста возможно его зацикливание на входных данных, при которых исходная функция не зацикливалась. Нетрудно понять, что проблема определения такого зацикливания неразрешима.

AutoGen “загрубляет” анализ задержанного управления и считает, что генерирующее расширение, рекурсивно вызываемое из задержанного контекста, всегда может зацикливаться. В этом случае вместо генерирующего расширения для соответствующей подфункции в код для исходной функции попадает абстракция этой подфункции. Безусловно, это предположение, упрощая анализ, сужает границы применимости данного алгоритма.

Доступные выражения в задержанном контексте: динамический АПС

Рассмотрим следующий пример:

```
let codegen f a i ‘b =
  a.(i+1) +                         (1)
    if b then a.(i) + 1 else 0      (2)
```

Выражение `a.(i)` в строке (2) доступно. Однако оно находится в задержанном контексте. Это означает, что его нельзя подставлять в генерируемый код как константу, что показывает следующий вызов:

```
f a (-1) false
```

Если генерирующее расширение подставит `a.(-1)` в генерируемый код, то произойдет исключение, хотя исходная функция определена для таких аргументов.

Ограничиваясь статическим АПС, все, что можно сделать в этой ситуации – это сгенерировать код для обращения к массиву, `{< ‘a.(‘i) >}`.

Однако, если перенести часть анализа на время выполнения, то ситацию можно улучшить.

В данной ситуации AutoGen при генерации `a.(i)` генерирует проверку на выход `i` за границы `a`. Если индекс правильный, генерирущее расширение вычисляет `a.(i)` и далее использует это значение. В противном случае, генерирующее расширение генерирует код обращения к массиву, `{< ‘a.(‘i) >}`.

При генерации сложения в строке (2), генерирующее расширение проверяет результат генерации `a.(i)`. Если этот результат – значение, то оно выполняет сложение и подставляет полученное значение в генерируемый код. В противном случае, оно генериует код для сложения, используя код, сгенерированный для выборки из массива.

Стоит отметить, что всегда вычислять значение `a.(i)` в строке (1) нам позволяет наше определение корректности. Если бы мы усилили его до $f_{mixed} = f_{curried}$, то пришлось бы генерировать такие же проверки и в доступном контексте.

Транзитивно замыкая описанную процедуру динамического АПС, мы получаем, что все значения и выражения исходной функции могут иметь еще и третий период связывания: *динамический*.

Кроме того, при генерации генерирующих расширений для подфункций следует принимать во внимание контекст их вызова и генерировать по версии генерирующего расширения на каждую комбинацию контекста и периода связывания параметров.

Еще один интересный вопрос заключается в том, что делать, если условие в условном выражении имеет динамический период связывания. В текущей реализации AutoGen считает, что в этом случае условие задержано, и ветви `if` оказываются в задержанном контексте.

Более правильным было бы, видимо, в данном случае генерировать две версии каждой ветки: в доступном и в задержанном контексте, и предоставить генрирующему расширению возможность подставлять одну из них в генерируемый код.

Изменяемые значения и императивные конструкции

AutoGen может сгенерировать некорректное генерирующее расширение при наличии побочного эффекта и изменяемых значений. Рассмотрим типичный пример:

```
let codegen f a i ‘x =
  a.(i) <- x;
  a.(i)
```

Так как и `a`, и `i` доступны во второй строке, AutoGen считает, что `a.(i)` тоже доступно, что неверно.

В этом, простом случае анализ не сильно усложнится от реализации такой проверки, но в более общей ситации для обработки подобных слчаев требуется тонкий анализ синонимов, что показывает, в частности, следующий пример:

```
let codegen f a i j g ‘x =
  let b = g a in
  b.(j).(j) <- x; a.(i).(j)
```

В общем случае, проблемы возникают тогда, когда целевое выражение присваивания доступно, а присваиваемое значение задержано.

AutoGen следит за такими ситуациями и выдает в этом случае предупреждение о возможной некорректности полученного генерируещего расширения.

5.2 Анализ периода связывания

Алгоритм анализа периода связывания в AutoGen реализован по принципам итеративного решеточного анализа потока данных (DFA). Однако анализируемым объектом является не граф потока управления, как в каноническом DFA, а дерево выражений Caml.

Целью анализа является атрибутирование узлов дерева следующим параметрами:

1. *Время связывания (binding time)*. Принимает одно из трех значений: *доступное (available)*, *задержанное (suspended)* и *динамическое (dynamic)*.

2. *Контекст вычислений (evaluation context)*. Может быть *строгим (strict)* если данная конструкция находится внутри задержанного управления, или *ленивым (lazy)*, в противном случае. Употребленные термины проводят аналогию со “строгой” и “ленивой” семантиками функциональных языков.

Кроме этих двух параметров, АПС вычисляет также множество необходимых сигнатур (комбинаций периодов связывания параметров и контекста вычисления вызова функций) подфункций и порождает соответствующие экземпляры тела подфункции.

Рассмотрим теперь более подробно процесс анализа. Анализ проводится итеративно. На каждой итерации значения аттрибутов узлов дерева перевычисляются по значениям аттрибутов, полученных на предыдущей итерации.

Введем несколько определений:

Определение 1 *Множество* времен связывания $B = \{available, dynamic, static\}$. *На множестве B вводится порядок:*

$$available < dynamic < suspended$$

Определение 2 *Множество* контекстов вычисления $E = \{strict, lazy\}$. *На множестве E вводится порядок: $strict < lazy$.*

Определение 3 Сигнатурой *n-арной функции f называется*

$$(e, b_1, \ldots, b_n) \in E \times B^n$$

где e - контекст вычисления вызова функции, а $b_1, \ldots, b_n$ - времена связывания ее параметров. Множество сигнатур данной функции будем обозначать S_f

Пусть $N(T)$ - множество узлов дерева T. Задачей АПС является построение по исходному дереву выражений T дерева T', полученного из дерева T добавлением к узлам вида `LetFun` поддеревьев по числу требуемых сигнатур вызовов функции, определяемой в таком узле. Также каждый узел дерева атрибутируется его временем связывания и контекстом вычисления.

Каждая итерация разбита на два этапа: обход дерева сверху вниз и снизу вверх.

5.2.1 Обход сверху вниз

При обходе дерева сверху вниз собирается информация об определения переменных и функций, а также определяется контекст и период связывания подвыражений.

Функция обработки каждого узла получает *нисходящее окружение*. Нисходящее окружение содержит следующую информацию:

- контекст вычисления;
- для каждой (определенной в данном узле) переменной: время ее связывания;
- для каждой (определенной в данном узле) функции: находится ли данный узел в определении данной функции, и, для каждой сигнатуры функции, контекст вычисления ее рекурсивного вызова. Последний может принимать следующие значения:

  none у тела функции с данной сигнатурой нет рекурсивных вызовов.

  strict рекурсивные вызовы есть и все они находятся в строгом контексте вычислений.

  lazy есть хотя бы один рекурсивный вызов в ленивом контексте вычислений.

По вычисленным на предыдущей итерации временам связывания поддеревьев, контексту вычисления узла и окружению вычисляется время связывания данного узла описанным ниже образом:

1. Вычисляется максимум из времен связывания поддеревьев.

2. Если полученное время связывания – *available*, а контекст вычислений – *lazy*, то возможны следующие варианты:

   (a) Если операция, выполняемая в данном узле, всюду определена (как, например, целочисленное сложение), то время связывания остается *available*.

   (b) Если операция определена не всюду, но известны условия, при которых она не определена (как, например, выборка из массива), то время связывания полагается *dynamic*.

   (c) В противном случае (например, применение внешней функции), время связывания полагается *suspended*.

Далее функция обработки формирует окружения для своих поддеревьев, добавляя в него определяемые в узле переменные и устанавливая контекст вычислений.

При обработке *узлов применения функций* узел атрибутируется следующими параметрами:

1. Является ли данный вызов рекурсивным вызовом функции?

2. Если да, то каков контекст рекурсивного вызова функции?

При обработке *узлов определения функции* в поддеревья узла, отвечающие различным сигнатурам функции, передаются различные нисходящие окружения. В каждом из них время связывания аргументов функции указывается соответствующим сигнатуре функции, породившей данное поддерево.

5.2.2 Обход снизу вверх

При обходе дерева снизу вверх собирается информация о вызовах функций и необходимых сигнатурах. На этом же этапе в узлах определения функций создаются новые подвыражения, соответствующие различным сигнатурам.

От листьев к корню функция обработки узла строит *восходящее окружение*, состоящее из множества требуемых сигнатур функций и контекста вычислений рекурсивного вызова каждой функции.

В дальнейшем *объединением* восходящих окружений будем называть восходящее окружение, можество сигнатур функций которого является объединением множеств сигнатур этих окружений, а контекст вычислений рекурсивного вызова каждой функции является максимумом из контекстов вычисления этой функции в этих восходящих окружениях.

Для большинства узлов восходящим окружением является объединение восходящих окружений от поддеревьев.

При обработке *узлов применения функций* строится требуемая сигнатура по временам связывания фактических аргументов и контексту вычисления. Сигнатура вычисляется по следующим правилам:

1. Если данный вызов не является рекурсивным вызовом функции, или вызов находится в строгом контексте, то сигнатура вычисляется по времени связывания аргументов и контексту вычисления данной функции.

2. Если вызов является рекурсивным вызовом и контекст вычисления задержанный, то, в зависимости от времени связывания тела функции, генерируется либо сигнатура $(lazy, suspended, \ldots, suspended)$, либо сигнатура $(lazy, available, \ldots, available)$. Эти две сигнатуры означают, что для этого вызова будет сгенерирован вызов либо абстракции данной функции (для случая задержанного тела), либо обычной функции (для случае доступного тела)

3. Если вызов является рекурсивным вызовом из экземпляра тела функции, соответсвующем ее абстракции, то независимо от контекста вычислений вызова генерируется сигнатура $(lazy, suspended, \ldots, suspended)$, соответствующая применению абстракции данной функции. Причины этого см. ниже, в разделе 5.3.

Полученная сигнатура добавляется к множеству сигнатур функции из объединения восходящих окружений от поддеревьев. Кроме того, в случае, если этот вызов является рекурсивным применением функции (что выясняется при нисходящем анализе), то контекст вычисления рекурсивного вызова применяемой функции в восходящем окружении становится равным максимуму из контекста вычисления рекурсивного вызова данной функции, вычисленного в объединении восходящих окружений от поддеревьев и контекста вычисления рекурсивного вызова.

При обработке *узлов определения функций* по множеству требуемых сигнатур от поддеревьев строятся поддеревья – экземпляры тела функции на каждую сигнатуру. Ненужные экземпляры тела при этом удаляются из дерева.

5.2.3 Завершаемость анализа периода связывания

Для доказательства того, что итеративный анализ периода связывания всегда достигает неподвижной точки, необходимо построить решетку и показать, что итерируемая функция монотонна на данной решетке.

Рассмотрим сначала дерево без поддеревьев, отвечающих сигнатурам подфункций.

Тогда структура дерева не меняется в процессе анализа, и в качестве решетки можно рассмотреть множество $\mathbf{T}$ всех возможных атрибутирований узлов данного дерева временами связывания и контекстом вычислений. Множество узлов дерева обозначим $N(\mathbf{T})$ Порядок на решетке введем следующим образом:

Определение 4 *Пусть $B_T(u) \in B$ и $E_T(u) \in E, T \in \mathbf{T}, u \in N(\mathbf{T})$ – соответственно время связывания и контекст вычисления узла u в аттрибутированном дереве T.*

Если $T_1, T_2 \in \mathbf{T}$, то $T_1 \leq T_2$ тогда и только тогда, когда для любого $u \in N(\mathbf{T})$ выполняется

$$B_{T_1}(u) \leq B_{T_2}(u),\ E_{T_1}(u) \leq E_{T_2}(u)$$

Очевидно, что функция итерации только увеличивает значения атрибутов узлов, значит она монотонна на данной решетке и итеративный анализ периода связывания на дереве без поддеревьев – тел подфункций завершается и достигает неподвижной точки.

При этом определено множество требуемых сигнатур всех подфункций.

Если далее запустить анализ тел подфункций для всех требуемых сигнатур, и отбросить тела подфункций этих подфункций, то, по тем же соображениям, анализ подфункций тоже завершается.

Продолжая далее спускаться к подфункциям, видим, что, так как глубина вложенности функций конечна, процесс анализа завершается.

Далее, наш алгоритм анализа периода связывания представляет собой анализ тел подфункций на всех уровнях, запушенный "параллельно", т.е. на каждой итерации выполняется итерация анализа на всех уровнях вложенности функций. Если какая-либо из сигнатур удаляется, то анализ ее прекращается, а если она появляется на последующих итерациях алгоритма, ее анализ начинается заново. Таким образом, и наш алгорим АПС завершается для любого входного дерева.

5.3 Генерация

Когда процесс анализа достигает неподвижной точки, все узлы дерева правильно атрибутированы их временами связывания и контекстом вычисления, и сгенерированы все экземпляры подфункций, соответствующие их сигнатурам.

Процедура генерации тогда обходит дерево снизу вверх, строя код для узлов по коду для поддеревьев.

Далее рассмотрим наиболее интересные моменты и проблемы, возникающие при генерации кода.

5.3.1 Генерация узлов с динамическим временем связывания

В этом разделе подразумевается (если не оговорено обратное), что все генерируемые выражения доступны и находятся в задержанном контексте.

Рассмотрим пример: генерацию выборки из массива `a.(i)` в задержанном контексте:

```
if i >= 0 && i < Array.length a
then Generator.Value a.(i)
else Generator.Code (array_get (quote a) (quote i))
```

Код для, например, `a.(i) + 1` будет выглядеть так:

```
match
  (if i >= 0 && i < Array.length a
  then Value a.(i)
  else Code (array_get (quote a) (quote i)))
with
    Value v -> Value (v+1)
  | Code c -> Code (int_bin c IOpADD (quote 1)) (***)
```

Видим, что при генерации "+" динамический код из поддеревьев непосредственно подставляется в динамический код для "+" (в строке `(***)`).

Однако так можно делать не всегда: рассмотрим

```
let x = a.(i) in
x + x
```

Если поступить по аналогии с вышеизложенным, получим следующее:

```
let x = (* код для a.(i) *) in
match x, x with
  Value a, Value b -> Value (a+b)
...
| Code a, Code b -> Code (int_bin a IOpADD b) (***)
```

Тогда в строке (***) в динамически получаемом коде выборка дублируется.

Чтобы этого избежать, заводится временная переменная в коде для значения выборки

```
match (* код для a.(i) *) with
  Value temp -> let x = Value temp in (* код для x+x *)
| Code temp ->
    let' temp (fun x -> let x = Code x in (* код для x+x*)
```

При таком способе генерации дублирования динамически генерируемого кода удается избежать.

В имеющейся реализации AutoGen никак не оптимизирует код, сгенерированный для динамичсекого периода связывания. Для дальнейшей оптимизации достаточно вычисления константных выражений на этапе компиляции (например, в обеих ветвях `match...with` из последнего примера уже известно, является ли `x` значением (`Value`) или кодом (`Code`), так что дополнительня проверка в коде для `x+x` не требуется).

5.3.2 Генерация определений функций

Для каждой функции различные ее сигнатуры порождают следующие виды экземпляров:

1. *Абстракцию* функции – экземпляр, содержащий свой рекурсивный вызов в задержанном контексте вычислений. Для каждой функции существует не более одной абстракции.

2. *Экспликации* функции – экземпляры, возвращающие код тела функции по кодам и значениям ее доступных и задержанных параметров.

Рассмотрим порождение этих видовэкземпляров на примере функции возведения в степень.

```
let rec power n x =
  if n = 0
  then 1
  else x * (power (n-1) x) in E
```

Вот все возможные сгенерированные экземпляры функции (для простоты все вызовы примитивов модуля `Rtcg` заменены на синтаксические расширения):

```
let'
  {< fix power'1 ->
    fun n x -> (* n и x - задержаны *)
      if n = 0 then 1
```

```
    else x * (power'1 (n-1) x) >}(***)
  (fun power'1 ->
    let rec power'2 n x = (* n -доступна, x - задержана *)
      if n = 0 then quote 1
      else {< x * @(power'2 (n-1) x) >}
    and power'3 n x = (* x - доступна, n - задержана *)
      {< if n = 0 then 1
        else `x * (power'1 (n-1) `x) >}in ...
```

При генерации экземпляров различных сигнатур возникает одна проблема: хотя AutoGen и не обрабатывает взаимно-рекурсивные функции, экземпляры одной и той же функции с различными сигнатурами могут быть взаимно-рекурсивны. Взаимная рекурсивность экспликаций не составляет проблемы, так как все экспликации находятся в одной и той же конструкции `let rec ...and ...`. Но что делать, если абстракция и экспликация взаимно рекурсивны (что возможно, если в абстракции присутствуют несколько рекурсивных вызовов с разными сигнатурами)?

AutoGen решает эту проблему просто: все вызовы в абстракции вызывают ее же (т.е. абстракция существует "сама по себе" и вообще не вызывает экспликаций). Данное поведение "загрубляет" генерацию абстракции.

5.3.3 Генерация вызовов функций

При генерации вызовов функций вызывается экземпляр с сигнатурой, соответствующей данному вызову.

Единственную трудность здесь представляет то, что при генерации экспликаций нельзя напрямую передавать задержанные и динамические параметры в экспликацию: чтобы избежать дублирования динамически генерируемого кода, необходимо в динамически генерируемом коде завести для них временные переменные, и уже их передавать в качестве кода задержанных и динамических параметров в экспликацию.

6 Заключение

Результатом дипломной работы явилось создание библиотеки динамической кодогенерации Dynamic Caml и прототипа смешанного вычислителя AutoGen.

Dynamic Caml – библиотека динамической кодогенерации для “живого” функционального языка, которая может быть использована в реальных задачах.

Библиотека имеет простой и безопасный интерфейс. Реализация библиотеки, по результатам тестирования, достаточно эффективна.

Синтаксические расширения для Objective Caml, разработанные стандартным для этого языка способом, упрощают и расширяют интерфейс библиотеки.

Принципы построения библиотеки могут быть использованы при разработке подобных средств для других функциональных языков.

На базе библиотеки Dynamic Caml построен прототип смешанного вычислителя AutoGen, оперирующий в терминах того же языка, для которого он написан.

Вычислитель проводит базовый анализ периода связывания значений и выражений Caml, основываясь в основном, на структуре исходной программы.

Данный вычислитель может служить основой для дальнейших разработок в области построения смешанных вычислителей для Objective Caml и реализации более сложных методов анализа периода связывания и построения генерирующих расширений.

A Техническая документация

A.1 Интерфейсные файлы

В данном разделе приведена документация к библиотеке, автоматически извлеченная из комментариев к интерфейсным файлам модулей.

A.1.1 Модуль `Rtcg`

Module `Rtcg` contains primitives for code creation.

```
type 'a code
```

Basic abstract type. Semantics is the following: value of type `'a code`, being evaluated, produces a value of type `'a`.

Core functions

```
val abstr : ('a code -> 'b code) -> ('a -> 'b) code
```

Function abstraction. “Raises” a function over `code` domain.

```
val apply : ('a -> 'b) code -> 'a code -> 'b code
```

Creates code of function application.

```
val quote : 'a -> 'a code
```

Quotes arbitrary OCaml value in code.

```
val fix : (('a -> 'b) code -> ('a -> 'b) code) ->
    ('a -> 'b) code
```

Fixpoint operator for recursive functions.

```
val let' : 'a code -> ('a code -> 'b code) -> 'b code
```

Functional representation of a `let` construct.

Booleans and conditionals

```
val if' : bool code -> 'a code -> 'a code -> 'a code
```

Equivalent for `if`-statement. Creates code that evaluates either second or third argument depending on whether result of evaluating first argument is `true` or `false`

```
val bor : bool code -> bool code -> bool code
```

Boolean “or”.

```
val band : bool code -> bool code -> bool code
```

Boolean “and”.

```
val bnot : bool code -> bool code
```

Boolean “not”.

All boolean operations have McCarthy semantics.

Equality and comparisons

```
val eq : 'a code -> 'a code -> bool code
```
Test for physical equality (like OCaml ==).

```
val neq : 'a code -> 'a code -> bool code
```
Test for physical inequality (like OCaml !=).

```
val equal : 'a code -> 'a code -> bool code
```
Test for structural equality (like OCaml =).

```
type cmp =
    CmpEQ | CmpNEQ | CmpLT | CmpGT |
    CmpLE | CmpGE
```
Comparisons

```
val compare : 'a code -> cmp -> 'a code -> bool code
```

Tuples

```
    Tuples
val tuple2 : 'a1 code -> 'a2 code -> ('a1*'a2) code
val tuple3 : 'a1 code -> 'a2 code -> 'a3 code ->
    ('a1*'a2*'a3) code
```
Functions for creating tuples of length 2 and 3.

```
val slice20 : ('a*'b) code -> 'a code
val slice21 : ('a*'b) code -> 'b code
val slice30 : ('a*'b*'c) code -> 'a code
val slice31 : ('a*'b*'c) code -> 'b code
val slice32 : ('a*'b*'c) code -> 'c code
```
Accessing tuple elements — tuples of length 2 and 3.

```
val unsafe_tuple : Obj.t -> Obj.t
```
Unsafe function for creating code creating tuples.
An argument must be OCaml memory block containing codes of all elements of a tuple. Result is code creating tuple with that elements.
This function should never be used directly.

```
val unsafe_slice : Obj.t -> int -> Obj.t
```
Unsafe function for creating code for getting elements of tuples. Arguments are tuple code and element to get (first has index 0). Result is a code for obtaining an element of a tuple.
This function should never be used directly.

Imperative

```
val seq : unit code -> 'a code -> 'a code
```
Create `code` that sequence evaluation of two `code`s.

```
val lose : 'a code -> unit code
```
"Loses" result of `code` so that it can be sequenced.

```
type direction = DTo | DDownTo
```
Direction of `for` loop.

```
val for' : int code -> direction -> int code ->
    (int code -> unit code) -> unit code
```

For loop. 3rd argument is applied for values starting from value of first argument, to the value of third argument increasing or decreasing depending on second argument.

```
val while' : bool code -> unit code -> unit code
```

While loop.

Arrays

```
val array_get : 'a array code -> int code -> 'a code
```

Creates `code` to get an element from array

```
val array_set : 'a array code -> int code -> 'a code ->
    unit code
```

Creates `code` to set an element of an array

```
val array_length : 'a array code -> int code
```

Creates `code` that calculates array length

Strings

```
val string_get : string code -> int code -> char code
```

Creates `code` to get a character from a string

```
val string_set : string code -> int code -> char code ->
    unit code
```

Creates `code` to set a character of a string

References

```
val ref_create : 'a code -> 'a ref code
```

Creates `code` of a reference

```
val ref_value : 'a ref code -> 'a code
```

Gets value of reference

```
val ref_set : 'a ref code -> 'a code -> unit code
```

Assign a value to a reference

Integers

```
type int_bin_op =
    IOpSUB | IOpADD | IOpMUL | IOpDIV | IOpMOD |
    IOpAND | IOpOR | IOpXOR |
    IOpLSL | IOpLSR | IOpASR
```

Binary operations with integers

```
type int_un_op = IUnNEG | IUnNOT
```

Unary operations with integers

```
val int_bin : int code -> int_bin_op -> int code -> int code
```

Code of binary integer operation

```
val int_un : int_un_op -> int code -> int code
```

Code of unary integer operation

`val int_cmp : int code -> cmp -> int code -> bool code`
Code of integer comparison

Floats

```
type flop =
    FLOP_ADD | FLOP_SUB | FLOP_MUL | FLOP_DIV |
    FLOP_MOD | FLOP_POWER
```

Floating-point operations

`val float_bin : float code -> flop -> float code -> float code`
Code of floating-point binary operation

`val float_neg : float code -> float code`
Code of floating-point negation

A.1.2 Модуль `Eval`

Module `Eval` contains primitives for code evaluation.

`val init : unit -> unit`
Initialize code generation system.

`val uninit : unit -> unit`
Uninitialize code generation system. Will be called automaitically at exit if `init` was called before. Use this function with caution — it freeze internal code generation pool, so if any dynamically generated function is applied after `uninit` is called, results will be disastrous.

`val eval : ('a -> 'b) code -> ('a -> 'b)`
Evaluate given code to obtain a function.

`val set_unsafe : bool -> unit`
Set code generation mode to safe or unsafe. In safe mode, no bounds checking for arrays or strings is peformed. This is equivalent to `ocamlc` "`-unsafe`" parameter.

Default mode is safe.

A.2 Синтаксические расширения

The following are syntax definitions for code expressions.

A.2.1 OCaml expressions

Definition of OCaml expression is extended to include the following:
expr ::= `{<` *cdexpr* `>}`

Here, *cdexpr* is *code expression* as defined further in this section.

A.2.2 Code expressions

Functions

```
simple_patt ::= identifier | _
cdpatt ::= simple_patt | ( [(simple_patt ,)*] simple_patt )
cdexpr ::= fun cdpatt* -> cdexpr
cdexpr ::= fix identifier -> cdexpr
cdexpr ::= cdexpr cdexpr
```

`let` constructs

```
cdexpr ::= let cdpatt cdpatt* = cdexpr in cdexpr
cdexpr ::= let rec identifier cdpatt* = cdexpr in cdexpr
```

Quoting and embedding

```
cdexpr ::= ` expr
cdexpr ::= @ expr
```

Conditionals and imperative constructs

```
cdexpr ::= if cdexpr then cdexpr else cdexpr
direct ::= to | downto
cdexpr ::= for identifier = cdexpr direct cdexpr do cdexpr done
cdexpr ::= cdexpr ; cdexpr
cdexpr ::= while cdexpr do cdexpr done
```

Operators

```
cdexpr ::= cdexpr binop cdexpr
binop ::= ==|!=|=|||&&|+|-|*|/|lor|land|lor|mod|+.|-.|*.|/.|%.|**|,|;|:=
cdexpr ::= unop cdexpr
unop ::= -|!|ref
```

Operator precedence.

Operator precendence in code expressions is the same as in conventional Objective Caml expressions.

Miscellanous

```
cdexpr ::= ( cdexpr )
cdexpr ::= cdexpr .( cdexpr )
cdexpr ::= cdexpr .[ cdexpr ]
cdexpr ::= cdexpr .( cdexpr ) <- cdexpr
cdexpr ::= cdexpr .[ cdexpr ] <- cdexpr
cdexpr ::= integer | float | string
cdexpr ::= identifier
```

Литература

[1] **Хартли Роджерс**. *Теория рекурсивных функций и эффективная вычислимость* — М., 1972

[2] **Luca Cardelli**. *Typeful Programming // Formal Description of Programming Concepts* — Springer-Verlag, 1991

[3] **Mark Leone, Peter Lee**. *Deferred compilation: The Automation of Run-time Code generation* — USA, 1993

[4] **Karoline Malmkjaer, Nevin Heintze, Olivier Danvy**. *ML Partial Evaluation Using Set-Based Analysis* — USA, 1994

[5] **Dawson R. Engler, Wilson C. Hsieh, M. Frans Kaasjoek**. *'C: A Language for High-level, Efficient and Machine-Independent Dynamic Code Generation. // Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages.* — USA, 1996

[6] **Dawson R. Engler**. *VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. // Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages.* — USA, 1996

[7] **Neil D. Jones**. *Introduction to Partial Evaluation.// ACM Computing Surveys, Vol. 28, No.3 Sept. 1996* — USA, 1996

[8] **Михаил Бульонков**. *Смешанные вычисления* — Издательство НГУ, 1997

[9] **Elvira Albert, Michael Hanus, Germán Vidal**. *Using an Abstract Representation to Specialize Functional Logic Programs* — Spain, 1998

[10] **Philip Wickline, Peter Lee, Frank Pfenning**. *Run-time Code Generation and Modal ML. // ACM SIGPLAN Conference on Programming Languages Design* — Pittsbugs, USA, 1998

[11] **Henning Makholm**. *Specializing C: an introduction to principles behind C-Mix* — Denmark, 1999

[12] **Xavier Leroy *et al.***. *Objective Caml version 3.0* — INRIA, France, 2000

[13] **Daniel de Rauglaudre**. *Caml Preprocessor and Pretty-printer version 3.0* — INRIA, France, 2000

[14] **Dmitry Lomov, Anton Moscal**. *Dynamic Caml version 0.1* — `http://oops.tepkom.ru/dml`