*Модел. и анализ информ. систем.* Т. **18**, № **4** (2011) 45–55

УДК 004.415.5+004.416.2

# Статический анализ с использованием систем типов и эффектов на основе LLVM

Беляев М. А., Цесько В. А.

*Санкт-Петербургский государственный политехнический университет*

*e-mail: belyaev@kspt.ftk.spbstu.ru* *tsesko@kspt.ftk.spbstu.ru*

*получена 6 октября 2011 года*

**Ключевые слова:** статический анализ, системы типов и эффектов, поиск дефектов, SSA

Описано разрабатываемое средство статического анализа программного обеспечения. Основной идеей данного средства является использование систем типов и эффектов для статического анализа реальных программ. Средство использует формат промежуточного представления LLVM в качестве входного представления программы, таким образом, давая возможность анализировать программы на любых языках, поддерживаемых системой LLVM. Разбор указанного формата осуществляется встроенным парсером, позволяющим осуществить формирование внутренней модели программы, схожей с моделью LLVM. Целью создания описываемого средства является исследование возможностей построения методов статического анализа программ на основе известных алгоритмов, использующих системы типов и эффектов, путём применения этих алгоритмов к модели и опосредованно к исходному коду.

## 1. Введение

Статический анализ программного обеспечения — это один из основных способов статического предсказания поведения программы при запуске. Основной целью такого предсказания является выявление программного кода, который может приводить к различным ошибкам на этапе выполнения.

Наиболее часто применяемые языки программирования, особенно чувствительные к ошибкам программистов, включают в себя языки C и C++ (в связи с наличием в этих языках арифметики указателей, непроверяемых преобразований типов, битовой арифметики, небезопасных функций с переменным числом параметров и ряда других особенностей). Этот вывод подтверждается статистическими данными ряда источников, в том числе ежегодным исследованием компании Coverity [1].

Существует три наиболее известных и широко используемых подхода к статическому анализу: абстрактная интерпретация, анализ потока данных и анализ на

основе вывода. Наиболее широко применяемыми в коммерческих продуктах являются первые два подхода. На их основе разработано множество средств, в том числе IBM Rational Software Analyzer, Coverity and Frama-C. Это связано с наличием большого количества опубликованных алгоритмов анализа и относительной простотой их реализации. Описание средства Aegis [4], разрабатываемого лабораторией программно-аппаратных разработок кафедры компьютерных систем и программных технологий СПбГПУ и использующего один из этих подходов (абстрактную интерпретацию), содержит достаточно полный обзор средств на их основе, а также основных алгоритмов и программных моделей. Данное исследование посвящено менее известному и реже используемому подходу — анализу на основе вывода и одной из его разновидностей — системам типов и эффектов. Несмотря на отсутствие поддержки со стороны коммерческих средств, данный подход имеет ряд преимуществ по сравнению с другими — в частности, он основан на логическом выводе, а не на анализе потока данных, и использует формальное описание, поэтому возможно формальное доказательство корректности конкретного вида анализа, построенного на основе данного подхода.

Подход к статическому анализу, обычно называемый «системы типов и эффектов», основан на расширении существующей в языке системы типов путём аннотации типов дополнительной информацией, выражающей необходимые для анализа семантические свойства программы. Более подробно данный подход описан в разделе 3. Языки C и C++, наиболее чувствительные к ошибкам и поэтому наиболее нуждающиеся в анализе, не могут использоваться в качестве входного языка выражений для такого анализа в связи с крайне сложными синтаксисом и семантикой. Поэтому необходимо формировать промежуточное представление на основе исходного кода на этих языках, которое может быть использовано в данном качестве. Таким представлением может служить программная модель, способная выразить полную семантику языка C и при этом предоставляющая в простом виде информацию, необходимую и достаточную для типизации. В качестве такой модели было выбрано представление на основе статического однократного присваивания (SSA-представление).

Представление на основе статического однократного присваивания (Static Single Assignment Form, SSA) обладает одноимённым свойством — значение каждой переменной в этом представлении присваивается только один раз [2, 13]. Система компиляции LLVM использует как раз такое представление, называемое «биткодом». Это представление лежит в основе модели, используемой в описываемом средстве, что позволяет избежать написания полноценного транслятора языка C с нуля.

Описываемое средство состоит из двух основных компонент: парсера биткода (`llvm-parser`) и библиотеки анализа (`llvm-analyzer`). Средство реализовано на языке Haskell с использованием библиотеки `sbv`, предоставляющей интерфейс к SMT-решателю. В разделе 2 приведено краткое описание парсера (дополнительная информация, а также исходный код парсера могут быть получены на странице проекта [8]). В разделе 3 описаны основные идеи, лежащие в основе библиотеки анализа.

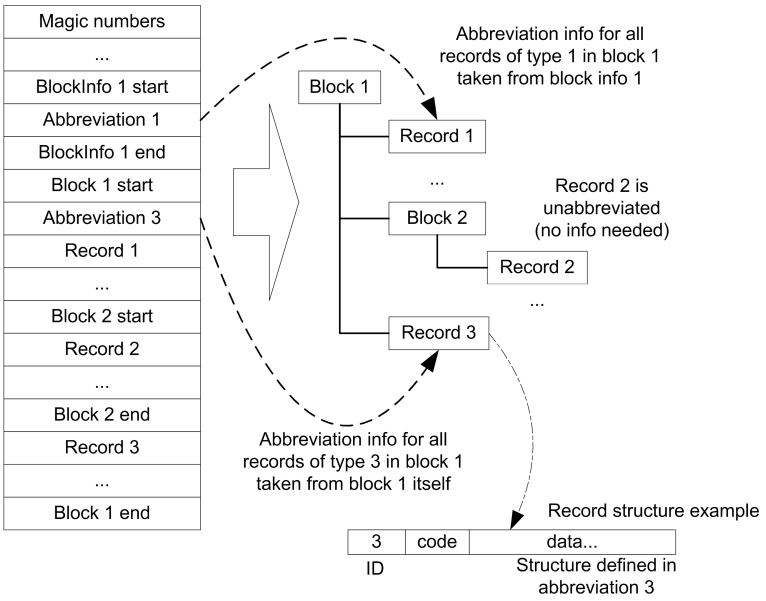


Рис. 1. Общая структура формата битового потока

# 2. Парсер промежуточного представления LLVM

Контейнер данных, используемый биткодом LLVM, называется форматом битового потока (Bitstream File Format, BFF). Контейнер отделен от непосредственных данных и может использоваться для хранения любой другой иерархической информации в двоичном виде. В следующем подразделе приведено краткое описание формата и части библиотеки, осуществляющей его чтение.

## 2.1. Разбор формата битового потока

Формат битового потока представляет собой обобщённый контейнер для эффективного хранения различного рода битовых иерархических данных посредством механизма так называемых «аббревиатур», позволяющих эффективно располагать данные в минимальном объёме памяти. Файл в таком формате представляет собой набор записей, состоящих из «примитивов». Дополнительная информация о формате приведена в [6]. Все записи могут быть поделены на две группы: служебные записи, используемые на этапе разбора и не содержащие непосредственно целевых данных, и собственно записи с данными. Общая структура файла в таком формате приведена на рис. 1.

Описываемое средство использует подход на основе монад для обработки сервисной информации, что позволяет реализовать весь этап разбора на чистом Haskell и значительно уменьшить общую сложность разбора. Подход основан на использовании пакета `binary` и позволяет не допустить побочных эффектов при разборе и уменьшить количество ненужных вычислений с учётом ленивой стратегии выполнения языка Haskell. Для обработки ошибок и разделения процесса разбора на уровни используются трансформеры монад. Результатом данного этапа работы программы

является древовидная структура данных на языке Haskell, соответствующая структуре блоков и записей изначального формата.

## 2.2. Построение программной модели

Промежуточное представление LLVM является ассемблеро-подобным языком с типами и может рассматриваться как типизированный ассемблер (пример использования такого языка для анализа приведён в [12]). Программа представляет собой единственный блок данных в файле битового потока.

Модуль программы включает в себя:

- вспомогательную информацию о модуле, такую как: имя модуля, целевая платформа, используемые сборщики мусора и т. д.;
- таблицу типов — список всех типов, присутствующих в модуле. Любая другая информация о типах внутри модуля вводится путём индексирования таблицы типов. Таблица типов представляет собой отдельный блок данных;
- символьные таблицы — отображения символьных имён различных сущностей внутри модуля на соответствующие индексы в таблицах типов или значений. Таблицы символов для типов и для значений хранятся как отдельные блоки данных внутри модуля;
- константы уровня модуля — хранятся в отдельном блоке данных;
- глобальные переменные и их псевдонимы — хранятся как отдельные записи;
- объявления и определения функций — хранятся в виде записей. Тела функций хранятся в отдельных блоках.

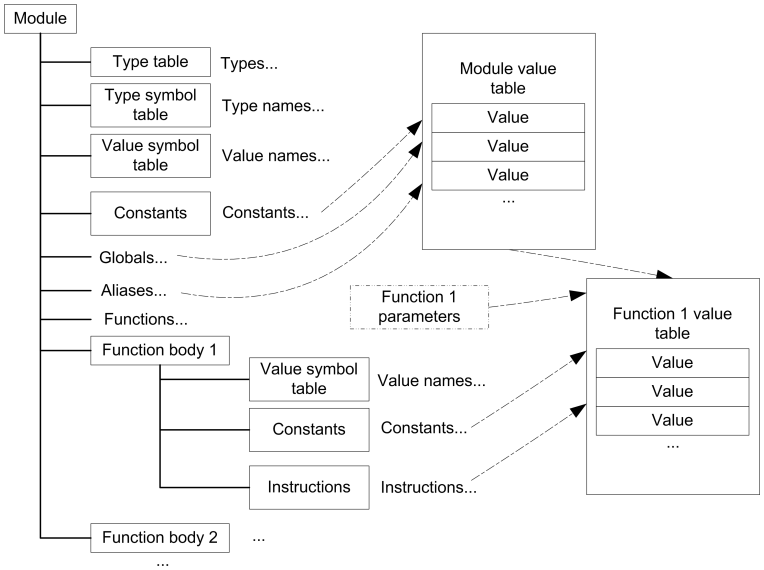


Рис. 2. Общая структура программной модели и процесс её преобразования

С целью упрощения итоговой модели вспомогательная и символьная информация при разборе отделяются от остальной модели, которая, в свою очередь, комбинируется в единую таблицу значений уровня модуля.

Тела функций представляют собой достаточно сложные сущности, хранящиеся в отдельных блоках внутри модуля. Каждый из них может включать:

- символьные таблицы уровня функции (аналогичные соответствующим таблицам на уровне модуля);
- константы уровня функции (локальные константы) в отдельном блоке;
- набор инструкций, непосредственно составляющий тело функции;
- отладочную информацию, в том числе привязку инструкций к их расположению в файле с исходным кодом.

Аналогично структуре каждого модуля, структура тела каждой функции преобразуется путём отделения вспомогательной и символьной информации и формирования единой таблицы значений уровня функции, включающей в себя:

- все значения уровня модуля;
- параметры функции как значения (в модели LLVM параметры функции отсутствуют, но при этом участвуют в индексации значений);
- константы уровня функции;
- собственно инструкции LLVM.

Результатом данного процесса является программная модель, отличная от оригинального SSA-представления в том, что она содержит дополнительные значения (подробнее они описаны в разд. 3.2.). Общая структура программной модели и процесс её преобразования приведены на рис. 2.

# 3. Анализ модели

В данном разделе описаны как уже реализованные идеи и принципы, так и находящиеся на стадии разработки. Главной целью, поставленной при проектировании библиотеки анализа, было формирование набора конфигурируемых средств, дающих пользователю библиотеки возможность реализовывать собственные виды анализа и тестировать их работу на реальных программах.

## 3.1. Системы типов и эффектов

Общий подход систем типов и эффектов состоит в ассоциировании типов программам. Как правило, типы описывают (с точки зрения некоторых семантических аспектов) данные, передаваемые в программу или возвращаемые из неё, в то время как эффекты описывают возможные побочные эффекты, возникающие в процессе выполнения (ошибки, исключения и т. д.). Ассоциирование типов с программой осуществляется на основе правил вывода, которые оперируют отношениями типизации. Отношения типизации обычно описываются в виде четверичных отношений вида $\Gamma \vdash p : \tau \& \varphi$, где $p$ — программа, $\tau$ — тип, $\varphi$ — набор эффектов, а $\Gamma$ — типовое окружение, которое отображает все свободные переменные в $p$ на их типы.

Для работы с рекурсивными выражениями и циклами система типов, как правило, включает в себя отношение частичного порядка между типами вида $\tau_1 \sqsubseteq \tau_2$.

$\tau_2$ часто называется *подтипом* типа $\tau_1$, а всё отношение — отношением подтипов. Существует большое количество теоретических работ, в которых описаны различные анализы с использованием данной техники [9, 10, 11], обычно оперирующие различными разновидностями языка ML и обладающие некоторыми формально доказанными свойствами.

## 3.2. Практическое применение

Определение системы типов и эффектов состоит из:

- языка выражений $e$ (фактически не является частью системы, но без языка систему определить нельзя);
- системы типов, определяющей различные свойства выражений;
- набора правил вывода, описывающих, как различные типы выражений зависят друг от друга;
- отдельных или интегрированных правил для описания эффектов, моделирующих различные ситуации, которые могут произойти в процессе работы программы.

В данной работе язык выражений $e$ всегда основывается на значениях биткода, описанных выше. Несмотря на то, что язык промежуточного представления LLVM является императивным, его свойство статического однократного присваивания гарантирует, что каждой переменной значение присваивается только один раз, таким образом, однозначно определяя любые зависимости по данным между переменными, не включающими в себя указатели [7]. Благодаря этому, возможно рассмотрение данного представления как некоторого функционального языка (исключая конструкции, работающие с указателями). К сожалению, использование указателей неизбежно, поскольку все переменные в LLVM выделяются с помощью инструкций `Alloca` или `Malloc`, возвращающих указатели. Другой проблемой является то, что некоторые инструкции не возвращают никаких значений и не могут рассматриваться как выражения. Отдельно следует рассмотреть инструкцию `Call`, которая вызывает внешнюю функцию, поведение которой неизвестно, и которая может модифицировать свои параметры и при этом не возвращать никакого значения. Для решения этих проблем в описываемой библиотеке использованы две концепции — *метазначения* и *цепочки псевдонимов*. Первая концепция связана с инструкциями `Store` и `Call`: инструкции сохранения значения по указателю не возвращают никаких значений, в представленной модели они возвращают модифицированный указатель; вызов функции может изменить значения параметров, и в модели добавляются метаинструкции для каждого параметра функции после вызова `Call`, виртуально «возвращающие» соответствующие изменённые параметры.

Например, инструкция вида `Store(a,b)`, где `b` — указатель, а `a` — значение, возвращает модифицированный указатель `b`, а вызов `f(a,b,c)` преобразуется в 4 значения: инструкцию `Call(f)` и изменённые значения `a`, `b` и `c`. Все эти преобразования никак не изменяют семантику изначальной программы, они лишь добавляют новые значения для распространения информации о типах.

Вторая концепция (цепочки псевдонимов) состоит в том, чтобы применить версионирование (основу формата SSA) к переменным-указателям. Любое значение, зависящее от некоторого указателя (например, инструкция `Load`), преобразуется так, чтобы ссылаться не на единственный исходный указатель, а на набор потенциально возможных последних версий этого указателя (например, набор инструкций `Store`) назад по потоку управления. Для осуществления данного преобразования необходима информация о псевдонимах указателей и о потоке управления. Получение этой информации — крайне сложная задача (фактически контекстно-независимый анализ указателей является NP-полной задачей [3]) и в рамках описываемого подхода осуществляется только для простейших случаев (к примеру, анализ указателей только первого уровня), в дальнейшем планируется реализация более точных решений этих задач или использование готовых алгоритмов.

Нужно отметить, что для большинства видов анализа полная система значений в биткоде LLVM является чересчур сложной и избыточной. В этом случае следует использовать некоторое упрощение набора значений в рамках конкретной задачи или класса задач.

В качестве примера рассмотрим простейший анализ, целью которого является поиск возврата неинициализированных значений из функции. Определение анализа включает в себя:

- уровень представления:

```
data UDRepresentation =
    Constant | Create | Delete [ValIndex] | Skip | Load [ValIndex] | Store [ValIndex] ValIndex |
        Return (Maybe ValIndex) | JoinAnd [ValIndex] | JoinOr [ValIndex] | Call ValIndex
        [ValIndex]
```

- отображение значений изначального языка в уровень представления;
- систему типов:

```
data DefType =  DDefined | DUndefined | DMaybeDefined | DPointerTo DefType
```

- кодирование типов и эффектов в логические значения:

```
-- Encoded type
type SType = (SBool,SBool,SWord8)
-- Encoding
encode :: DefType → SType
encode (DPointerTo t) = case (encode t) of
                            (u,d,ptrs) → (u,d,ptrs + 1)
encode (DDefined) = (false,true,0)
encode (DUndefined) = (true,false,0)
encode (DMaybeDefined) = (true,true,0)
-- Domain of encoded types
-- (the only illegal value is (false,false,_))
isInDomain :: SType → SBool
isInDomain (u,d,ptrs) = u ∨ d
-- Partial ordering (on encoded values for optimization)
-- Actually means that for any pointer level
-- DMaybeDefined ≥ DDefined and
-- DMaybeDefined ≥ DUndefined
pge :: SType → SType → SBool
pge a@(a0,a1,a2) b@(b0,b1,b2) =
    a2 ≡ b2 ∧ ( (a0 ∧ a1) ∨ (a0 ≡ b0) )
-- We encode effects into smallest integer values possible
type SEffect = SWord8
```

- возможности для описания побочных эффектов функций, вызываемых в процессе вычислений. Невозможно вывести поведение внешних функций, исходя

из имеющейся информации, поэтому это поведение должно быть описано вручную, по крайней мере, для наиболее часто используемых функций. На данный момент эта функциональность ещё не реализована.

Для разрешения отношений типов правила необходимо кодировать в логические уравнения, которые затем можно использовать в качестве теорем для вывода посредством SMT-решателя. Такой подход требует дополнения определения анализа описанием кодирования правил в логические выражения от типа $t$, эффектов $e$ и типового окружения Γ:

```
-- Here x stands for the type
-- e for the effect
-- s for the type environment (gamma)
-- Relation "x `tops` {y0,y1..yn}"
-- is same as "x ≥ y0 ∧ x ≥ y1 ∧ ... ∧ x ≥ yn"
-- Relation "s ⊩ {v0,v1..vn} : {t0,t1..tn}"
-- is same as "s ⊢ v0:t0 ∧ s ⊢ v1:t1 ∧ ... ∧ s ⊢ vn:tn"
-- Simple rule - constant value (eq. to `x ≡ encode(DDefined)`)
rules Constant = definition DDefined
rules Create = definition (DPointerTo DUndefined)
rules (Delete vals) =
    λ x e s → x `tops` (s ⊩ vals)
rules (Skip) = noRule
rules (Load vals) =
    λ x e s → pointerTo x `tops` (s ⊩ vals)
rules (Store ptrs val) =
    λ x e s → x ≡ pointerTo (s ⊢ val)
-- Returning and Undefined or MaybeDefined value
-- results in an effect 0
rules (Return (Just v)) =
    λ x e s → let t =  s ⊢ v in
        if_ (DUndefined `typeEq` t ∨ DMaybeDefined `typeEq` t)
            (e `effect` 0)
rules (JoinAnd vs) =
    λ x e s →
        let types = s ⊩ vs in
        if_ (bAny (typeEq DUndefined) types)
            (DUndefined `typeEq` x)
        elif_ (bAny (typeEq DMaybeDefined) types)
            (DMaybeDefined `typeEq` x)
        else_
            (DDefined `typeEq` x)
rules (JoinOr vals) =
    λ x e s → x `tops` (s ⊩ vals)
rules (Call _ _) = noRule -- to be implemented
```

Каждое из этих правил может быть использовано как утверждение в рамках задачи SMT и передано SMT-решателю для нахождения удовлетворяющего решения. Объединение правил для конкретной программы позволяет проверить их состоятельность путём нахождения любого решения, удовлетворяющего всем правилам. Если таковое решение может быть найдено, система непротиворечива для заданной программы. Дополнительные условия, накладываемые на объединённое утверждение, позволяют выбрать из всего спектра решений (решений системы типов с отношением подтипов, как правило, больше одного) то, которое соответствует истинному набору дефектов.

Поскольку каждый эффект выражает дефект в программе, основное из этих условий может быть сформировано следующим образом: следует найти распределение с наименьшим числом эффектов. К сожалению, такое условие в рамках зада-

чи SMT сформировать нельзя, но можно задать ограничение по числу эффектов в программе. Таким образом, следует найти наименьшее $n$ такое, что при числе эффектов в распределении, меньшем либо равном $n$, распределение корректно. Если такое распределение может быть найдено, оно содержит в себе только истинные с точки зрения заданных правил эффекты. Данный подход отличается от стандартного (стандартный подход к поиску решения заключается в том, чтобы для каждой неоднозначности в распределении найти наименьший с точки зрения отношения подтипов тип) тем, что минимизирует только те операторы, которые непосредственно влияют на число эффектов. Поскольку не все операторы программы могут приводить к дефектам или как-то влиять на их количество, это позволяет быстро получить результат. Данный подход был проверен на ряде примеров как с дефектами, так и без них и подтвердил свою работоспособность.

# 4. Заключение

Целью работы является описание разрабатываемого средства статического анализа программ. Основной идеей данного средства является предоставление возможностей для экспериментов по применению методов на основе систем типов и эффектов к реальным программам на сложных языках, таких, как язык C. Несмотря на то, что описываемое средство находится на стадии реализации, получены следующие результаты:

- библиотека разбора биткода LLVM может быть использована отдельно от средства для разбора программ, написанных на C;
- система типов и эффектов для поиска возврата неинициализированного значения из функции (расширенная версия системы, приведённой в этой статье) реализована с помощью описываемого средства и протестирована на реальных программах.

Представленный пример анализа демонстрирует простоту расширения и модификации предложенного подхода. Реализация поиска возврата неинициализированных значений из функций достаточно проста, и можно сказать, что реализация других видов анализа в рамках представленного средства так же не представляет сложности. Таким образом, предложенный абстрактный подход может использоваться как основа для множества реализаций анализа, в том числе после незначительных модификаций, не только для поиска дефектов.

Перспективы расширения средства включают в себя:

- межпроцедурный анализ на основе идей, описанных в разделе 3., в том числе аннотирование поведения основных функций из стандартной библиотеки языка C;
- реализация и тестирование более сложных видов анализа на основе систем типов и эффектов.

Необходимо провести дополнительное тестирование работоспособности и надёжности представленного средства с целью доказательства практической применимо-

сти и возможного уточнения идей, представленных в разделе 3. Дальнейшее развитие системы включает в себя поддержку возможностей комбинирования различных видов анализа, а также включение других видов анализа, в том числе для комбинирования систем типов и эффектов с анализом на основе потока данных.

## Список литературы

1. Coverity Scan Open Source Report, 2009. `http://scan.coverity.com/report/`

2. Ertl M.A., Wien TU: Domination-Based Scoping and Static Single Assignment Languages. Static Single-Assignment Form Seminar, 2009.

3. Horwitz S. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. ACM Transactions on Programming Languages and Systems, 1–6. ACM Association for Computing Machinery, 1997.

4. Itsykson V., Moiseev M., Tsesko V., Zakharov A. Automatic defects detection in industrial C/C++ software. Software Engineering Conference in Russia (CEE-SECR), 2009.

5. Jim T., Palsberg J. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

6. Lattner C., Haberman J., Housel P. S. LLVM Bitcode File Format. `http://llvm.org/docs/BitCodeFormat.html`

7. Lattner C. LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`

8. llvm-parser – A haskell library for parsing LLVM binary bitcode files. `http://code.google.com/p/llvm-parser`

9. Marino D., Millstein T. A Generic Type-and-Effect System. TLDI, 2010.

10. Nielson F., Nielson H. R. Type and Effect Systems. Correct System Design, 1999. P. 114–136.

11. Nielson F., Nielson H. R., Hankin C. Principles of Program Analysis. Springer-Verlag, 2005.

12. Ross K.P.D. From System F to Typed Assembly Language. Twenty-Fifth ACM SIGPLAN Symposium on Principles of Programming Languages. 1998. P. 85–97.

13. Zadeck K. The Development of Static Single Assignment Form. Static Single-Assignment Form Seminar, 2009.

# LLVM-based Static Analysis Tool Using Type and Effect Systems

Belyaev M. A., Tsesko V. A.

**Keywords:** static program analysis, type and effect system, low level virtual machine, defect detection, SSA

The intention of this paper is to describe a static analysis tool under development. The principal idea behind the design of this tool is to use type and effect systems for static analysis of real programs. The tool uses LLVM bitcode files as input, thus extending the set of analyzed languages to those supported by LLVM compiler infrastructure. It uses its own parser of bitcode files and a program model similar to that of LLVM. The approach taken is to research feasibility of designing instruments for static analysis by applying known type and effect system based algorithms for detecting defects to LLVM bitcode language and effectively to the original source code.

**Сведения об авторах:**

Беляев Михаил Анатольевич,
Санкт-Петербургский государственный политехнический университет,
аспирант кафедры компьютерных систем и программных технологий;
Цесько Вадим Александрович,
Санкт-Петербургский государственный политехнический университет,
ассистент кафедры компьютерных систем и программных технологий