A Framework for Building Verified Partial Evaluators

Построение сертифицированных частичных вычислителей

Екатерина Вербицкая

Лаборатория языковых инструментов JetBrains

20.04.2020

Частичные вычисления (partial evaluation)

$$p(s, d)$$

$$peval(p, s) = p'$$

$$\forall d : [\![p']\!](d) \equiv [\![p]\!](s, d)$$

Пример: двоичное возведение в степень

```
pow (n, a) = if n == 0
             then 1
             else if n `mod` 2 == 1
                  then pow (n - 1, a) * a
                  else let r = pow (n `div` 2, a)
                       in  r * r

peval (pow, 10) = \ a -> ((a ^ 2) ^ 2 * a) ^ 2
```

Пример: префикс-суммы

$$sum(lst, l) = \sum_{i=0}^{l} (i * lst[i])$$

$$prefixSum(lst) = reverse\ lst',\ \text{где}\ lst'[i] \equiv sum(lst, i)$$

Хочется, чтобы в результате специализации получалось следующее:

```
prefixSum ([a,b,c,d]) =
  let acc = b + c * 2 in
  let acc' = acc + d * 3 in
  [acc', acc, b, 0]
```

Зачем нужны частичные вычисления?

Для улучшения производительности реализаций

- Можно писать обобщенные библиотеки, потом специализировать под разные условия
- Улучшать в соответствии с правилами, верными в конкретном домене
- Не надо создавать свой компилятор, который знает про эти правила

Цели и контекст этой работы

- Улучшать существующие библиотеки, не адаптируя их
- Использование в сертифицированных системах
  - Библиотеки реализованы на функциональном подмножестве Coq
  - Частичные вычисления должны производиться *быстро* и генерировать *пруф-термы*, гарантирующие их корректность
  - Количество доверенных компонент не должно увеличиться (нельзя добавлять новые стратегии редукций в ядро Coq)

Вклад этой работы

- Фреймворк для создания сертифицированных частичных вычислителей
  - Без изменений в Coq
  - Высокая скорость частичных вычислений
  - Использование для частичных вычисления definitional equality и пользовательских теорем
  - Переиспользование общих подтермов
  - Извлечение частичных вычислителей
  - Поддержка правил переписывания с дополнительными условиями
- Эвалюация на Coq библиотеке для криптографии Fiat Cryptography

Поддерживаются:

- Функции высших порядков
- Индуктивные типы данных
- Арифметические законы

Пример про префикс-суммы

код

Про доверенный код

- Важно не добавлять в ядро кода, которому доверяем
- Новые стратегии редукций могут быть полезны для производительности
- Невозможно обосновывать каждый маленький шажок преобразований
- Coq не требует обосновывать то, что следует из definitional equality

Решение: максимально использовать встроенный механизм редукций Coq

Подходы к переписыванию термов: definitional equality

Основной механизм переписывания в Coq

- β-редукция
- Замена идентификаторов на их значения
- Сопоставление с шаблоном при известном значении сопоставляемого
- ...

Свободные переменные быстро водят в тупик

Подходы к переписыванию термов: autorewrite

- Используется база квантифицированных равенств
- Каждое равенство порождает фрагменты пруф-термов

- В итоге получается много больших пруфтермов
  - Для обоснования $C[e_1] = C[e_2]$, при условии $e_1 = e_2$ необходимо скопировать весь контекст C

Подходы к переписыванию термов: Aehlig et al.

Комбинирование вычислений над λ-термами и использования обоснованных равенств

- На языке пруф-асистанта реализуется deep-embedded ML и его операционная семантика
- Для конкретного терма и правила переписывания порождается deep-embedded терм на ML, который будет порождать упрощенный терм (если завершается)

- Много нового кода, которому нельзя доверять
- Нет гарантии, что семантика реализованного ML совпадает с тем, на котором будет исполняться частичные вычисления

Подходы к переписыванию термов: эта работа

- Реализуем правила переписывания непосредственно на Coq, не на встроенном языке
- Используем ядро Coq для непосредственного осуществления редукций

Создадание частичного вычислителя по шагам

- Пользователь пишет леммы, которые будут использоваться для переписывания
- Порождаются индуктивные типы для всех примитивных типов и функций
- Порождаются вспомогательные определения и доказываются леммы, необходимые для работы с индуктивными определениями
- Реифицируются утверждения о правилах переписывания. Пользователь доказывает их полноту (soundness) и синтаксическую корректность (syntactic well-formedness)
- Все нужные утверждения передаются специальной тактике

Применение частичного вычислителя

- Цель превращается в логическую формулу
  - Если в ней есть свободные переменные, то она превращается в функцию на этих переменных
- Реифицируем часть, которую хотим упрощать
  - Потом она заменяется на денотацию реифицированной версии
- Используем теорему о том, что правило переписывания сохраняет денотацию термов, чтобы обосновать переписывание
  - Для применения правила переписывания применяется vm-compute
- Используем cbv для упрощения вызовов к денотации функции

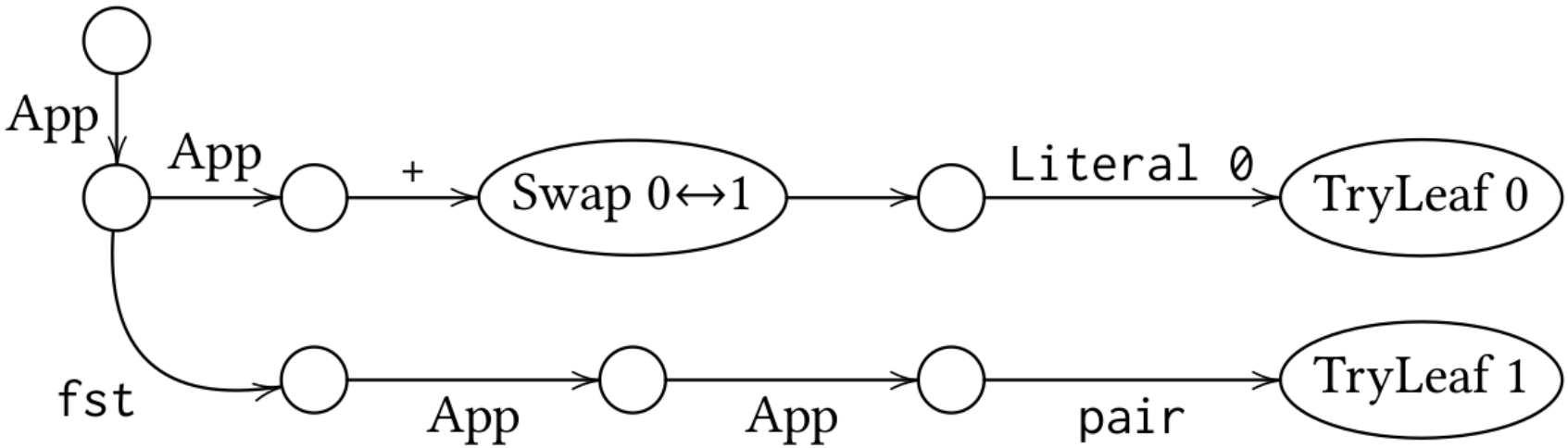
Выбор правила переписывания из набора

$$?n + 0 \to n$$

```
App (App (Ident +) Wildcard) (Ident 0)
```

$$fst_{\mathbb{Z},\mathbb{Z}}(?x, ?y) \to x$$

```
App (Ident fst) (App (App (Ident pair) Wildcard) Wildcard)
```



Normalization by Evaluation

$$t ::= t \to t \mid b \qquad\qquad e ::= \lambda v.e \mid e\ e \mid v \mid c$$

$$NbE_t(t_1 \to t_2) = NbE_t(t_1) \to NbE_t(t_2)$$
$$NbE_t(b) = expr(b)$$

$$reify_t : NbE_t(t) \to expr(t)$$
$$reify_{t_1 \to t_2}(f) = \lambda v.reify_{t_2}(f(reflect_{t_1}(v)))$$
$$reify_b(f) = f$$

$$reflect_t : expr(t) \to NbE_t(t)$$
$$reflect_{t_1 \to t_2} = \lambda x.reflect_{t_2}(e(reify_{t_1}(x)))$$
$$reflect_b(e) = e$$

Normalization by Evaluation

$$reduce : expr(t) \to NbE_t(t)$$
$$reduce(\lambda v.e) = \lambda x.reduce([x/v]e)$$
$$reduce(e_1\ e_2) = (reduce(e_1))(reduce(e_2))$$
$$reduce(x) = x$$
$$reduce(c) = reflect(c)$$

$$NbE : expr(t) \to expr(t)$$
$$NbE(e) = reify(reduce(e))$$

Применение правила переписывания

$$reflect_t : expr(t) \to NbE_t(t)$$
$$reflect_{t_1 \to t_2} = \lambda x.reflect_{t_2}(e(reify_{t_1}(x)))$$
$$reflect_b(e) = \not{e}\ rewriteHead(e)$$

Реализация: переменные

Самое сложное — следить за переменными

Варианты представления термов с переменными:

- Представление переменных строками
- Индексы де Брауна
- Higher-order abstract syntax
- *Parametric higher-order abstract syntax*

Higher-order abstract syntax

$$
\begin{aligned}
term &: Type \\
App &: term \to term \to term \\
Abs &: (term \to term) \to term
\end{aligned}
$$

$$
\begin{aligned}
id &= Abs(\lambda x.x) \\
diverge &= App(Abs(\lambda x.App\ x\ x))(Abs(\lambda x.App\ x\ x))
\end{aligned}
$$

Parametric higher-order abstract syntax

$$
\begin{aligned}
term(V) &: Type \\
Var &: V \to term(V) \\
App &: term(V) \to term(V) \to term(V) \\
Abs &: (V \to term(V)) \to term(V)
\end{aligned}
$$

$$id = Abs(\lambda x.Var\ x)$$

$$Term = \forall V : Type.term(V)$$

PHOAS: подсчет переменных

$$numVars : term(unit) \to (N)$$
$$numVars(Var\ \_) = 1$$
$$numVars(App\ e_1\ e_2) = numVars(e_1) + numVars(e_2)$$
$$numVars(Abs\ e) = numVars(e())$$

$$NumVars : Term \to (N)$$
$$NumVars(E) = numVars(E\ unit)$$

PHOAS: допустимость η-редукции $(\lambda x.M\ x \to M)$

$$
\begin{aligned}
canEta' &: term(bool) \to bool \\
canEta'(Var\ b) &= b \\
canEta'(App\ e_1\ e_2) &= canEta'(e_1)\ \&\&\ canEta'(e_2) \\
canEta'(Abs\ e) &= canEta'(e\ true)
\end{aligned}
$$

$$
\begin{aligned}
canEta &: term(bool) \to bool \\
canEta(Abse) &= match\ e\ false\ with \\
&\quad \mid App\ e_1(Var\ false) \Rightarrow canEta'(e_1) \\
&\quad \mid \_ \Rightarrow false \\
CanEta &: Term \to bool \\
CanEta(E) &= canEta(E\ bool)
\end{aligned}
$$

PHOAS: capture-avoiding substitution

$$subst : \forall V : Type.term(term\ V) \to term(V)$$
$$subst(Var\ e) = e$$
$$subst(App\ e_1\ e_2) = App(subst(e_1))(subst(e_2))$$
$$subst(Abs\ e) = Abs(\lambda x.subst(e(Var\ x)))$$

$$Term1 = \forall V : Type.\ V \to term(V)$$
$$Subst : Term1 \to Term \to Term$$
$$Subst\ E_1\ E_2 = \forall V : Type.\ subst(E_1\ (term(V)\ (E_2\ V))$$

PHOAS: представление целевого языка

```
Inductive type := arrow (s d : type)
| base (b : base_type).
Infix "->" := arrow.
Inductive expr (var : type -> Type) : type -> Type :=
| Var {t} (v : var t) : expr var t
| Abs {s d} (f : var s -> expr var d) : expr var (s -> d)
| App {s d} (f : expr var (s -> d)) (x : expr var s) :
expr var d
| Const {t} (c : const t) : expr var t
Definition Expr (t : type) : Type :=
  forall var, expr var t.
```

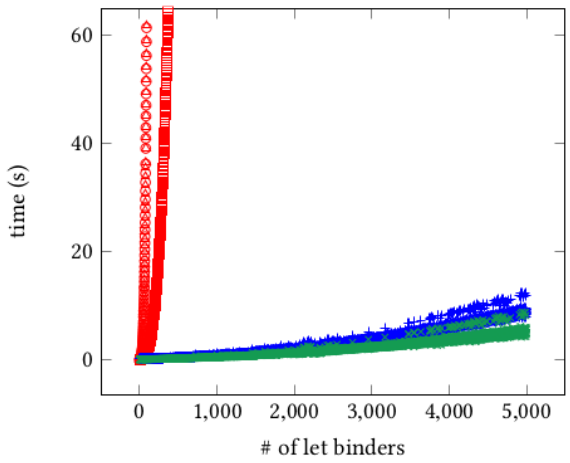
Дополнительные условия

$$n_1 + m - n_2 \to m, n_1 = n_2$$

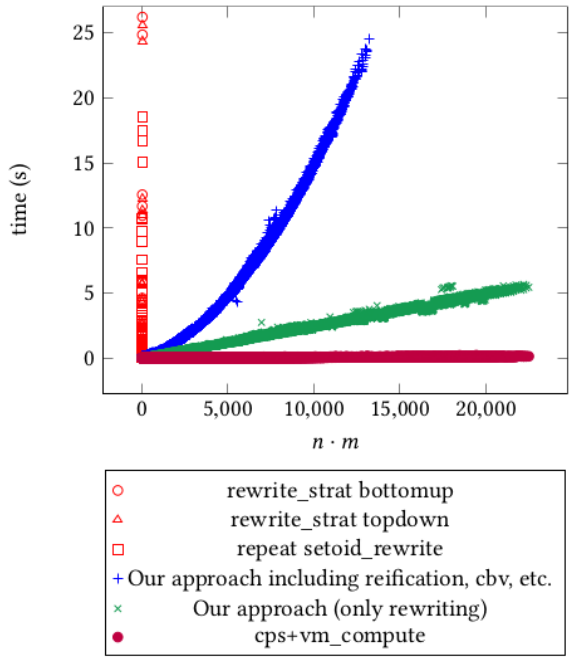
- Условия — вычислимые булевы функции на переменных, которые являются константами времени компиляции
- Реификация ожидает найти реализацию предиката среди гипотез
- Если в них используются переменные-не-константы, то применять правило переписывания нельзя

- Некоторые ограничения можно получать из абстрактной интерпретации

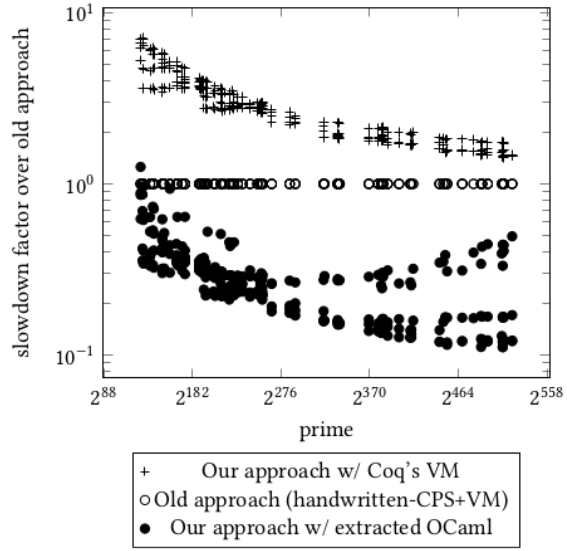
Эвалюация



(a) Nested binders



(b) Binders and recursive functions



(c) Fiat Cryptography

Figure 3. Timing of different partial-evaluation implementations

Источники

- Jason Gross. A Framework for Building Verified Partial Evaluators
  - Библиотека
- Klaus Aehlig et al. A compiled implementation of normalization by evaluation
- Adam Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics