サーベイ： Combinators for Program Generation [7]

すみい*

1999/4/22

要旨

Peter Thiemann の論文 [7] を紹介する。この論文では、 compositional に定義された (offline) partial evaluator から、同じ特化をより効率的に行う “combinator” の集合を導出している。結果として得られる combinator たちは、 Scheme, ML, Haskell といった大抵の関数型言語で実装できる。同様の手法は partial evaluation だけでなく、一般に compositional に定義された domain specific language の実装に利用できる。この論文は “Cogen in Six Lines” [8] の拡張・改訂版に当たる。

1 準備

部分評価 (partial evaluation, PE) とは、プログラム p とその入力の一部 s を受け取り、任意の残りの入力 d について $p(s,d) = p_s(d)$ となるような、特化したプログラム p_s を生成する処理である。

たとえば、整数を含む λ-calculus において、 $p = \lambda s.\ \lambda d.\ (s+1)+d$, $s = 2$ とすると、通常は $p_s = \lambda d.\ 3+d$ となる。 s を static な入力といい、 d を dynamic な入力という。

s の実際の値が与えられる前に、プログラムのどの部分を PE の際に実行し、どの部分を p_s にコードとして残すかという解析 (binding-time analysis, BTA) を行っておくやり方を、 offline PE という。 PE の際に実行できる部分は static であるといい、コードとして残す部分は dynamic であるという。一般に、 BTA の結果は、プログラムの static な部分に upperline を、 dynamic な部分に underline を annotate して表す。ただし、前者は簡単のために省略することが多い。

たとえば、先の例では、 $s+1$ における $+$ は PE の際に実行でき、 $3+d$ における $+$ はコードとして残す必要がある。また、 λ は「closure を作る」という計算を表すと考えると、 $\lambda s.\ \ldots$ における λ は PE の際に実行でき、 $\lambda d.\ \ldots$ における λ はコードとして残す必要がある。したがって、 p に対する annotation は $\lambda s.\ \underline{\lambda} d.\ (s+1) \mathbin{\underline{+}} d$ となる。

なお、関数適用は、 $e_1\ e_2$ のように単に二つの式を並べて表すと、 static なものと dynamic なものを区別できないので、ここでは @ を用いて $e_1 \mathbin{@} e_2$ や $e_1 \mathbin{\underline{@}} e_2$ のように表す。

BTA は「dynamic な式に依存する式は、また dynamic でなければならない」というルールを守りながら、できるだけ多くの式を static にするように行う。ただし、実際には PE を停止させるために、 static にできる部分を dynamic にしなければならないこともある。

これから紹介する論文では、すでに BTA により annotation が与えられたプログラムに対する offline PE を扱う。対象言語としては pure な untyped λ-calculus を、実装言語としては OCaml を用いる。

2 古典的な部分評価器

はじめに、 λ-mix [2] のような古典的な部分評価器について見ておく。 λ-mix は BTA と、 annotation のついた λ 式に対するインタプリタとからなる。

* sumii@yl.is.s.u-tokyo.ac.jp

後者を OCaml で書くと、 Figure 2 (と Figure 1) のようになる。exp は annotation のついた λ 式を、value はそれを評価して得られる値を表すデータ型である。値としては、通常の関数 (Fun) の他に、 PE の際には実行されずに残るコード (Code) がある。部分評価器の本体 (spec) は、式と環境 (変数名から値への有限写像) を受け取り、それを評価した値を返す。

たとえば、 dynamic な関数 LamD (x, e) は、

1. 新しい変数名 xx を生成する
2. 変数 x にコード Code (Var xx) を bind した環境で、式 e を評価する
3. (annotation が誤っていなければ) コード Code body が得られるはずなので、コード Code (Lam (xx, body)) を返す

という手順で評価する。

例 **1** $(\lambda x.\ x) @ (\underline{\lambda} x.\ x) \implies \underline{\lambda} x.\ x$

```
# spec (App (Lam ("x", (Var "x")), LamD ("x", (Var "x")))) [] ;;
- : value = Code (Lam ("x1", Var "x1"))
```

例 **2** $\underline{\lambda} x.\ \underline{\lambda} y.\ (\lambda x.\ x) @ y \implies \underline{\lambda} x.\ \underline{\lambda} y.\ y$

```
# spec (LamD ("x", LamD ("y", App (Lam ("x", (Var "x")), (Var "y"))))) [] ;;
- : value = Code (Lam ("x5", Lam ("x6", Var "x6")))
```

3 問題点

上のような部分評価器は本質的にインタプリタであるので、通常の評価と比較して以下のようなオーバーヘッドが存在する。

- match e with ... のような syntax dispatch
- lookup と upd のような environment manipulation
- Fun や Code のような tagging (実装言語が静的に型付けられており、 dependent type をサポートしていない場合)

第一と第二のオーバーヘッドは、対象言語と実装言語が等しく、しかも untyped ならば、プログラムを static な入力として部分評価器を部分評価すること (self-application) により、ある程度は解決できる —— $\mathcal{PE}(p,s) = p_s$ と $\mathcal{PE}$ を定めると、 $\mathcal{PE}(\mathcal{PE},p) = \mathcal{PE}_p$ に対して、 $\mathcal{PE}_p(s) = \mathcal{PE}(p,s) = p_s$ となるが、 $\mathcal{PE}_p(s)$ は $\mathcal{PE}(p,s)$ より少ないオーバーヘッドで p_s を求めることができる (はずである)。さらに、 $\mathcal{PE}(\mathcal{PE},\mathcal{PE}) = \mathcal{PE}_{\mathcal{PE}}$ に対して、 $\mathcal{PE}_{\mathcal{PE}}(p) = \mathcal{PE}(\mathcal{PE},p) = \mathcal{PE}_p$ であるが、 $\mathcal{PE}_{\mathcal{PE}}(p)$ は $\mathcal{PE}(\mathcal{PE},p)$ より少ないオーバーヘッドで $\mathcal{PE}_p$ を求めることができる (はずである)。

しかし、たとえば Scheme (のフルセット) のような現実の言語に対して、 self-application がうまくいくような部分評価器を書くのは容易ではない。また、第三のオーバーヘッドは self-application では除去できない。そこで、最初から $\mathcal{PE}$ のかわりに $\mathcal{PE}_{\mathcal{PE}}$ に相当するもの (歴史的な経緯から cogen と呼ばれる) を与えることにより部分評価器を実現する方法が考えられる。そのような方法を cogen-by-hand approach という。 cogen-by-hand approach は、 ML のような (dependent type のない) 静的な型システムを持つ言語に対する部分評価器を実現するために最初に利用された [3]。

従来の cogen-by-hand approach にもとづく部分評価器では、その名の通りに cogen は人間の手によって ad hoc に実装されていた。この論文では、同様に効率的な部分評価器を、 Figure 2 のような compositional に記述された $\mathcal{PE}$ から、正当性が証明されたいくつかの変換によって systematic に導出する。

4 Combinators

まず、第一のオーバーヘッドを取り除く。 Figure 2 の部分評価器は、まずexp 型の式を構築してから、それを pattern matching により分解しながら部分評価していた。そこで、exp 型の式の constructor と、それに対する部分評価器を合成したものを最初から定義すると、 Figure 3 のようになる。使い方としては、spec e r のかわりに、e' r を実行することになる。ただし、e' はe の中の Var や Lam などを varや lam などでおきかえたものである。

例 **1** $(\lambda x.\ x) @ (\underline{\lambda} x.\ x) \implies \underline{\lambda} x.\ x$

```
# app (lam ("x", (var "x")), lamD ("x", (var "x"))) [] ;;
- : value = Code (Lam ("x8", Var "x8"))
```

例 **2** $\underline{\lambda} x.\ \underline{\lambda} y.\ (\lambda x.\ x) @ y \implies \underline{\lambda} x.\ \underline{\lambda} y.\ y$

```
# lamD ("x", lamD ("y", app (lam ("x", (var "x")), (var "y")))) [] ;;
- : value = Code (Lam ("x9", Lam ("x10", Var "x10")))
```

この変換が正しいことを示すには、任意の e : exp とr : (ident, value) env に対して spec e r とe' r を実行した結果が等しいことを示せばよい。どちらにもgensym という副作用のある関数が含まれているので、証明には side effect を考慮した λ-calculus の equational theory である computational λ-calculus [4] を用いている。

5 Higher-Order Abstract Syntax

次に、第二のオーバーヘッドを取り除くために、高階抽象構文 (higher-order abstract syntax, HOAS) [5] を用いる。 HOAS というのは、対象言語の binding を実装言語の binding で代用することにより、環境を明示的に操作する手間を省く技法である。たとえば、 $\lambda x.\ \underline{\lambda} y.\ x @ y$ という式は、

```
Lam ("x", LamD ("y", App (Var "x", Var "y")))
```

のかわりに

```
HLam (fun x -> HLamD (fun y -> HApp (HVar x, HVar y)))
```

のように表す。ただし、HVar や HLam などは、

```
type hexp =
  HVar of ident
| HLam of (ident -> hexp)
| HApp of hexp * hexp
| HLamD of (ident -> hexp)
| HAppD of hexp * hexp
```

のように定義された hexp の constructor である。 HOAS で表した式と通常の式とは、変数名の違いを除いて、容易に相互に変換できる (Figure 5)。

Figure 3 において、 source program に HOAS を用いると、 Figure 4 のようになる。

例 **1** $(\lambda x.\ x)\ @\ (\underline{\lambda} x.\ x) \implies \underline{\lambda} x.\ x$

```
# happ (hlam (fun x -> hvar x), hlamD (fun x -> hvar x)) ;;
- : value = Code (Lam ("x1", Var "x1"))
```

例 **2** $\underline{\lambda} x.\ \underline{\lambda} y.\ (\lambda x.\ x)\ @\ y \implies \underline{\lambda} x.\ \underline{\lambda} y.\ y$

```
# hlamD (fun x -> hlamD (fun y -> happ (hlam (fun x -> hvar x), hvar y))) ;;
- : value = Code (Lam ("x2", Lam ("x3", Var "x3")))
```

これが正しく働くことを示すは、実装言語 (OCaml) の semantics を考慮しなければならないが、ここでは環境に関する部分だけを考えれば十分なので、それを明示化する変換を Figure 6 のように定義する。その上で、任意の`e : exp` と `r : (ident, value)` に対して、`e' r` と $[\![$`e''`$]\!]\ \rho$ を実行した結果が等しいことを示せばよい。ただし、`e''` は `e` を `Lam`や `Var` を用いた式から `hlam` や `hvar` を用いた式に変換したもので、ρ は `r` を `ident` $\rightarrow$ `mlexpr` 型の関数に変換したものである。証明は `e` の構造に関する帰納法による。

さらに、`gensym` で変数名を生成する手間を省くために、結果として生成されるコードのほうにも HOAS を利用する。具体的には、部分評価器の出力として、`value` のかわりに

```
type hvalue =
  HFun of (hvalue -> hvalue)
| HCode of hexp
```

と定義した `hvalue` を用いる。ただし、結果を表示する際には、`HCode` を通常の `Code` に変換する必要がある。`hvalue` を用いた combinator たちは、 Figure 8 のようになる。 Figure 4 と本質的に異なるのは、`hhlamD` の部分である。

この変換が正しいことを示すには、`value` と `hvalue` の間の変換を Figure 7 のように定義し、任意の `e : exp` に対して`e'` と `hvalue2value e'' (fun x -> x)` を実行した結果が等しいことを示せばよい。

6 Untagging

最後に、第三のオーバーヘッドを除去することを考える。 Figure 8 の combinator たちをよく見ると、`hvalue` についているタグ (`HFun` と `HCode`) は、実はいらないことがわかる —– `hhapp` では `f` は `HFun` であることが、`hhlamD` や `hhappD` では `x_e (HCode (HVar xx))` や `f, a` は `HCode` であることが最初から仮定されている。 λ-mix では `spec` の結果が関数かコードかは引数に依存していたので、 dependent type のない静的型システムのもとでは、タグが必要だった。しかし、 combinator にもとづく部分評価器では、それらは不要である[1]。 Figure 8 からそれらを省くと、 Figure 9 のようになる。なお、`huvar, hulam, huapp` は、実装言語 (OCaml) の変数・λ抽象・関数適用で代用することもできる。

この変換が正しいことは、以下の手順で示す。

まず、 BTA による annotation が正しければ、 Figure 8 の combinator は well-typed であることを示す。 annotation の正しさは、 Figure 10 のような型付け規則を持った、一種の型システムとして定式化できる [2]。ただし、型の syntax は $t ::= D \mid t \rightarrow t$ と定める。ここで、`Var` や `Lam` を `huvar` や`hulam` に書き換え、 D を `hexp` と読みかえれば、 Figure 10 の型付け規則は ML の型付け規則に包摂されている。したがって、任意の`e` について、`e` が Figure 10 の型付け規則のもとで well-typed であれば、`e'` は、 ML の型付け規則のもとで well-typed である (*)。ただし、`e'` は、`e` においてVar を `Lam` から `huvar` や `hulam` に書き換えた式とする。

次に、任意の `e` に対して、 Figure 8 (タグあり) と Figure 9 (タグなし) の combinator たちによる部分評価の結果が等しいことを示す。ただし、 Figure 8 と Figure 9 では結果の型が異なるので、型に index された次のような equivalence を定義し、`e` の構造に関する帰納法を用いる。

[1] 一般に、あるデータ型の引数に対して compositional に定義された dependent function は、そのデータ型の constructor と合成することにより、 ML のような言語でも書くことができる [9]。

- `HCode` $k \sim_D h$ iff $k = h$ in the computational λ-calculus
- `HFun` $f \sim_{t_2 \to t_1} g$ iff $x \sim_{t_2} y$ implies $f(x) \sim_{t_1} g(y)$ for all x, y

なお、 (*) の逆は一般に成り立たない。

$$let\ id = \lambda x.\ x\ in\ (id\ @\ (\lambda x.\ x))\ @\ (id\ @\ (\underline{\lambda} x.\ x)) \implies \underline{\lambda} x.\ x$$

```
# let id = fun x -> x in
  (id (fun x -> x)) (id (hulamD (fun x -> x)));;
- : exp = Lam ("x1", Var "x1")
```

これは、 Figure 10 の型システムが monomorphic であるのに対し、 ML は let-polymorphism をサポートしているためである。逆に、 ML における well-typedness をもって annotation の正しさの基準とすれば、 Figure 10 の型システムに let-polymorphism を導入したことになる。

7 拡張と実験

前節まででは、 λ-mix のような単純な部分評価器から出発して、単純な combinator たちを導出したが、より高度な部分評価器から出発すれば、より高度な combinator たちを導出できる。論文では、対象言語に let 式・if 式・定数・再帰などを導入し、それらをうまく扱う combinator たちを導出している (時間が足りないと思うので、詳細は省略する) 。

論文の筆者 (Thiemann) は、以上の方式で実現した Scheme の部分評価器 (PGG) と、古典的な部分評価器 (Similix) の性能を、 Figure 16 のプログラムについて実験により比較した。処理系には Scheme48 を用いている。結果は Figure 17 のようになった。各列の意味は以下の通りである。

gen Similix において、 $\mathcal{PE}_{\mathcal{PE}}(p)$ により $\mathcal{PE}_p$ を得るのにかかった時間

comp Similix において、 $\mathcal{PE}_p(s)$ により p_s を得るのにかかった時間

spec Similix において、 $\mathcal{PE}(p, s)$ により p_s を得るのにかかった時間

CPS let 式を扱うために CPS で書かれた PGG において、 p_s を得るのにかかった時間

DS let 式を扱うために Filinski による shift/reset の実装 [1] を用いた PGG において、 p_s を得るのにかかった時間

この結果によれば、全体的に PGG のほうが Similix よりも効率が良い。 Similix では、 self-application により効率がかえって悪化している場合さえある。 PGG では、 CPS と DS の効率は comparable である。各々のベンチマーク・プログラムに対する詳しい考察はなく、結果がばらついている原因は不明である。

8 余談

11 月のミーティングで、 compositional に書かれた一般のインタプリタについて、この論文と同様の変換によりオーバーヘッドの少ない実装を導出するという話を発表したのですが、浅井さんに “Cogen in Six Lines” [8] を紹介され、すでに同じことがやられているのを知って、ちょっとショックだったという過去があります。また、 3 月の PPL で発表した話 [6] も、本質的にはこの論文の結果を online な PE に適用しただけになっていて、もう PE に関しては自分のネタがなくなってしまい、ちょっと困っているこの頃です。

参考文献

[1] Andrzej Filinski. Representing monads. In *Proceedings of the 21st Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, 1994.

[2] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[3] John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings of Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218. Glasgow University, 1991.

[4] Eugenio Moggi. Computational lambda-calculus and monads. Technical Report LFCS-88-86, University of Edinburgh, 1988.

[5] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[6] Eijiro Sumii. Online type-directed partial evaluation for dynamically-typed languages. In *Draft Proceedings of First JSSST Workshop on Programming and Programming Languages*, pages 55–66, 1999.

[7] Peter Thiemann. Combinators for program generation. *Journal of functional programming*. To appear.

[8] Peter J. Thiemann. Cogen in six lines. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, 1996.

[9] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 289–300, 1998.