プログラミング－言語・基礎・実践－　1－2
（1991. 4. 26）

関数型言語の一般部分計算法

高野明彦

（株）日立製作所 基礎研究所

概要

一般部分計算法は、部分計算と定理証明の技法に基づくプログラム変換法である。従来の部分計算が既知データの値だけを利用してプログラムの特殊化を行なっていたのに対し、一般部分計算は定理証明系を使うことにより、プログラムの論理構造、抽象データ型や基本関数の代数的性質などの情報を有効に活用できる。本論文では、一階の関数型言語を対象言語とする一般部分計算法を定式化する。まず、評価順序を問わず適用可能な変換法を定める。次に、対象言語を遅延評価意味論を持つものに限定し、対象プログラムの文脈情報を利用する変換法へ拡張する。射影により表現された文脈情報を用いて冗長な変換を除去している。

Generalized Partial Computation for a Lazy Functional Language

Akihiko Takano

Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama 350-03, JAPAN
takano@harl.hitachi.co.jp

Abstract

Generalized Partial Computation (GPC) is a program optimization principle based on partial computation and theorem proving. Techniques in conventional partial computation make use of only static values of given data to specialize programs. GPC employs a theorem prover to explicitly utilize more information such as logical structure of programs, axioms for abstract data types, algebraic properties of primitive functions, etc. In this paper we formalize a GPC transformation method for a lazy first-order language, and extend it to use context information of a program to be partially evaluated. Context information is represented as a projection and is utilized to eliminate redundant transformation.

1 Introduction

The recent progress in the field of partial evaluation is striking enough to make us believe that it will continue to be a promising and powerful technique over the next decade ([1,6,7]). But these advances also reveal the essential limit of its transformational power. Techniques in conventional partial evaluation use only static values of given data to specialize programs.

Generalized Partial Computation (GPC henceforth) was proposed to overcome these drawbacks. The key idea of GPC is to employ a theorem prover to explicitly utilize more information such as logical structure of programs, axioms for abstract data types, algebraic properties of primitive functions, etc. GPC was first proposed in [4], and its transformational power was demonstrated through many examples in [5]. But the concrete transformation methods given there expected a restricted form of language (e.g. u-form) with strict semantics.

In the first part of this paper we formalize a GPC transformation method which is applicable to both strict and lazy first-order functional languages. In the rest of the paper we concentrate on a lazy language and extend the method to one which utilizes context information of the program to be partially evaluated. This extension is based on *backward strictness analysis* using projections [3,8].

Projections are used to describe the context in which an expression is evaluated. Context information represents how much definedness is sufficient for an expression in that context. Our GPC transformation method for a lazy language eliminates redundant transformation using this form of context information.

This paper is organized as follows. Section 2 describes a first-order functional language. In section 3 we formalize a GPC transformation method for the language. Section 4 introduces a description of context that will be used in the analysis of lazy languages. In section 5 we extend the GPC transformation method for application to a lazy first-order language by utilizing context information that describes the sufficient definedness of a value.

2 Language

We consider a set of mutually recursive function definitions

$$\{\ f_i\ v_1 \ldots v_{n_i} = e_i \mid 1 \leq i \leq n\ \}$$

and an expression to be evaluated in the context of these definitions. Expressions have the syntax given by the following grammar:

e	::=	v	variable
	|	$c\ e_1 \ldots e_n$	constructor application
	|	$p\ e_1 \ldots e_n$	primitive function application
	|	$f\ e_1 \ldots e_n$	function application
	|	case e_0 of $cp_1 : e_1 \mid \ldots \mid cp_m : e_m$	case term
cp	::=	$c\ v_1 \ldots v_n$	case pattern

In applications, $e_1 \ldots e_n$ are called the *arguments*, and in a case term, e_0 is called the *selector*, and $cp_1 : e_1$, $\ldots$, $cp_m : e_m$ are called the *branches*. The case patterns may not be nested. Methods to transform case terms with nested patterns to ones without nested patterns are well known. Each constructor c, primitive function p and user-defined function f has a fixed arity n. An example definition is shown in Fig.1.

```
boolean ::= T | F
h       :  integer → integer
h n     =  case n ≤ 1 of
             T  :  1
             F  :  case odd(n) of
                     T  :  h (n − 1) + h (n − 2)
                     F  :  h (n − 1)
```

Figure 1: Example definition

3 GPC method $\mathcal{G}_0$

In this section we define a GPC method which transforms a program in the first-order functional language above. The intended operational semantics of the language is either strict order (leftmost innermost first) or normal order (leftmost outermost first) graph reduction.

3.1 GPC Principle

Given a syntax tree representing an expression and the information regarding that expression, GPC transformation will involve specializing the expression using that information, and propagating the information toward the leaves of the tree to yield information about the subexpressions to be specialized. The information at the root describes the assumable constraints on the values of the variables, with the result of the propagation describing the constraints on the values of the variables in the lower structure of the expression, which are deducible from the condition at the root. The information at each node is expressed in some logical formula which is used by a theorem prover to specialize the corresponding expression. The specializing process yields a specialized expression (residual expression) equivalent to the original expression on the assumption of the attached information. GPC employs a theorem proving

technique to guarantee this equivalence that utilizes additional information such as semantics of the language, axioms for abstract data types, algebraic properties of primitive functions, etc., which are rarely used in conventional partial evaluation.

In this paper the information at each node is expressed as a set of identities, each of which we write here in the form $l \leftrightarrow r$, where l and r are expressions. The equational reasoning is used for theorem proving. There is a special set E_0 of identities which are all valid throughout the program. E_0 is called the *underlying logic* for the language, and is used implicitly in the specializing process together with the information attached to the node. The formal definition of *compatibility* between logics and languages, which must be satisfied by an underlying logic [4], is not given here. It will suffice to say that the intended meaning of compatibility is that if two expressions are proved to be equal under the logic, they calculate the same result.

3.2 Rules for $\mathcal{G}_0$

GPC transformation method $\mathcal{G}_0$ is defined by the set of fourteen rules shown in Fig.2. Write $\mathcal{G}_0[\![e]\!]E$ to denote the residual expression of specializing expression e with the information E. Here $l \leftrightarrow_E^* r$ means that there is a finite sequence of $l = t_1 \leftrightarrow t_2 \leftrightarrow \cdots \leftrightarrow t_n = r$ $(n \geq 1)$ of application of equations in $E \cup E_0$. $FV(e)$ denotes the set of free variables in e.

It is easy to examine that the rules cover all possible expressions: of the five kinds of expression (variable, constructor application, primitive function application, function application, case term) the first four are covered directly, and for case terms, all five possibilities for the selector are considered.

It is required that $\mathcal{G}_0[\![e]\!]E = e$ whenever e satisfies the constraints E. That is, $\mathcal{G}_0[\![e]\!]E$ and e should compute the same value if e satisfies E. It is clear that each of the rules preserves equivalence.

In rules (2), (3), and (5), the basic form of the expression is not changed, and the components are converted recursively.

Rules (1), (4), (7), and (11) are applied if the expression can be reduced to the constructor application form using the information E. The simplified expressions are converted recursively. The `let` construct is used in the residual program to introduce local variables.

Rule (6) introduces a new function f' to continue specializing that function application. It emulates the popular loop for program transformation: instantiate/unfold/simplify/fold [9]. This is the only source of nontermination. Later we will elaborate this rule to make the transformation terminate more often.

In rule (9), the form of the expression tells which branch should be used, and the right hand expression of that branch is converted recursively. Rule (10) is used to eliminate the expression which is never used to calculate the result. We use $\perp$ to represent these useless expression. It is possible to eliminate the case branch whose right hand expression is $\perp$.

In (8), (12), and (13), we don't know which branch is used to calculate the result. The form of the case term is preserved and the components are converted recursively. Each branch has different information, which reflects the semantics of the case term: the right hand expression of each branch is selected to calculate the result if and only if the selector matches with the case pattern. The information for each branch is extended with an identity which expresses that the selector matches the corresponding case pattern.

Variables in the identities are treated as constants in the equational reasoning. Especially, variables in the case pattern should be renamed in the branch to avoid conflict with the constant names in E.

After adding the identity, the consistency of the information $E \cup \{e \leftrightarrow p_i\}$ is checked. If it generates the contradiction $(F \leftrightarrow^* T)$, the corresponding branch $p_i : e_i'$ is eliminated from the residual case term.

This extension of the information is the central idea of GPC and the most essential difference from conventional partial evaluation. It can be said that the whole GPC method is devised to utilize this extended information.

For rule (14), the nested case term is simplified, and the result is converted recursively.

3.3 Examples

We illustrate the application of the method $\mathcal{G}_0$ with a simple example. Let

$$
\begin{array}{l}
f\ n\ =\ \textbf{case}\ n \leq 1\ \textbf{of} \\
\qquad\qquad T\ :\ 0 \\
\qquad\qquad F\ :\ (\textbf{case}\ odd(n)\ \textbf{of} \\
\qquad\qquad\qquad\qquad T\ :\ f(n-1) \\
\qquad\qquad\qquad\qquad F\ :\ n-2 \qquad)
\end{array}
$$

The transformation of the expression $f\ n$ is shown in Fig.3.

It is obvious that this version of $\mathcal{G}_0$ is too weak. It works well only with an expression which is transformed into one without recursion after finite unfolding. If the given expression is inherently recursive (e.g. function h in Fig.1), this method always goes into the infinite loop. As mentioned above, the only source of nontermination is rule (6). We elaborate this rule to avoid nontermination.

3.4 Memoization for $\mathcal{G}_0$

Infinite introduction of new functions causes the infinite loop. The memoization trick is used to avoid this infinite regress. Each new function introduced during

$$\mathcal{G}_0[\![v]\!]\ E = c \quad \text{if } v \leftrightarrow_E^* c \tag{1}$$
$$= v \quad \text{otherwise} \tag{2}$$

$$\mathcal{G}_0[\![c\ e_1 \ldots e_n]\!]\ E = c\ (\mathcal{G}_0[\![e_1]\!]\ E) \ldots (\mathcal{G}_0[\![e_n]\!]\ E) \tag{3}$$

$$\mathcal{G}_0[\![p\ e_1 \ldots e_n]\!]\ E = \textbf{let } c\ v_1 \ldots v_k = p\ e_1 \ldots e_n \textbf{ in} \tag{4}$$
$$\mathcal{G}_0[\![c\ v_1 \ldots v_k]\!]\ E \cup \{v_1 \leftrightarrow e'_1, \ldots, v_k \leftrightarrow e'_k\}$$
$$\text{if } p\ e_1 \ldots e_n \leftrightarrow_E^* c\ e'_1 \ldots e'_k$$
$$= p\ (\mathcal{G}_0[\![e_1]\!]\ E) \ldots (\mathcal{G}_0[\![e_n]\!]\ E) \quad \text{otherwise} \tag{5}$$

$$\mathcal{G}_0[\![f\ e_1 \ldots e_n]\!]\ E = f'\ v_1 \ldots v_k \tag{6}$$
where f is defined as: $f\ x_1 \ldots x_n \stackrel{\text{def}}{=} e$
Define a new function f' by:
$$f'\ v_1 \ldots v_k \stackrel{\text{def}}{=} \mathcal{G}_0[\![e[x_1 := e_1, \ldots, x_n := e_n]]\!]\ E$$
where $\{v_1 \ldots v_k\} = FV(e_1) \cup \cdots \cup FV(e_n)$

$$\mathcal{G}_0[\![\textbf{case } v \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E$$
$$= \textbf{let } c\ v_1 \ldots v_k = v \textbf{ in} \tag{7}$$
$$\mathcal{G}_0[\![\textbf{case } c\ v_1 \ldots v_k \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E \cup \{v_1 \leftrightarrow e''_1, \ldots, v_k \leftrightarrow e''_k\}$$
$$\text{if } v \leftrightarrow_E^* c\ e''_1 \ldots e''_k$$
$$= \textbf{case } v \textbf{ of } p_1 : \mathcal{G}_0[\![e'_1]\!]\ E \cup \{v \leftrightarrow p_1\} | \ldots | p_m : \mathcal{G}_0[\![e'_m]\!]\ E \cup \{v \leftrightarrow p_m\} \tag{8}$$
$$\text{otherwise}$$

$$\mathcal{G}_0[\![\textbf{case } c\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E$$
$$= \mathcal{G}_0[\![e'_i[x_1 := e_1, \ldots, x_n := e_n]]\!]\ E \quad \text{if } p_i = c\ x_1 \ldots x_n \tag{9}$$
$$= \bot \quad \text{otherwise} \tag{10}$$

$$\mathcal{G}_0[\![\textbf{case } p\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E$$
$$= \textbf{let } c\ v_1 \ldots v_k = p\ e_1 \ldots e_n \textbf{ in} \tag{11}$$
$$\mathcal{G}_0[\![\textbf{case } c\ v_1 \ldots v_k \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E \cup \{v_1 \leftrightarrow e''_1, \ldots, v_k \leftrightarrow e''_k\}$$
$$\text{if } p\ e_1 \ldots e_n \leftrightarrow_E^* c\ e''_1 \ldots e''_k$$
$$= \textbf{case } e \textbf{ of} \tag{12}$$
$$p_1 : \mathcal{G}_0[\![e'_1]\!]\ E \cup \{e \leftrightarrow p_1\} | \ldots | p_m : \mathcal{G}_0[\![e'_m]\!]\ E \cup \{e \leftrightarrow p_m\}$$
$$\textbf{where } \mathcal{G}_0[\![p\ e_1 \ldots e_n]\!]\ E = e$$
$$\text{otherwise}$$

$$\mathcal{G}_0[\![\textbf{case } f\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ E$$
$$= \textbf{case } e \textbf{ of} \tag{13}$$
$$p_1 : \mathcal{G}_0[\![e'_1]\!]\ E \cup \{e \leftrightarrow p_1\} | \ldots | p_m : \mathcal{G}_0[\![e'_m]\!]\ E \cup \{e \leftrightarrow p_m\}$$
$$\textbf{where } \mathcal{G}_0[\![f\ e_1 \ldots e_n]\!]\ E = e$$

$$\mathcal{G}_0[\![\textbf{case } (\textbf{case } e \textbf{ of } p_1 : e_1 | \ldots | p_n : e_n) \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m]\!]\ E$$
$$= \mathcal{G}_0[\![\textbf{case } e \textbf{ of} \tag{14}$$
$$p_1 : (\textbf{case } e_1 \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m) | \ldots |$$
$$p_n : (\textbf{case } e_n \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m)]\!]\ E$$

Figure 2: Transformation rules for $\mathcal{G}_0$

$$\mathcal{G}_0[\![f\ e_1 \ldots e_n]\!]\ E \ = \ (f'\ v_1 \ldots v_k)\theta \qquad \text{(6-1)}$$

if $^{\exists} f'$: memoized function s.t.

memoized with call pattern : $f\ e'_1 \ldots e'_n$ and information : E'

f' is defined as : $f'\ v_1 \ldots v_k \stackrel{\text{def}}{=} e$

and $^{\exists}\theta$: substitution over variables s.t.

$\{f\ e'_1 \ldots e'_n \mid E'\} \ = \ \{(f\ e'_1 \ldots e'_n)\theta \mid E'\theta\}$,

$f\ e_1 \ldots e_n \leftrightarrow^*_{E'} (f\ e'_1 \ldots e'_n)\theta$,

$\{f\ e_1 \ldots e_n \mid E\} \ = \ \{f\ e_1 \ldots e_n \mid E'\theta\}$

$$= \ f'\ v_1 \ldots v_k \qquad \text{(6-2)}$$

where

f is defined as: $f\ x_1 \ldots x_n \stackrel{\text{def}}{=} e$

Define a new function f' by:

$f'\ v_1 \ldots v_k \stackrel{\text{def}}{=} \mathcal{G}_0[\![e[x_1 := e_1, \ldots, x_n := e_n]]\!]\ E$

where $\{v_1 \ldots v_k\} \ = \ FV(e_1) \cup \cdots \cup FV(e_n)$,

and memoize f' with call pattern : $f\ e_1 \ldots e_n$ and information : E

otherwise

Figure 4: Improved rules for $\mathcal{G}_0$ using memoization

$\mathcal{G}_0[\![f\ n]\!]\ \emptyset \ = \ f_0\ n$ (by (6))

where

$f_0\ n$

$\stackrel{\text{def}}{=}$ $\mathcal{G}_0[\![$**case** $n \leq 1$ **of**

T : 0

F : (**case** $odd(n)$ **of**

T : $f(n-1)$

F : $n-2$ $)$ $]\!]\ \emptyset$

$=$ **case** $n \leq 1$ **of** (by (12)(3)(12))

T : 0

F : (**case** $odd(n)$ **of**

T : $\mathcal{G}_0[\![f(n-1)]\!]$

$\{n \leq 1 \leftrightarrow F,\ odd(n) \leftrightarrow T\}$

F : $\mathcal{G}_0[\![n-2]\!]$

$\{n \leq 1 \leftrightarrow F,\ odd(n) \leftrightarrow F\}$)

$=$ **case** $n \leq 1$ **of** (by (6)(5)(2))

T : 0

F : (**case** $odd(n)$ **of**

T : $f_1\ n$

F : $n-2$)

where

$f_1\ n$

$\stackrel{\text{def}}{=}$ $\mathcal{G}_0[\![$**case** $n-1 \leq 1$ **of** (by (6))

T : 0

F : (**case** $odd(n-1)$ **of**

T : $f(n-1-1)$

F : $n-1-2$ $)]\!]$

$\{n \leq 1 \leftrightarrow F,\ odd(n) \leftrightarrow T\}$

$=$ $\mathcal{G}_0[\![$**case** $odd(n-1)$ **of** (by (11)(9))

T : $f(n-2)$

F : $n-3$ $]\!]$

$\{n \leq 1 \leftrightarrow F,\ odd(n) \leftrightarrow T\}$

$=$ $\mathcal{G}_0[\![n-3]\!]\{n \leq 1 \leftrightarrow F,\ odd(n) \leftrightarrow T\}$ (by (11)(9))

$=$ $n-3$ (by (5)(2))

Figure 3: Example transformation with $\mathcal{G}_0$

the transformation is memoized with the original call pattern and the information attached to it. Upon introduction of a new function, the expression and the information are checked against the already memoized functions to determine they deserve the new function.

Here we state a required property (or a conjecture) for $\mathcal{G}_0$ without proof. We cannot prove that the current definition of $\mathcal{G}_0$ satisfies this because the properties of the underlying logic and its relation to the semantics of the language has not been studied enough. Further work is needed in this direction.

Requirement

If a substitution θ over variables in expression e has the property $\{e \mid E\} \ = \ \{e\theta \mid E\theta\}$ as the set of ground expressions,

then

$(\mathcal{G}_0[\![e]\!]\ E)\theta \ = \ \mathcal{G}_0[\![e\theta]\!]\ (E\theta)$ holds.

The criterion for reusability of the memoized function follows directly. Rule (6) is safely replaced with the two rules shown in Fig.4.

If there exist a memoized function f' and a substitution θ which satisfy the condition of rule (6-1), the following equation holds. This justifies rule (6-1).

$$\begin{aligned} \mathcal{G}_0[\![f\ e_1 \ldots e_n]\!]\ E \ &= \ \mathcal{G}_0[\![(f\ e'_1 \ldots e'_n)\theta]\!]\ (E'\theta) \\ &= \ (\mathcal{G}_0[\![f\ e'_1 \ldots e'_n]\!]\ E')\theta \\ &= \ (f'\ v_1 \ldots v_k)\theta \end{aligned}$$

With this new set of fifteen rules, the expression $h\ n$ with the function h in Fig.1 is successfully transformed as follows:

$\mathcal{G}_0[\![h\ n]\!]\ \emptyset \ = \ h_0\ n$ (by (6-2))

where h_0 **is memoized with call pattern** : $h\ n$

and information : $\emptyset$

$h_0\ n$

$\stackrel{\text{def}}{=}\ \mathcal{G}_0[\![\textbf{case}\ n \le 1\ \textbf{of}$
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n)\ \textbf{of}$
$\quad\quad T : h(n-1) + h(n-2)$
$\quad\quad F : h(n-1)\quad)\ ]\!]\ \emptyset$

$=$ **case** $n \le 1$ **of** (by (12)(3)(12)(5))
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n)\ \textbf{of}$
$\quad\quad T : \mathcal{G}_0[\![h(n-1)]\!]$
$\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$
$\quad\quad\quad + \mathcal{G}_0[\![h(n-2)]\!]$
$\quad\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$
$\quad\quad F : \mathcal{G}_0[\![h(n-1)]\!]$
$\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto F\}\quad)$

$=$ **case** $n \le 1$ **of** (by (6-2))
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n)\ \textbf{of}$
$\quad\quad T : h_1\ n + \mathcal{G}_0[\![h(n-2)]\!]$
$\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$
$\quad\quad F : \mathcal{G}_0[\![h(n-1)]\!]$
$\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto F\}\quad)$

where h_1 is memoized with call pattern : $h\ (n-1)$
and information : $\{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$

$h_1\ n$

$\stackrel{\text{def}}{=}\ \mathcal{G}_0[\![\textbf{case}\ n-1 \le 1\ \textbf{of}$
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n-1)\ \textbf{of}$
$\quad\quad T : h(n-1-1) + h(n-1-2)$
$\quad\quad F : h(n-1-1)\quad)\ ]\!]$
$\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$

$=\ \mathcal{G}_0[\![\textbf{case}\ odd(n-1)\ \textbf{of}$ (by (11)(9))
$\quad T : h(n-2) + h(n-3)$
$\quad F : h(n-2)\quad]\!]$
$\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$

$=\ \mathcal{G}_0[\![h(n-2)]\!]\{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$ (by (11)(9))

$=\ h_2\ n$ (by (6-2))

where h_2 is memoized with call pattern : $h\ (n-2)$
and information : $\{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$

$h_2\ n$

$\stackrel{\text{def}}{=}\ \mathcal{G}_0[\![\textbf{case}\ n-2 \le 1\ \textbf{of}$
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n-2)\ \textbf{of}$
$\quad\quad T : h(n-2-1) + h(n-2-2)$
$\quad\quad F : h(n-2-1)\quad)\ ]\!]$
$\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$

$=$ **case** $n \le 3$ **of** (by (12)(3))
$\quad T : 1$
$\quad F : \mathcal{G}_0[\![\textbf{case}\ odd(n-2)\ \textbf{of}$
$\quad\quad T : h(n-3) + h(n-4)$
$\quad\quad F : h(n-3)\quad]\!]$
$\quad\quad \{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$

$=$ **case** $n \le 3$ **of** (by (11)(9)(5))
$\quad T : 1$
$\quad F : \mathcal{G}_0[\![h(n-3)]\!]\{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$
$\quad\quad + \mathcal{G}_0[\![h(n-4)]\!]\{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$

For term $\mathcal{G}_0[\![h(n-3)]\!]\{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$,
function h_1 and substitution $\theta : n \mapsto n-2$
satisfy the condition for reusability of (6-1).

$=$ **case** $n \le 3$ **of** (by (6-1))
$\quad T : 1$
$\quad F : h_1\ (n-2)$
$\quad\quad + \mathcal{G}_0[\![h(n-4)]\!]\{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$

For term $\mathcal{G}_0[\![h(n-4)]\!]\{n \le 3 \mapsto F,\ odd(n) \mapsto T\}$,
function h_2 and substitution $\theta : n \mapsto n-2$
satisfy the condition for reusability of (6-1).

$=$ **case** $n \le 3$ **of** (by (6-1))
$\quad T : 1$
$\quad F : h_1\ (n-2) + h_2\ (n-2)$

Return to the intermitted transformation of $h_0\ n$,

For term $\mathcal{G}_0[\![h(n-2)]\!]\{n \le 1 \mapsto F,\ odd(n) \mapsto T\}$,
function h_2 and substitution $\theta : n \mapsto n$
satisfy the condition for reusability of (6-1).

$h_0\ n$

$=$ **case** $n \le 1$ **of** (by (6-1))
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n)\ \textbf{of}$
$\quad\quad T : h_1\ n + h_2\ n$
$\quad\quad F : \mathcal{G}_0[\![h(n-1)]\!]$
$\quad\quad\quad \{n \le 1 \mapsto F,\ odd(n) \mapsto F\}\ )$

For term $\mathcal{G}_0[\![h(n-1)]\!]\{n \le 1 \mapsto F,\ odd(n) \mapsto F\}$,
function h_2 and substitution $\theta : n \mapsto n+1$
satisfy the condition for reusability of (6-1).

$=$ **case** $n \le 1$ **of** (by (6-1))
$\quad T : 1$
$\quad F : (\textbf{case}\ odd(n)\ \textbf{of}$
$\quad\quad T : h_1\ n + h_2\ n$
$\quad\quad F : h_2\ (n+1)\quad)$

4 Representing Contexts with Projections

In this section we introduce how projections can be used to model contexts of programs. This form of context information will be utilized in our new GPC method for a lazy language in section 5. The formulation of contexts using projections is first provided by Wadler and Hughes in the analysis of strictness [8]. For a fuller and formal development the reader is referred to [3,8].

The basic problem concerning contexts is, given a function, how much information is required from the argument in order to determine a certain amount of information about the result of the function. The notion of projection from domain theory can provide a concise description of the required amount of information.

Definition 4.1 *A projection α is a continuous function from a domain $\mathcal{D}$ to itself, such that*

$$\begin{aligned} \alpha &\sqsubseteq \mathrm{ID} \\ \alpha \circ \alpha &= \alpha \end{aligned}$$

The first condition says that a projection only remove information from its argument, but does not change its type. The second condition says that all the information is removed at once, so further application has no

effect. A projection is used to represent the context, where the information removed represents information not needed by that context. In the following the terms *projection* and *context* will be used interchangeably. α and β (sometimes subscripted) will always denote projections.

Definition 4.2 (Safe Projections) *Given a projection α and a function f of arity n, if a projection β satisfies the safety requirement*

$$\alpha(f(u_1, \ldots, u_n)) \sqsubseteq f(u_1, \ldots, (\beta u_i), \ldots, u_n)$$

for all objects $u_1, \ldots, u_n$, then we say β is a safe context for the i'th argument of f in context α.
And we write $f^i : \alpha \Rightarrow \beta$.

To specify necessity with projections Wadler and Hughes extended a domain with a new element $\lightning$, called "abort". The interpretation of $\alpha u = \lightning$ is that context α *requires* a value more defined than u. Simple strictness is defined with the projection STR, where STR maps $\perp$ to $\lightning$, and acts as the identity on all other values. To make this work, $\lightning$ must be a new bottom element to be added to every domain $\mathcal{D}$. This new domain is written $\mathcal{D}_{\lightning}$. Every function $f : \mathcal{D}_1 \rightarrow \mathcal{D}_2$ is naturally extended to a function in $\mathcal{D}_{1\lightning} \rightarrow \mathcal{D}_{2\lightning}$ by making strict in $\lightning$. A formal development of these technical devices are studied in [2].

Projections form a complete latice under the ordering $\sqsubseteq$, with the identity function ID as the greatest element and FAIL as the least element. FAIL is defined by

$$\text{FAIL } u = \lightning, \text{ for all u}$$

Another useful projection is ABS, defined by

$$\begin{array}{lcl} \text{ABS } \lightning & = & \lightning \\ \text{ABS } u & = & \perp,\ u \neq \lightning \end{array}$$

The notions of strictness and ignored arguments can be defined using these.

Definition 4.3 *f is strict on the i'th argument iff*

$$f^i : \text{STR} \Rightarrow \text{STR}$$

Definition 4.4 *f ignores its i'th argument iff*

$$f^i : \text{STR} \Rightarrow \text{ABS}$$

For each constructor c_i, we define projections of the form $\mathrm{C}_i\ \alpha_1 \ldots \alpha_{n_i}$ where

$$\mathrm{C}_i\ \alpha_1 \ldots \alpha_{n_i}(c_j\ u_1 \ldots u_{n_j}) = \begin{cases} c_j(\alpha_1 u_1) \ldots (\alpha_{n_j} u_{n_j}) & \text{if } j = i \\ \lightning, & \text{otherwise} \end{cases}$$

For each sum-of-products type with constructors $c_1, \ldots, c_m$, FAIL and the projections of the form $\sqcup_{i=1}^{m} C_i\ \alpha_{i,1} \ldots \alpha_{i,n_i}$ consist a complete lattice.

Analysis of context is a *backward analysis* : given a context α for a function f, we want to know the smallest context β_i for each argument which satisfies the condition $f^i : \alpha \Rightarrow \beta_i$.

The *backward strictness analysis* given in [3,8] provides us a computable approximation of the ideal solution for the problem. Given a set of function definitions, the analysis calculates a set of projection transformers $f^{\#i}$ for each function, which satisfy the safety property

$$f^i : \alpha \Rightarrow f^{\#i}\alpha, \qquad \text{for all } \alpha.$$

5 GPC method $\mathcal{G}$ for a Lazy Language

In this section we concentrate on a lazy first-order functional language and define a GPC method $\mathcal{G}$ which adapts to it. $\mathcal{G}$ is an extended version of $\mathcal{G}_0$ to utilize context information of program to eliminate redundant transformation. $\mathcal{G}$ terminates more often than $\mathcal{G}_0$.

The structure of GPC method is analogous to that of backward analysis. It propagates the information at the root node toward the leaves to yield the information for each lower nodes with appropriate transformation. The information propagated by GPC method $\mathcal{G}_0$ is the constraints on each node which are assumable for the value of that node whenever its value is required. It does not include the context information about how much definedness is required for the value of each node.

For analyzing a language with strict semantics, it is sufficient to see whether the value of the expression can be needed or not, because once the value of the expression is required, the whole part of the value must be calculated. Since the calculation in a lazy language is done as much as it is needed to calculate the result of the root node, the context information about required definedness should be taken into account in the analysis of a lazy language. Without these information the transformation may go further into the expression whose value would never be used, and it may leads us to non-termination.

To treat these context information properly, we adopt the formulation of a context used by the backward strictness analysis shown in section 4. The context information is represented by a projection.

In order to adapt our GPC method to analyzing a lazy language, the rules for transformation are parameterized with a projection which represents a context corresponding to the expression.

$$\mathcal{G}[\![e]\!]\ \alpha\ E = \bot \quad \text{if } \alpha \sqsubseteq \mathbf{ABS} \tag{1}$$

$$\mathcal{G}[\![v]\!]\ \alpha\ E = c \quad \text{if } v \mapsto^*_E c \tag{2}$$

$$= v \quad \text{otherwise} \tag{3}$$

$$\mathcal{G}[\![c\ e_1 \ldots e_n]\!]\ \alpha\ E = c\ (\mathcal{G}[\![e_1]\!]\ \alpha_1\ E) \ldots (\mathcal{G}[\![e_n]\!]\ \alpha_n\ E) \quad \text{if } {}^\exists\beta \text{ s.t. } \alpha = (C\ \alpha_1 \ldots \alpha_n) \sqcup \beta \tag{4}$$

$$= \bot \quad \text{otherwise} \tag{5}$$

$$\mathcal{G}[\![p\ e_1 \ldots e_n]\!]\ \alpha\ E = \textbf{let } c\ v_1 \ldots v_k = p\ e_1 \ldots e_n \textbf{ in} \tag{6}$$

$$\mathcal{G}[\![c\ v_1 \ldots v_k]\!]\ \alpha\ E \cup \{v_1 \mapsto e'_1, \ldots, v_k \mapsto e'_k\} \quad \text{if } p\ e_1 \ldots e_n \mapsto^*_E c\ e'_1 \ldots e'_k$$

$$= p\ (\mathcal{G}[\![e_1]\!](p^{\#1}\alpha)E) \ldots (\mathcal{G}[\![e_n]\!](p^{\#n}\alpha)E) \quad \text{otherwise} \tag{7}$$

$$\mathcal{G}[\![f\ e_1 \ldots e_n]\!]\ \alpha\ E = (f'\ v_1 \ldots v_k)\theta \tag{8}$$

where ${}^\exists f'$: function memoised with
call pattern : $f\ e'_1 \ldots e'_n$, context : α, and information : E'
f' is defined as : $f'\ v_1 \ldots v_k \stackrel{\text{def}}{=} e$
and ${}^\exists\theta$: substitution over variables s.t.
$\{f\ e'_1 \ldots e'_n \mid E'\} = \{(f\ e'_1 \ldots e'_n)\theta \mid E'\theta\}$,
$f\ e_1 \ldots e_n \mapsto^*_{E'} (f\ e'_1 \ldots e'_n)\theta$,
$\{f\ e_1 \ldots e_n \mid E\} = \{f\ e_1 \ldots e_n \mid E'\theta\}$

$$= f'\ v_1 \ldots v_k \tag{9}$$

where f is defined as: $f\ x_1 \ldots x_n \stackrel{\text{def}}{=} e$
Define a new function f' by: $f'\ v_1 \ldots v_k \stackrel{\text{def}}{=} \mathcal{G}[\![e[x_1 := e_1, \ldots, x_n := e_n]]\!]\ \alpha\ E$
where $\{v_1 \ldots v_k\} = FV(e_1) \cup \cdots \cup FV(e_n)$,
and memoize f' with call pattern : $f\ e_1 \ldots e_n$, context : α, and information : E
otherwise

$$\mathcal{G}[\![\textbf{case } v \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E$$

$$= \textbf{let } c\ v_1 \ldots v_k = v \textbf{ in} \tag{10}$$

$$\mathcal{G}[\![\textbf{case } c\ v_1 \ldots v_k \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E \cup \{v_1 \mapsto e''_1, \ldots, v_k \mapsto e''_k\}$$
$$\text{if } v \mapsto^*_E c\ e''_1 \ldots e''_k$$

$$= \textbf{case } v \textbf{ of } p_1 : \mathcal{G}[\![e'_1]\!]\ \alpha\ E \cup \{v \mapsto p_1\} | \ldots | p_m : \mathcal{G}[\![e'_m]\!]\ \alpha\ E \cup \{v \mapsto p_m\} \tag{11}$$
$$\text{otherwise}$$

$$\mathcal{G}[\![\textbf{case } c\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E$$

$$= \mathcal{G}[\![e'_i[x_1 := e_1, \ldots, x_n := e_n]]\!]\ \alpha\ E \quad \text{if } p_i = c\ x_1 \ldots x_n \tag{12}$$

$$= \bot \quad \text{otherwise} \tag{13}$$

$$\mathcal{G}[\![\textbf{case } p\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E$$

$$= \textbf{let } c\ v_1 \ldots v_k = p\ e_1 \ldots e_n \textbf{ in} \tag{14}$$

$$\mathcal{G}[\![\textbf{case } c\ v_1 \ldots v_k \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E \cup \{v_1 \mapsto e''_1, \ldots, v_k \mapsto e''_k\}$$
$$\text{if } p\ e_1 \ldots e_n \mapsto^*_E c\ e''_1 \ldots e''_k$$

$$= \textbf{case } e \textbf{ of} \tag{15}$$

$$p_1 : \mathcal{G}[\![e'_1]\!]\ \alpha\ E \cup \{e \mapsto p_1\} | \ldots | p_m : \mathcal{G}[\![e'_m]\!]\ \alpha\ E \cup \{e \mapsto p_m\}$$

where $\mathcal{G}[\![p\ e_1 \ldots e_n]\!]\ \beta\ E = e$ (β is a safe context for the selector)
otherwise

$$\mathcal{G}[\![\textbf{case } f\ e_1 \ldots e_n \textbf{ of } p_1 : e'_1 | \ldots | p_m : e'_m]\!]\ \alpha\ E$$

$$= \textbf{case } e \textbf{ of} \tag{16}$$

$$p_1 : \mathcal{G}[\![e'_1]\!]\ \alpha\ E \cup \{e \mapsto p_1\} | \ldots | p_m : \mathcal{G}[\![e'_m]\!]\ \alpha\ E \cup \{e \mapsto p_m\}$$

where $\mathcal{G}[\![f\ e_1 \ldots e_n]\!]\ \beta\ E = e$ (β is a safety context for the selector)

$$\mathcal{G}[\![\textbf{case (case } e \textbf{ of } p_1 : e_1 | \ldots | p_n : e_n) \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m]\!]\ \alpha\ E$$

$$= \mathcal{G}[\![\textbf{case } e \textbf{ of} \tag{17}$$

$$p_1 : (\textbf{case } e_1 \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m) | \ldots |$$
$$p_n : (\textbf{case } e_n \textbf{ of } p'_1 : e'_1 | \ldots | p'_m : e'_m)]\!]\ \alpha\ E$$

Figure 5: Transformation rules for $\mathcal{G}$

5.1 Rules for $\mathcal{G}$

Our extended GPC method $\mathcal{G}[\![\cdot]\!]$ is defined by the set of seventeen rules shown in Fig.5. Write $\mathcal{G}[\![e]\!]\,\alpha\,E$ to denote the residual expression of specializing expression e with the information E in the context α.

The context information is treated in the similar way to the constraints. It is propagated to the subexpressions with proper trasformations. The point is any expressions in the context ABS will never be used. We can safely replace them with dummy values (we use $\perp$ for this purpose).

Before applying these rules, the backward strictness analysis is applied to the set of primitive functions and user defined functions. In the following it is assumed that we already know the set of projection transformers for the primitive functions and user defined functions. For each primitive function p of arity n, the projection transformers $p^{\#1},\ldots,p^{\#n}$ are defined to have the safety property $p^i : \alpha \Rightarrow p^{\#i}\alpha$, for all context α. Projection transformers $f^{\#1},\ldots,f^{\#n}$ are also calculated for each user defined function f of arity n, to satisfy corresponding safety condition.

In rule (15) and (16) we need a safe context β for the selector. It can be calculated as follows. For each case branch, the context for right hand expression is α. Using known projection transformers, a safe context for each case variable can be calculated. Putting these contexts together with the projection transformer corresponding to the type constructor in the case pattern, we get the projection for each case branch. The least upper bound over the set of projections for all case branches gives a context β.

5.2 Examples

We illustrate the application of the method $\mathcal{G}$ with a simple example. Let

$$
\begin{array}{lll}
\mathit{append}\ x\ y & = & \textbf{case}\ x\ \textbf{of} \\
 & & \quad \mathit{Nil} \quad : y \\
 & & \quad \mathit{Cons}\ z\ zs \ : \mathit{Cons}\ z\ (\mathit{append}\ zs\ y) \\
\mathit{squares}\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad \mathit{Nil} \quad : \mathit{Nil} \\
 & & \quad \mathit{Cons}\ y\ ys \ : \mathit{Cons}\ (y*y)\ (\mathit{squares}\ ys) \\
\mathit{second}\ xs & = & \mathit{hd}\ (\mathit{tail}\ xs)
\end{array}
$$

The example transformation is for the expression *second* (*squares* (*append* x y))
with information $\{x \mapsto \mathit{Cons}\ a\ (\mathit{Cons}\ b\ c)\}$ in context STR. In the following, a projection transformer CONS which corresponds to the constructor *Cons* is written in infix notation (:). For example, ABS : (STR : ABS) is equivalent to (CONS ABS (CONS STR ABS)).

$\mathcal{G}[\![\mathit{second}\ (\mathit{squares}\ (\mathit{append}\ x\ y))]\!]$ STR
$\quad \{x \mapsto \mathit{Cons}\ a\ (\mathit{Cons}\ b\ c)\}$
$= \mathit{second}_0\ x\ y$ (by (9))

where second_0 is memoized with
call pattern : *second* (*squares* (*append* x y)),
context : STR, and
information : $E = \{x \mapsto \mathit{Cons}\ a\ (\mathit{Cons}\ b\ c)\}$

$\mathit{second}_0\ x\ y$
$\stackrel{\text{def}}{=} \mathcal{G}[\![\mathit{hd}\ (\mathit{tail}\ (\mathit{squares}\ (\mathit{append}\ x\ y)))]\!]$ STR E
$= \mathit{hd}\ (\mathit{tail}\ (\mathcal{G}[\![\mathit{squares}\ (\mathit{append}\ x\ y)]\!]$
$\qquad (\text{ABS} : (\text{STR} : \text{ABS}))\ E))$ (by (7)(7))
$= \mathit{hd}\ (\mathit{tail}\ (\mathit{squares}_0\ x\ y))$ (by (9))

where $\mathit{squares}_0$ is memoized with
call pattern : *squares* (*append* x y),
context : (ABS:(STR:ABS)), and
information : $E = \{x \mapsto \mathit{Cons}\ a\ (\mathit{Cons}\ b\ c)\}$

$\mathit{squares}_0\ x\ y$
$\stackrel{\text{def}}{=} \mathcal{G}[\![\textbf{case}\ (\mathit{append}\ x\ y)\ \textbf{of}$
$\qquad \mathit{Nil} \quad : \mathit{Nil}$
$\qquad \mathit{Cons}\ z\ zs \ : \mathit{Cons}\ (z*z)\ (\mathit{squares}\ zs)]\!]$
$\quad (\text{ABS} : (\text{STR} : \text{ABS}))\ E$
$=$ **case** $(\mathit{append}_0\ x\ y)$ **of** (by (16))
$\quad \mathit{Nil} \quad : \mathcal{G}[\![\mathit{Nil}]\!]\ (\text{ABS} : (\text{STR} : \text{ABS}))$
$\qquad E \cup \{\mathit{append}_0\ x\ y \mapsto \mathit{Nil}\}$
$\quad \mathit{Cons}\ z\ zs \ : \mathcal{G}[\![\mathit{Cons}\ (z*z)\ (\mathit{squares}\ zs)]\!]$
$\qquad (\text{ABS} : (\text{STR} : \text{ABS}))$
$\qquad E \cup \{\mathit{append}_0\ x\ y \mapsto \mathit{Cons}\ z\ zs\}$
$=$ **case** $(\mathit{append}_0\ x\ y)$ **of** (by (5)(4))
$\quad \mathit{Nil} \quad : \perp$
$\quad \mathit{Cons}\ z\ zs \ : \mathit{Cons}$
$\qquad \mathcal{G}[\![z*z]\!]\ \text{ABS}$
$\qquad\quad E \cup \{\mathit{append}_0\ x\ y \mapsto \mathit{Cons}\ z\ zs\}$
$\qquad \mathcal{G}[\![\mathit{squares}\ zs]\!]\ (\text{STR} : \text{ABS})$
$\qquad\quad E \cup \{\mathit{append}_0\ x\ y \mapsto \mathit{Cons}\ z\ zs\}$
$=$ **case** $(\mathit{append}_0\ x\ y)$ **of** (by (1)(9))
$\quad \mathit{Cons}\ z\ zs \ : \mathit{Cons}\ \perp\ (\mathit{squares}_1\ zs)$

where $\mathit{squares}_1$ is memoized with
call pattern : *squares* zs,
context : (STR:ABS), and
information : $E' = \{\ x \mapsto \mathit{Cons}\ a\ (\mathit{Cons}\ b\ c),$
$\qquad\qquad \mathit{append}_0\ x\ y \mapsto \mathit{Cons}\ z\ zs\}$

$\mathit{squares}_1\ zs$
$\stackrel{\text{def}}{=} \mathcal{G}[\![\textbf{case}\ zs\ \textbf{of}$
$\qquad \mathit{Nil} \quad : \mathit{Nil}$
$\qquad \mathit{Cons}\ u\ us \ : \mathit{Cons}\ (u*u)\ (\mathit{squares}\ us)]\!]$
$\quad (\text{STR} : \text{ABS})\ E'$
$=$ **case** zs **of** (by (11))
$\quad \mathit{Nil} \quad : \mathcal{G}[\![\mathit{Nil}]\!]\ (\text{STR} : \text{ABS})$
$\qquad E' \cup \{zs \mapsto \mathit{Nil}\}$
$\quad \mathit{Cons}\ u\ us \ : \mathcal{G}[\![\mathit{Cons}\ (u*u)\ (\mathit{squares}\ us)]\!]$
$\qquad (\text{STR} : \text{ABS})$
$\qquad E' \cup \{zs \mapsto \mathit{Cons}\ u\ us\}$
$=$ **case** zs **of** (by (5)(4))
$\quad \mathit{Nil} \quad : \perp$
$\quad \mathit{Cons}\ u\ us \ : \mathit{Cons}\ \mathcal{G}[\![u*u]\!]\ \text{STR}$
$\qquad\qquad E' \cup \{zs \mapsto \mathit{Cons}\ u\ us\}$
$\qquad\quad \mathcal{G}[\![\mathit{squares}\ us]\!]\ \text{ABS}$
$\qquad\qquad E' \cup \{zs \mapsto \mathit{Cons}\ u\ us\}$
$=$ **case** zs **of** (by (7)(3)(3)(1))
$\quad \mathit{Cons}\ u\ us \ : \mathit{Cons}\ (u*u)\ \perp$

Return to the suspended introduction of $append_0$,

$append_0$ is memoized with
call pattern : $append\ x\ y$,
context : (ABS:(STR:ABS)), and
information : $E = \{x \mapsto Cons\ a\ (Cons\ b\ c)\}$

$$
\begin{array}{ll}
& append_0\ x\ y \\
\stackrel{\text{def}}{=} & \mathcal{G}[\![\text{case } x \text{ of} \\
& \quad Nil \quad : \ y \\
& \quad Cons\ z\ zs \ : \ Cons\ z\ (append\ zs\ y)]\!] \\
& \quad (\text{ABS} : (\text{STR} : \text{ABS}))\ E \\
= & \text{let } Cons\ z\ zs = x \text{ in} \qquad \text{(by (10)(12))} \\
& \mathcal{G}[\![Cons\ z\ (append\ zs\ y)]\!] \\
& \quad (\text{ABS} : (\text{STR} : \text{ABS}))\ E \cup \{z \mapsto a, zs \mapsto Cons\ b\ c\} \\
= & \text{let } Cons\ z\ zs = x \text{ in} \qquad \text{(by (4))} \\
& Cons\ \mathcal{G}[\![z]\!]\ \text{ABS}\ E \cup \{z \mapsto a, zs \mapsto Cons\ b\ c\} \\
& \quad \mathcal{G}[\![append\ zs\ y]\!] \\
& \quad (\text{STR} : \text{ABS})\ E \cup \{z \mapsto a, zs \mapsto Cons\ b\ c\} \\
= & \text{let } Cons\ z\ zs = x \text{ in} \qquad \text{(by (1)(9))} \\
& Cons\ \bot\ (append_1\ zs\ y)
\end{array}
$$

where $append_1$ is memoized with
call pattern : $append\ zs\ y$,
context : (STR:ABS), and
information : $E'' = \{\ x \mapsto Cons\ a\ (Cons\ b\ c),\ z \mapsto a, zs \mapsto Cons\ b\ c\ \}$

$$
\begin{array}{ll}
& append_1\ zs\ y \\
\stackrel{\text{def}}{=} & \mathcal{G}[\![\text{case } zs \text{ of} \\
& \quad Nil \quad : \ y \\
& \quad Cons\ u\ us \ : \ Cons\ u\ (append\ us\ y)]\!] \\
& \quad (\text{STR} : \text{ABS})\ E'' \\
= & \text{let } Cons\ u\ us = zs \text{ in} \qquad \text{(by (10)(12))} \\
& \mathcal{G}[\![Cons\ u\ (append\ us\ y)]\!] \\
& \quad (\text{STR} : \text{ABS})\ E'' \cup \{u \mapsto b, us \mapsto c\} \\
= & \text{let } Cons\ u\ us = zs \text{ in} \qquad \text{(by (4))} \\
& Cons\ \mathcal{G}[\![u]\!]\ \text{STR}\ E'' \cup \{u \mapsto b, us \mapsto c\} \\
& \quad \mathcal{G}[\![append\ us\ y]\!]\ \text{ABS}\ E'' \cup \{u \mapsto b, us \mapsto c\} \\
= & \text{let } Cons\ u\ us = zs \text{ in } Cons\ u\ \bot \qquad \text{(by (3)(1))}
\end{array}
$$

Summing up these transformation, we get the result:

$$\mathcal{G}[\![second\ (squares\ (append\ x\ y))]\!]\ \text{STR}\ \{x \mapsto Cons\ a\ (Cons\ b\ c)\} = second_0\ x\ y$$

where

$$
\begin{array}{ll}
second_0\ x\ y & = hd\ (tail\ (squares_0\ x\ y)) \\
squares_0\ x\ y & = \text{case } (append_0\ x\ y) \text{ of} \\
& \quad Cons\ z\ zs \ : \ Cons\ \bot\ (squares_1\ zs) \\
squares_1\ zs & = \text{case } zs \text{ of } Cons\ u\ us : Cons\ (u * u)\ \bot \\
append_0\ x\ y & = \text{let } Cons\ z\ zs = x \text{ in} \\
& \quad Cons\ \bot\ (append_1\ zs\ y) \\
append_1\ zs\ y & = \text{let } Cons\ u\ us = zs \text{ in } Cons\ u\ \bot
\end{array}
$$

6 Conclusion

In this paper we formalize a GPC transformation method for a lazy first-order functional language. GPC provides the natural setting to utilize additional information such as logical structure of programs, axioms for abstract data types, algebraic properties of primitive functions, etc. which have been rarely used in conventional partial evaluation.

The given two methods do not always terminate but more powerful than conventional partial evaluation techniques. The first method $\mathcal{G}_0$, which is applicable to both strict and lazy first-order languages, employs the memoization technique for proper treatment of recursion. The second method $\mathcal{G}$, which is specific for a lazy language, extends the first method utilizing the context information represented by projections. The result of backward strictness analysis is used to eliminate redundant transformation effectively, which helps the method terminate more often.

References

[1] A. Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. In *ESOP '90*, 70-87, LNCS 432, Springer-Verlag, 1990.

[2] G.L. Burn. A relationship between abstract interpretation and projection analysis(extended abstract). In *17th ACM Symposium on Principles of Programming Languages*, January 1990.

[3] K. Davis and P. Wadler. Backward Strictness Analysis: Proved and Improved. In *Proceedings of Glasgow Workshop on Functional Programming*, 12-30, Springer-Verlag, 1990.

[4] Y. Futamura and K. Nogi. Generalized Partial Computation. In D. Bjørner, A. P. Ershov and N. D. Jones, eds., *Partial Evaluation and Mixed Computation*, 133-151, North-Holland, 1988.

[5] Y. Futamura, K. Nogi, A. Takano. Essence of Generalized Partial Computation. In D. Bjørner and V. Kotov, eds., *Images of Programming*, North-Holland, 1991(to appear).

[6] N.D. Jones, P. Sestoft and H. Søndergaard. MIX: An Self-applicable Partial Evaluator for Experiments in Compiler Generation. *LISP and Symbolic Computation*, **2** (1) : 9-50, 1989.

[7] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In D. Bjørner, A. P. Ershov and N. D. Jones, eds., *Partial Evaluation and Mixed Computation*, 485-506, North-Holland, 1988.

[8] P. Wadler and R.M.J. Hughes. Projections for Strictness Analysis. In *Functional Programming Languages and Computer Architectures*, 385-407, LNCS 274, Springer-Verlag, 1987.

[9] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, **73** : 231-248, 1990.