ソフトウェア基礎論　30−5
（1989. 9. 8）

Computation Path Analysis with Path Valid Condition

Generalized Approach for Strictness Analysis on Non-flat Domains

経路有効条件を持つ計算経路解析

ノンフラットドメイン上のストリクトネス解析の一般化

Satoshi ONO, Mizuhito OGAWA and Yukio TSURUOKA
NTT Software Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo 180 Japan
ono%sonami-2.ntt.jp@relay.cs.net

小野　諭、小川　瑞史、鶴岡　行雄
NTTソフトウェア研究所
東京都武蔵野市緑町3-9-11
ono@sonami-2.ntt.jp

Atstract　A new global dataflow analysis for applicative languages named *Computation Path Analysis (CPA) with path valid conditions* is proposed. The *CPA* detects all possible demand propagation patterns from a result to parameters, and has superiority to strictness analysis in detecting divergence/redundancy of functions, and in optimizing demand propagation.

In the proposed analysis, path valid conditions are associated to each path so that the condition when a path is selected can be clarified. A simple framework for detecting and propagating path valid conditions is shown. It infers properties for paths only by fixpoint computation.

As for the application of path valid conditions, this paper also proposes the method named *conditional path absorption*, that enhances the analytical power of the *CPA* on non-flat domains.

あらまし　本稿では、関数型言語の新たな広域解析として、経路有効条件を持つ計算経路解析を提案する。計算経路解析は、関数適用において必ず評価される引数を検出するストリクトネス解析を一般化したもので、結果へのデマンドを引数側に波及させるパターンをすべて列挙するものである。この解析は、関数の冗長性や発散性の検出、要求駆動型計算の最適化などの応用において、ストリクトネス解析より優れた能力を持っている。

経路有効条件とは、可能な計算経路について、その経路が選択される時には必ず満足されている条件のことである。本稿では、停止性を持つ不動点計算により、経路有効条件の検出および関数間に渡る波及を行う方法を述べる。また、経路有効条件に基づき異なる計算経路を融合する条件付き経路吸収法を提案し、この方法によりノンフラット・ドメインにおける計算経路解析の解析力を向上できることを示す。

1 Introduction

In order to implement lazy functional programming languages efficiently, *Strictness Analysis* has been developed. A function $f(x_1, ..., x_n)$ is said to be *strict* in parameter x_i, if its result is undefined whenever x_i is undefined. Strictness analysis detects strict parameters of recursive functions. The actual parameter for a strict formal parameter can be passed on in *call-by-value* instead of in *call-by-name*. Optimization based on the analysis is more effective, when parallel evaluation is considered [4, 8].

There exist two methods for formalizing strictness analysis: *abstract interpretation*, and *program transformation*. The abstract interpretation is based on the continuous mapping from the original domain to the finite domain. The abstract functions are induced from this abstraction [2]. In contrast, in the program transformation approach, functions on the original domain are directly mapped to the functions on algebra, e.g. Boole lattice [6, 5]. Intuitively speaking, parameters evaluated in a computation path are joined by $\cup$ operation, and the strict parameters are computed as the intersection ($\cap$) of the sets over all possible computation paths. For extending strictness analysis to higher-order functions, the program transformation approach is more attractive since the function computing strictness information can be transformed by usual fold/unfold and simplification mechanisms.

Independent of strictness analyses, part of authors have proposed an analysis named *Computation Path Analysis (CPA)* for obtaining information required for controlling partial computation[7]. It has been clarified that the *CPA* can be formalized by the program transformation approach[9]. The algebra used, however, is not Boole lattice but a weaker algebra that has no *absorption laws*, namely, rules such as $x \cup (x \cap y) = x$ and $x \cap (x \cup y) = x$. The advantages of adopting the weaker algebra are summarized as follows[9] :

- Property of *completely-undefined functions* can be induced correctly. This feature is useful for a language that accepts user annotation on computation strategy (i.e. *strict-cons*), since misplacement of this kind of annotation will generally make some functions completely-undefined (infinite loop).
- In addition to strict parameters, *relevant parameters* (parameters that *may* be evaluated) can be computed simultaneously. This feature is useful when a language has partial computation feature, that is, giving only a subset of parameters, and specializing the program. In general, specialized program has redundant parameters, related computation for which should be removed at compile time.
- Strictness analysis makes data-driven computation of parameters possible when the result is demanded. In addition, the *CPA* makes data-driven computation possible when a parameter is demanded. It is because a demand can be regarded as a selection signal of a subset of possible paths. Thus, it provides more opportunity for demand propagation optimization[8].

In spite of these advantages, the analytical power of the *CPA* becomes quite inadequate when the domain is extended to *non-flat* (a domain with partially-defined data structures). In fact, the *CPA* cannot detect important information such as *head strictness*[1] and *tail strictness*[2] [11] even for simple functions. To give an example, consider a function $length(x)$ which computes a length of a list x. This function is apparently tail-strict. When x happens to be *nil*, the result of $length(x)$ in *Head Normal Form (HNF)*[3] can be obtained by only evaluating x in *HNF*.

The *CPA* on non-flat domains computes all possible least demand propagation patters from a result to each parameter. Therefore, results of the *CPA* contain a statement that can be interpreted as "there exists a path where the result of $length(x)$ in *HNF* can be obtained by evaluating x to *HNF*." This statement is inconvenient from optimization viewpoint, since when the result of $length(x)$ is demanded, x will be evaluated only in *HNF* prior to the function application, although x can be safely evaluated into *Spine Form*[4].

In this paper, we propose a new enrichment for the *CPA* named *path valid condition*, and also proposes its application named *conditional path absorption*, that enhances analytical power of the *CPA* over non-flat domains.

These proposals are based on the following observations:

- At the computation on the original domain, only one computation path is selected. Thus, for each path on the abstract domain, some predicates should exist that specify on what conditions each path is selected. We call them path valid conditions.
- Some of the path valid conditions can be derived from program texts. For example, consider the conditional function $if(x, y, z)$. Either the *then* part or the *else* part is selected according to x. If a demand is propagated to y, then x should be *non-nil*. Similarly, when a demand is propagated to z, then x should be *nil*.

[1] A function $f(x)$ is said to be *head strict* if $x = (x_1\ x_2\ ...\ x_{k-1}\ \omega\ x_{k+1}\ ...)$ then $f(x) = f(x')$ where $x' = (x_1\ x_2\ ...\ x_{k-1}\ .\ \omega)$, where ω stands for an *undefined value*.

[2] A function $f(x)$ is said to be *tail strict* if $x = (x_1\ x_2\ ...\ x_k\ .\ \omega)$ then $f(x) = \omega$.

[3] An expression e is said to be in *Head Normal Form* if there exist no top-level redexes in e. For more precise definition, please refer [10].

[4] A list e is said to be in *Spine Form* if e does not have an undefined tail after finite number of elements.

Thus, the path valid condition for x can be derived for each path of $if(x,y,z)$.

- It is sometimes possible that demand propagation patterns for parameters in a path can be unified to another path's patterns by taking the path valid conditions for them into account. For example, suppose there exist two paths stating that "x can be safely evaluated into *HNF*" and "x can be safely evaluated into *Spine Form*." Suppose also the path valid condition for the former be "x is *nil*." Then, the former path can be unified into the latter, since evaluating *nil* until *Spine Form* is equivalent to evaluating it until *HNF*. We name this unification the conditional path absorption, since a path on the abstract domain is absorbed into another path under the control of path valid conditions.[5]

We will first describe the similarity and difference between *CPA* and strictness analysis on flat domains. Next, the extension method of *CPA* on non-flat domains are explained. Then, a simple framework for detecting and propagating path valid conditions is shown that infers properties only by fixpoint computation. Finally, as the application of path valid conditions, the analytical power of the *CPA* with conditional path absorption is discussed.

2 CPA on flat domains

In the following, the *CPA* is defined using the program transformation approach. Thus, the analysis is defined by specifying mapping of functions and the axioms of the algebra by which transformed programs are substituted and simplified. Let D be a functional that transforms a function on the original domain to a function of the *CPA*.

[Function mapping for the *CPA*]

a-p1) Strict functions

$$D(\lambda(x_1,..,x_n).\, f_{strict}(x_1,..,x_n)) = \lambda(x_1,..,x_n).\, x_1 * ... * x_n$$

a-p2) Conditional functions

$$D(\lambda(x,y,z).\, \text{if } x \text{ then } y \text{ else } z) = \lambda(x,y,z).\, x*y \;+\; x*z$$

a-p3) Serial-or with 3 args.

$$D(\lambda(x,y,z).\, sor\_3(x,y,z)) = \lambda(x,y,z).\, x \;+\; x*y \;+\; x*y*z$$

a-p4) Completely-undefined function

$$D(\lambda(x_1,..,x_n).\, omega(x_1,..,x_n)) \;=\; \lambda(x_1,..,x_n).\, 0'$$

a-p5) Constant function

$$D(\lambda(x_1,..,x_n).\, const\_n(x_1,..,x_n)) \;=\; \lambda(x_1,..,x_n).\, 1'$$

a-t1) Function composition

$$\begin{aligned} &D(\lambda(x_1,..,x_n).\, f(e_1(x_1,..,x_n)\ldots e_m(x_1,..,x_n))) \\ &\quad = D(\lambda(y_1,..,y_m).\, f(y_1,..,y_m)) \\ &\qquad D(\lambda(x_1,..,x_n).\, e_1(x_1,..,x_n))\ldots \\ &\qquad\quad D(\lambda(x_1,..,x_n).\, e_m(x_1,..,x_n)) \end{aligned}$$

a-t2) Distributivity of λ over $*$

$$\lambda(x_1,..,x_n).\,(exp_1 * \ldots * exp_m) = \lambda(x_1,..,x_n).\, exp_1 * \ldots * \lambda(x_1,..,x_n).\, exp_m$$

a-t3) Distributivity of λ over $+$

$$\lambda(x_1,..,x_n).\,(exp_1 + \ldots + exp_m) = \lambda(x_1,..,x_n).\, exp_1 + \ldots + \lambda(x_1,..,x_n).\, exp_m$$

[Simplification rules for CPA]

a-s1)	$x*x = x$	$x+x = x$
a-s2)	$x*y = y*x$	$x+y = y+x$
a-s3)	$(x*y)*z = x*(y*z)$	$(x+y)+z = x+(y+z)$
a-s4)	$x*(y+z) = x*y \;+\; x*z$	
a-s5)	$0'*x = 0'$	$0'+x = x$
a-s6)	$1'*x = x$	

where rules a-s1) - a-s6) above correspond to idempotent law, commutative law, associative law, distributive law, zero element law and unit element law, respectively.

Using above rules iteratively, the result of the *CPA* is computed. For example, suppose

```
f(x,y,z)=if x>1 then x-1 else z+2.
```

Then, $D(\lambda(x,y,z).\,$`f(x,y,z)`$)$ is computed as follows:

$D($`f(x,y,z)`$)$
= $D($`if x>1 then x-y else z+2`$)$
= $\{\, a*b + a*c$
  where $a = D($`x>1`$);\ b = D($`x-y`$);\ c = D($`z+2`$)\,\}$
= $\{\, a*b + a*c$
  where $a = x*1';\ b = x*y;\ c = z*1'\,\}$
= $\{\, a*b + a*c$ where $a = x;\ b = x*y;\ c = z\}$
= $x*(x*y) \;+\; x*z$
= $x*y \;+\; x*z$

where all the surrounding λ-abstractions are discarded for simplicity.

Mapping for recursive functions can be computed using *Kleene's ascending chain*. The initial approximation for each function is 0'. For example, suppose

```
fact(n) = if n<2 then 1 else n*fact(n-1).
```

Then,

$$\begin{cases} fact'_i(n) &= 0' & (\text{if } i = 0) \\ fact'_i(n) &= (n*1')*1' \;+\; n*fact'_{i-1}(n*1') & \\ &= n \;+\; n*fact'_{i-1}(n) & (\text{if } i > 0) \end{cases}$$

[5] Obviously, absorption laws of Boole lattice can be named as *unconditional path absorption*.

where $fact'_i(n)$ $(i = 1, \ldots)$ stands for the i-th approximation to the $D(\lambda(n).\texttt{fact(n)})$. Then,

$$
\begin{aligned}
fact'_0(n) &= 0' \\
fact'_1(n) &= n + n * 0' = n \\
fact'_2(n) &= n + n * n = n \qquad \text{(converged)}
\end{aligned}
$$

Thus, $D(\lambda(n).\texttt{fact(n)})$ becomes $\lambda(n).n$.

The strictness analysis of Hudak and Young[5] can be induced by introducing the absorption law for + and by changing abstraction for the completely-undefined function as follows:

[The Strictness Analysis of Hudak and Young
(difference from the *CPA*)]
(Remove all axioms concerning $0'$)

a-p4') Completely-undefined function
$D(\lambda(x_1,..,x_n).\, omega(x_1,..,x_n))$
$= \lambda(x_1,..,x_n).\, x_1 * \ldots * x_n$

a-s7') Absorption law for +
$x * y + y = y \qquad 1' + x = 1'$

The change of the mapping for *omega* is required for keeping the safeness of the analysis. Thus, this analysis cannot distinguish completely-undefined functions from strict functions. For example, suppose

```
diverge(x,y)=if x>y then diverge(x-1,y)
                    else diverge(x,y-1)} .
```

Then, $D(\lambda(x,y).\texttt{diverge(x,y)})$ becomes $\lambda(x,y).0'$ by the *CPA* whereas becomes $\lambda(x,y).x * y$ by the strictness analysis.

3 CPA on the non-flat list domain

3.1 Domain abstraction

On non-flat domains, strictness informations are represented by a demand propagation mode from a result to each parameter. For example, suppose a non-flat list domain with *cons*, *car* and *cdr* be abstracted as Fig.1. This simple abstraction named $A_{spine-2}$ has 7 elements {**0**,...,**6**}, and it detects evaluation process to *spine* direction, as well as *HNF* evaluation of the first two elements of a list.

This abstraction is induced from the abstraction $A_{spine-\infty}$ shown in Fig.2(a). $A_{spine-\infty}$ preserves only top-level list structures, and ignores all sub-structures under them. The abstraction $A_{spine-n}$ (n=1,...) can be induced from $A_{spine-\infty}$ by regarding lists of *length* more than n as infinite lists. Thus, $A_{spine-m}$ can be induced from $A_{spine-n}$ when $m \leq n$. These relationships are shown in Fig.2 (b) and (c) using $A_{spine-3}$ and $A_{spine-1}$.

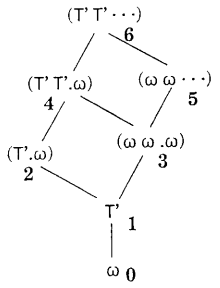
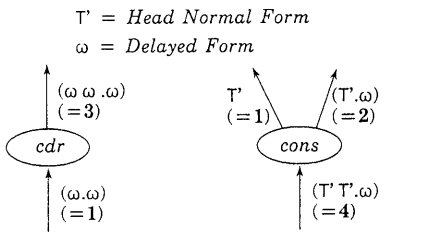


Figure 1: $A_{spine-2}$ Structure



(a) Demand Propagation for *cdr* (b) Demand Propagation for *cons*

Figure 3: Examples of Demand Propagation on $A_{spine-2}$

3.2 Demand propagation on abstract domains

The demand propagation mode is described based on this abstraction. For example, as shown in Fig.3 (a), if the *cdr* of the result is demanded in **1** form, the parameter should be evaluated at least in **3** form. For functions having more than one parameter, propagation mode is checked for each parameter, as in Fig.3 (b).

There exists a problem when finite lists and infinite lists are abstracted to the same element, since the properties of finite lists and infinite lists are different.

We will reach finite lists with any length by *cons*ing an element to *nil* iteratively, but will never reach infinite lists by such operations. Reversely, for any finite list, we will reach *nil* by getting *cdr* of it iteratively. For infinite list, however, we will never reach *nil* by such operations.

For modeling above-mentioned properties of infinite lists in finite (abstract) domains, there exist two models, *warp-down model* and *warp-up model*. In the warp-down model (Fig.4(a)), *cdr* of an infinite list becomes finite list in the abstract domain, whereas *cons*ing an element to a finite list becomes always finite list. To the contrary, in the warp-up model (Fig.4(b)), *cdr* of an infinite list remains an infinite

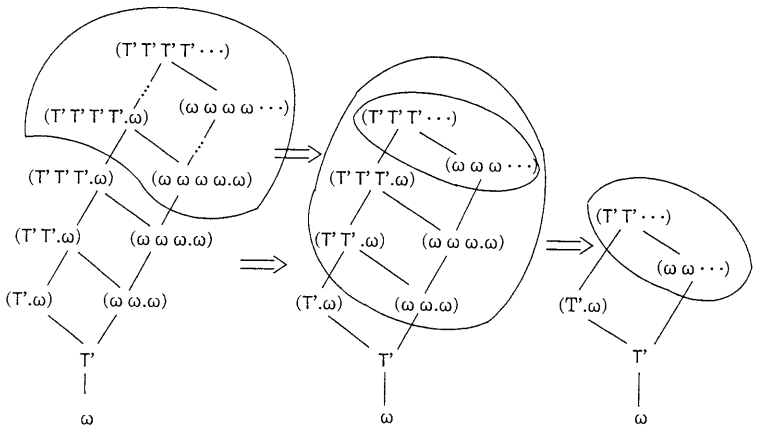


Figure 2: Successive Abstraction on Abstract Domains

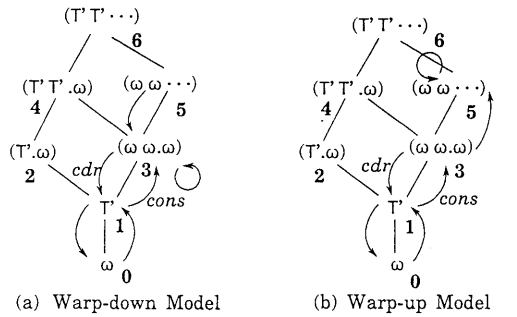


Figure 4: Ascending/Descending Transitions around Limit-point of $A_{spine-2}$

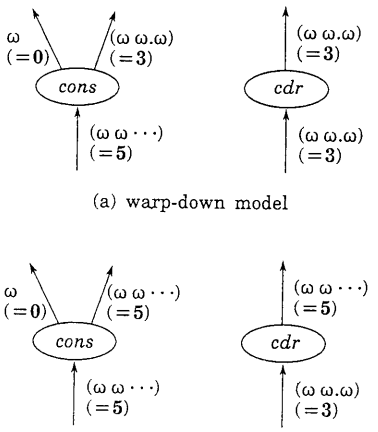


Figure 5: Transition between a Finite List and a Infinite List

list, and *cons*ing an element to some finite list results in an infinite list.

The CPA analyzes functions from result to arguments (backward analysis). Therefore, propagation from the result to the second parameter of *cons*, works as the *cdr* function, and propagation from the result to the parameter of *cdr* works as the *cons* function. Thus, the propagation patterns are summarized in Fig.5, where $A_{spine-2}$ is used as the abstraction.

In the warp-down model (Fig.5(a)), *cdr* of an infinite list (demand pattern **5**) becomes the longest finite list in the abstract domain (demand pattern **3**). Therefore, the demand pattern **5** to the result of *cons* will be propagated to the second parameter as the demand pattern **3**. Similarly, the demand pattern **3** to the result of *cdr* will be propagated to the parameter of it as the demand pattern **3**, since *cons*ing an element to a finite list becomes always a finite list.

In the warp-up model (Fig.5(b)), *cdr* of an infinite list remains an infinite list. Thus, the demand pattern **5** to the result of *cons* will be propagated to the second parameter as the demand pattern **5**. Similarly, the demand pattern **3** to the result of *cdr* will be propagated to the parameter of it as the demand pattern **5**.

The warp-down model is a *safe* approximation of the actual demand propagation, that is, the analyzed demand propagation pattern does not induce unnecessary computation. The warp-up model, however, does not always give

Table 1: Demand Propagation for primitives

(Abstract domain is $A_{spine\text{-}2}$)

symbol for mods	Demand for Result 0 1 2 3 4 5 6	Example
:ev-0	0 0 0 0 0 0 0	*delay*
:ev-1	0 1 1 1 1 1 1	*atom, null, x* of *if(x, y, z)*
:ev-2	0 2 2 2 2 2 2	*car* (or, *head*)
:cdr	0 3 3 5 5 5 5	*cdr* (or, *tail*)
:cons 1	0 0 1 0 1 0 1	*x* of *cons(x, y)*
:cons 2	0 0 0 1 2 5 6	*y* of *cons(x, y)*
:ident	0 1 2 3 4 5 6	*identity, y* and *z* of *if(x, y, z)*
:ev-3	0 3 3 3 3 3 3	
:ev-4	0 4 4 4 4 4 4	
:ev-5	0 5 5 5 5 5 5	*spine* evaluation
:ev-6	0 6 6 6 6 6 6	*head* and *tail strict* function

safe results, since this model possibly generates demands for evaluating infinite lists, where only evaluating finite elements is safe. For example, consider the function `cddr(x) = cdr(cdr(x))`. When it is analyzed with the warp-up model on the abstraction $A_{spine-2}$, the demand pattern **1** to the result of *cddr* will be propagated to the parameter as the pattern **5**. This result is not safe since the top-level structure after the third element need not be evaluated.

The warp-up model becomes a safe approximation when the abstract domain is sufficiently rich so that the irregular behaviors of the demand propagations caused by non-recursive part of programs can be handled by the finite elements of the abstract domains. In other words, complex finite structures that will be mapped to the abstract element where infinite structures are mapped, will be created only in corporation of the recursive part of the programs. Thus, the limit point of the finite structures becomes an infinite structure.

Table 1 shows the demand propagation modes for primitives. with the warp-up model on the abstraction $A_{spine-2}$. For example, y of $cons(x, y)$ has a demand propagation mode functions named `:cons2` that maps the demand **0,1,2,3,4,5** and **6** to **0,0,0,1,2,5** and **6** respectively.

3.3 Extending the *CPA* over non-flat list domain

Using these mode functions, the *CPA* over the non-flat list domain is formalized as follows:

[Function abstractions and axioms for the *CPA* on the non-flat list domain]

b-p1) Strict functions

$$D(\lambda(x_1,..,x_n).f_{strict}(x_1,..,x_n)) = \lambda(x_1,..,x_n).x_1^{:\text{ev-6}} * ... * x_n^{:\text{ev-6}}$$

b-p2) Conditional functions

$$D(\lambda(x,y,z).\text{if } x \text{ then } y \text{ else } z) = \lambda(x,y,z).x^{:\text{ev-1}} * y^{:\text{ident}} + x^{:\text{ev-1}} * z^{:\text{ident}}$$

b-p3) Serial-or with 3 args.

$$D(\lambda(x,y,z).sor\_3(x,y,z)) = \lambda(x,y,z).x^{:\text{ident}} + x^{:\text{ev-1}} * y^{:\text{ident}} + x^{:\text{ev-1}} * y^{:\text{ev-1}} * z^{:\text{ident}}$$

b-p4) Completely-undefined function

$$D(\lambda(x_1,..,x_n).omega(x_1,..,x_n)) = \lambda(x_1,..,x_n).0'$$

b-p5) Constant function

$$D(\lambda(x_1,..,x_n).const\_n(x_1,..,x_n)) = \lambda(x_1,..,x_n).1'$$

b-p6) *cons* function

$$D(\lambda(x,y).cons(x,y)) = \lambda(x,y).x^{:\text{cons1}} * y^{:\text{cons2}}$$

b-p7) *car* function

$$D(\lambda(x).car(x)) = \lambda(x).x^{:\text{car}}$$

b-p8) *cdr* function

$$D(\lambda(x).cdr(x)) = \lambda(x).x^{:\text{cdr}}$$

b-p9) *null* function

$$D(\lambda(x).null(x)) = \lambda(x).x^{:\text{ev-1}}$$

b-t1) Function composition (inverse direction)

$$(\ x^{\text{mode-x}}\ )^{\text{mode-y}} = x^{\text{mode-x} \circ \text{mode-y}}$$

b-t2) Distributivity of λ over $*$

$$\lambda(x_1,..,x_n).(exp_1 * ... * exp_m) = \lambda(x_1,..,x_n).exp_1 * ... * \lambda(x_1,..,x_n).exp_m$$

b-t3) Distributivity of λ over $+$

$$\lambda(x_1,..,x_n).(exp_1 + ... + exp_m) = \lambda(x_1,..,x_n).exp_1 + ... + \lambda(x_1,..,x_n).exp_m$$

b-t4) Distributivity of modes over $*$

$$(x_1^{\text{mode-1}} * ... * x_n^{\text{mode-n}})^{\text{mode-y}} = (\ x_1^{\text{mode-1}}\ )^{\text{mode-y}} * ... * (\ x_n^{\text{mode-n}}\ )^{\text{mode-y}}$$

b-t5) Distributivity of modes over $+$

$$(x_1^{\text{mode-1}} + ... + x_n^{\text{mode-n}})^{\text{mode-y}} = (\ x_1^{\text{mode-1}}\ )^{\text{mode-y}} + ... + (\ x_n^{\text{mode-n}}\ )^{\text{mode-y}}$$

In the extended *CPA*, each variable is associated with the mode such as $x^{:\text{ev-1}}$. The mode specifiers such as `:ev-1` are a function defined in Table 1 that map a demand pattern of the result to a demand pattern of the parameter. The term $x^{:\text{ev-1}}$ means that when the demand pattern for the result is **n**, then the demand pattern for the parameter x is `:ev-1(n)`.

Note that as show in the rule b-t1), the composition of modes are inverse direction, namely, $(\ x^{\text{mode-x}}\ )^{\text{mode-y}}$ is not $x^{\text{mode-y} \circ \text{mode-x}}$ but $x^{\text{mode-x} \circ \text{mode-y}}$. It is because the *CPA*is a backward analysis.

4 Path Valid Condition and Conditional Path Absorption

In general, the assertion for path valid conditions are generated in different functions. Thus, informations detected should be propagated interfunctionally. For this purpose, two properties are introduced: *value property* and *function property*. The value property is a predicate on the original domain data, such as "x is *nil*" and "*cdr* of x is *nil*." Function property transforms value properties bidirectionally, namely from parameter to result and *vice versa*. Table 2 (a) and (b) show the rules for the transformation. For example, `:get-cdr` function property transforms `:cdr-is-nil` value property to `:is-nil` for forward direction, and to `:cddr-is-nil` for backward direction. The property `:omega` stands for unsatisfiability of path valid conditions, that is, the path is a dead path. Such a path is removed from answers.

When two or more value properties exist for same variable on a path, possible conflicts among them are resolved by *confliction resolution rules*, as shown in Table 2 (c). In addition, function properties are composed as usual way. Table 2 (d) shows the example of *function property composition rules*. For example, composition of `:make-cons-1` and `:get-cdr` results in `:no-ref`, that means the *car* part of a *cons* is discarded when *cdr* is taken.

In order to keep properties finite and make the analysis terminating, these inferences rules should be *bounded*. For example, the composition of `:get-cdr` and `:get-cdr` may results in the property not `:get-cddr` but `:void`, that means "nothing can be inferred."

Fig. 6 (a) shows the results of *length*(x) when analyzed by the *CPA* with path valid condition. The result clarifies that there are three computation paths for *length*(x), each evaluates x in **1**, **3** and **5**, respectively. The processes how path valid conditions are induced are shown in Fig.6 (b) and (c), each corresponding to ((*x* `:is-nil :ev-1`)) and ((*x* `:cdr-is-nil :ev-3`)).

The greatest demand mode common to all paths in the above example is `:ev-1`. When the conditional path absorption is done using path valid conditions, the result becomes ((*x*`:ev-5`)) that means *length*(x) is tail strict in x. Note that, in general, the conditional path absorption is performed, not only at the time final results are obtained, but continuously while analysis. It is because there may exist *transient* properties that are discarded by the limitation of bounded inference rules.

Table 2: Inference Rules for value/function Properties

(a) Inference Rules for $p_f(p_v)$ (From Parameter to Result)

p_f \ p_v	:is-nil	:is-non-nil	:cdr-is-nil
:get-cdr	:omega	-	:is-nil
:make-cons-1	-	:is-non-nil	:is-non-nil
:make-cons-2	:cdr-is-nil	:is-non-nil	:cddr-is-nil

(b) Inference Rules for $p_f^{-1}(p_v)$ (From Result to Parameter)

p_f \ p_v	:is-nil	:is-non-nil	:cdr-is-nil
:get-cdr	:cdr-is-nil	:is-non-nil	:cddr-is-nil
:make-cons-1	:omega	-	-
:make-cons-2	:omega	-	-

(c) Confliction Resolution Rules for p_v

p_{v1} \ p_{v2}	:is-nil	:is-non-nil	:cdr-is-nil
:is-nil	:is-nil	:omega	:omega
:is-non-nil	:omega	:is-non-nil	:cdr-is-nil
:cdr-is-nil	:omega	:cdr-is-nil	:cdr-is-nil

(d)Inference Rules for $p_f \circ p_g$

p_f \ p_g	:get-car	:get-cdr	:make-cons-1	:make-cons-2
:get-car	:get-caar	:get-cadr	:ident	:no-ref
:get-cdr	:get-cdar	:get-cddr	:no-ref	:ident
:make-cons-1	:car-is-eq	-	-	-
:make-cons-2	-	:cdr-is-eq	-	-

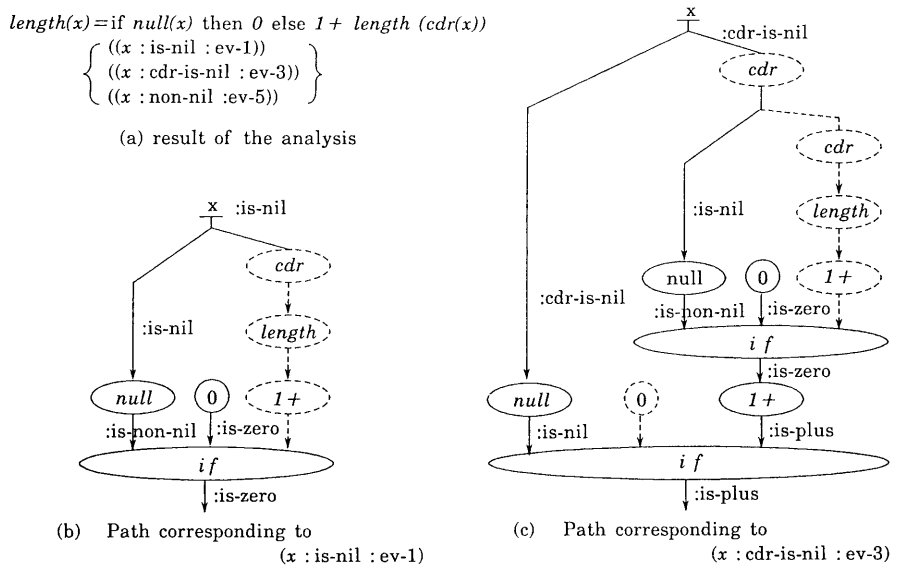


Figure 6: Example of the *CPA* with Path Valid Condition

5 Evaluation of the Analyzer

We have built a table-driven *CPA* analyzer that accepts tables specifying abstract domains, modes for primitives and value/function properties.

Following tables have been built almost manually, and fed to the analyzer.

Abstract domain : $A_{spine-2}$
Modes for primitives : (See Table 1)
Value properties :
{ `:is-nil, :cdr-is-nil, :is-non-nil` }
Function properties :
{ `:get-cdr, :make-cons-1, :make-cons-2` }

These parameters are selected so that at least simple tail-strictness can be detected. For the test of the analyzer, following functions are used:

Results are summarized in Table 3. For functions over flat domains, function divergence, redundant parameters and path selection dependency on *if* are detected correctly. For functions over non-flat domains, almost all tail-strictness are detected. One exception is a tail-strictness of y in $append(x, y)$. The analyzer's result of $append(x, y)$ can be interpreted as follows:

- If the result is demanded in **5** , then x can be safely evaluated to *Spine Form*.
- If the result is demanded in **5** and y is demanded in *HNF*, then y can be safely evaluated to *Spine Form*.
- There may exist a path where the result can be computed without y.

The last statement is reasonable, since when x is an infinite list, y will be never demanded. In this case, however, the result becomes always undefined, and therefore, y can be safely evaluated anyway. Current inference system of the analyzer does not have such reasoning, and thus cannot detect tail-strictness of y in $append(x, y)$.

For head-strictness, detection is completely failed. It is because the abstract domain $A_{spine-2}$ is not sophisticated enough to detect head-strictness. By making more sophisticated abstract domains, it will be easy to detect head-strictness when accompanied with tail-strictness. Head-strictness without tail-strictness is difficult only by domain sophistication. We might require other reasoning mechanisms.

6 Relation to Other Work

Our approach associates each path the predicate when the path is selected. Information about the satisfiability can be computed not only at compile time, but at runtime. In this case, the demand propagation patterns can be computed at runtime, by checking the property of already evaluated values,

As long as static strictness detection power is concerned, our approach is behind advancing work of [11] and [1]. However, in our approach, corporation of abovementioned static and dynamic information is possible. In addition, our ap-

```
diverge(x,y)    = if x>y then diverge(x-1,y) else diverge(x,y-1)
easy(x,y)       = if x=0 then 0 else easy(x-1,easy(y,x))
sync_add(p.x,y)= (if p then x else y) + (if null(p) then x else y)
length(x)       = if null(x) then 0 else 1+length(cdr(x))
append(x,y)     = if null(x) then y else cons(car(x),append(cdr(x),y))
reverse(x)      = if null(x) then nil else append(reverse(cdr(x)),cons(car(x),nil))
len_of_appended_list(x) = length(append(x,y))
len_of_rev_rev(x)       = length(reverse(reverse(x)))
sum_of_list(x)          = if null(x) then 0 else car(x)+sum_of_list(cdr(x))
sum_of_appended_list(x) = length(append(x,y))
sum_of_rev_rev(x)       = sum_of_list(reverse(reverse(x)))
```

Figure 7: Definitions of tested functions

Table 3: Analytical Power of the non-flat *CPA* with Conditional Path Absorption

(Abstract domain is $A_{spine\text{-}2}$)

(a) functions over flat domains

function	features to be detected	result
diverge(x, y)	completely undefined	detected
easy(x, y)	y is redundant	detected
sync-add(p, x, y)	strict	detected

(b) functions over non-flat domains

function	features to be detected	result
length(x)	tail strict	detected
append(x, y)	tail strict in x when demanded in **5**	detected
	tail strict in y when demanded in **5**	detected when y is required
reverse(x)	tail strict in x when demanded in **5**	detected
rev-of-rev(x)	tail strict in x when demanded in **5**	detected
len-of-appended-list(x, y)	tail strict in x and y	detected for x
len-of-rev-rev(x)	tail strict	detected
sum-of-list(x)	head / tail strict	tail-strictness is detected
sum-of-appended-list(x, y)	head / tail strict in x and y	tail-strictness in x is detected
sum-of-rev-rev(x)	head / tail strict	tail-strictness is detected

proach will provide programmers more information such as divergence, redundancy and possible demand generation pattern after all strict parameters are evaluated and passed on to the function body.

Our work is also closely related to *Generalized Partial Computation (GPC)*[3]. The *GPC* also keeps track of the predicate when some path is selected. Our approach confine property inference system rather simple, so that the termination can be ensured, and the property can be computed not by general unification, but simple fixpoint computation.

7 Conclusion

A new global dataflow analysis named *Computation Path Analysis (CPA) with path valid conditions* have been proposed. The *CPA* detects all possible demand propagation patterns from a result to parameters, and has superiority compared with strictness analysis in detecting divergence, redundancy of functions, and in optimizing demand propagation.

In the proposed analysis, path valid conditions are associated to each path so that the condition when a path is selected can be clarified. A simple framework for detecting and propagating path valid conditions is shown that infers properties only by fixpoint computation. As the application of path valid conditions, we have also proposed the method named conditional path absorption, that enhances the analytical power of the *CPA* on non-flat domains.

Acknowledgements. The authors would like to thank Dr. Katsuji Tsukamoto, Director of NTT Software Research Laboratories, for his guidance and encouragement. They also wishes to thank Mr. Masaru Takesue and Dr. Naohisa Takahashi for their useful discussions and helpful comments.

References

[1] Burn,G.L., "Evaluation transformers - A model for the parallel evaluation of functional languages," *Functional Programming Languages and Computer Architecture*, *LNCS* **274**, Springer-Verlag, pp.446-470 (1987)

[2] Clack,C. and Peyton Jones,S.L., "Strictness analysis – a practical approach," *Functional Programming Languages and Computer Architecture*, *LNCS* **201**, Springer-Verlag, pp.35-49 (1985)

[3] Futamura,Y., "Program evaluation and generalized partial computation," *International Conference on Fifth Generation Computer Systems 1988*, ICOT, pp.685-692 (1988)

[4] Hankin,C.L., Burn,G.L., and Peyton Jones,S.L., "A safe approach to parallel combinator reduction," *European Symposium on Programming*, *LNCS* **213**, Springer-Verlag, pp.99-110 (1986)

[5] Hudak,P. and Young,R., "Higher-order strictness analysis in untyped lambda calculus," *13th ACM POPL*, pp.97-109 (1986)

[6] Maurer,D., "Strictness Computation Using Special λ-expressions," *Workshop on Programs as Data Objects*, *LNCS* **217**, Springer-Verlag, pp.136-155 (1985)

[7] Ono,S., Takahashi,N. and Amamiya,M., "Non-strict partial computation with a dataflow machine," *6th RIMS Symposium on mathematical methods in software science and engineering*, TR.**547**, *RIMS Kyoto Univ.*,pp.196-229 (1984)

[8] Ono,S., Takahashi,N. and Amamiya,M., "Optimized demand-driven evaluation of functional programs on a dataflow machine," *IEEE ICPP'86*, pp.421-428 (1986)

[9] Ono,S., "Computation path analysis : Towards an autonomous global dataflow analysis" *The Second France-Japan Artificial Intelligence and Computer Science Symposium, Sophia, France* (1987)

[10] Peyton Jones,S.L., "The implementation of functional programming languages," Prentice-Hall (1987)

[11] Wadler,P., and Hughes,R.J.M., "Projections for strictness analysis," *Functional Programming Languages and Computer Architecture*, *LNCS* **274**, Springer-Verlag, pp.385-407 (1987)