Implementation of a static validity checking algorithm for putback transformation in PDL

Thibaut Tachon

This long title won't be explained later. Hence, if not understood, I highly encourage you to read the paper [1], because this work leans on this article. Moreover, some part of this explanation are extracted from my internship report so please bear with the first singular person and the useless narrative style.

Pdl

PDL stand for *Putback Definition Language*. PDL is a *treeless* and constructor-based language for describing putback transformation. By treeless, I mean that no composition can be used.

Here is the syntax of PDL :

Rule Definition

r	$::= f\ p_s\ p_v \mathrel{\hat{=}} e$	Putback rule

Pattern

p	$::= \mathcal{C}\ p_1 \cdots p_n$	Constructed pattern
	$\mid x\ @\ p$	Look-ahead variable
	$\mid x$	Variable

Expression

e	$::= \mathcal{C}\ e_1 \cdots e_n$	Constructed expression
	$\mid x$	Variable
	$\mid f\ x_s\ x_v$	Function call

Rule A PDL transformation is written with a set of rules. If source is matched by p_s and view by p_v then rule is taken and e is produced.

Pattern may be a constructed pattern which means a constructor and each of its component are patterns or a variable. The purpose of a Look-ahead variable is to use a pattern as argument of a function call because, as you may observe, a function call take only variable as argument (treeless form).

Expression may be constructed expression or variable, just like patterns, but may also be a function call.

Part I
Implementation

Here, I will present the algorithm implemented and describe each important point. Figure 1 represent the original algorithm presented in [1]. During implementation, I had to modify some point that I will explain in due course.

```
Function:
    check syntactic constraint

    {* check totality: *}
    check pattern exhaustiveness;

    for each rule f p_s p_v ≙ e do

        {* check view determination: *}
        check injectivity: FV(p_v) ⊆ FV(e);
        derive and normalise R_f;
        check single-valuedness of R_f ;

        {* check source stability: *}
        define pr s v ≙ f s (R_f v);
        check property pr s s = s inductively;
    end for;
```

Figure 1: Original Validity Checking Algorithm

1 Syntactic Constraint on rules $f\ p_s\ p_v \mathrel{\hat{=}} e$

Body of the right-hand side of a rule (e) is required to be in *structured treeless form* [4]. This means that parameters of a function call must be variables, so nested call are not allowed.

$$f\ p_s\ p_v = f\ x_s\ x_v \qquad (x_s \in p_s,\ x_v \in p_v)$$

Moreover, those variables have to come from their respected pattern, so that there is no way an element of source will be treated as a view and vice versa.

Beside, at least one of x_s or x_v is required to be strictly smaller than its original pattern. This constraint purpose is to ensure termination.

Rules also need to be *affine* which means that each variable in the left-hand side of a rule occur at most once in the corresponding right side. I cannot explain the purpose of this constraint now, because I would have to talk about the derivation of get transformation but it only occur during view determination checking (treated at section 3.3)

2 Pattern exhaustiveness

Exhaustive pattern matching To ensure that the transformation works on any source and view (*totality checking*), the pattern matching need to be exhaustive. The paper [2] presenting a pattern exhaustiveness checking with an improvement that give an example of non-matching value when pattern is not exhaustive. This is the algorithm used in OCaml compiler. I implemented this one. Because algorithm is fully explained in paper and that I did not invent anything, I won't explain it.

However, an important precondition is typing which is something that [1] do not talk about at all, so I will talk about it a bit. In a compiler, this check occur after typing phase and in this way, a lot of checks won't be needed anymore.

Typing I had two possibilities at this time. the first one is asking user to precise type of functions and type from this information. The second one is to introduce polymorphism and deduce function type when unifying types. I chose the first one because I wanted to quickly finish a first stable version. and add polymorphism after if I had more time. Eventually, I did not but if someone continue this project, it would be a good improvement. Anyway, the user still has to describe constructor he is using by providing type of all components and type of the constructor himself.

Let's sum up what is checked during typing :

- For constructors
  - number of components
  - type of components
  - type of constructor
- For rules
  - type of source argument
  - type of view argument
  - type of output expression (same as source argument)
- Right hand side
  - variables exist in left-hand side and have same type
  - function call has same type as source argument

3 View Determination $\quad put\ s\ v = put\ s'\ v' \implies v = v'$

What is view determination? If you remember well, I said that a put transformation accept only one get transformation that, coupled with put, will be well-behaved ie satisfy GetPut and PutGet laws. This 2 law have been transformed in [1] in a way that the get function do not appear inside them. This is essential because get is not derived yet. This two new properties are called View Determination and Source Stability. It has been proved that the combination of them is equivalent to GetPut and PutGet.

GetPut	$put\ s\ (get\ s) = s$	$\iff$	Source Stability	$\forall s, \exists v.\ put\ s\ v = s$
PutGet	$get\ (put\ s\ v) = v$		View Determination	$put\ s\ v = put\ s'\ v' \implies v = v'$

View determination means that if two *put* have the same result then the view argument must be the same in both. As a reminder, put take a source, an updated view and reflect changes made to the view back to the source. So, put type is Put :: S → V → S. Let's see with schema what a good put and bad put function looks like. I represent the function in a curried way, so that it will be easier to show the reverse function.

Satisfy view determination Do not satisfy

Figure 2: Schema of view determination

In the right example, *put* do not satisfy view determination because two different applications that have the same result do not have the same view argument.

3.1 Injectivity of *put s*

$$(put\ s)\ v = (put\ s)\ v' \implies v = v'$$

Now, we need to prove View Determination. For this, put need to be injective. Intuitively, we can see that injectivity of *put s* looks like view determination a lot (only one prime differ). It was proved that for any source s, *put s* is injective if and only if $\mathcal{FV}(p_v)$[1] $\subseteq \mathcal{FV}(e)$ holds for every any rule $f\ p_s\ p_v \mathrel{\hat{=}} e$. This property is hopefully a lot easier to check. Let's take an example.

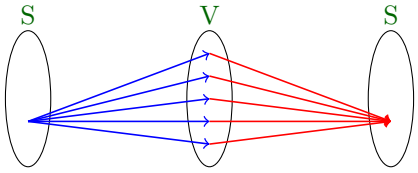
putNoInj (A s) v = A s ($v \notin$ A s)

This *put* takes a tagged value as source, a view and update only the tagged value. Because, the free variable v is not in the right-hand side of function, our definition is not respected.

$\forall v_1, v_2$:

putNoInj (A s) v_1
putNoInj (A s) v_2
} = A s

If we look at several application with a different view as argument but with same source, the result will always be the same.



If we use again a schema, we may see that view determination is absolutely not respected.

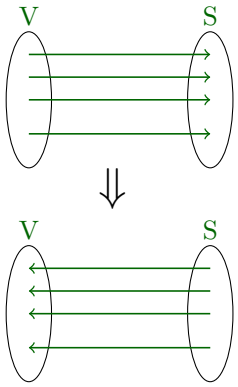
I showed that injectivity of *put s* is necessary for view determination but not that it imply this. And it does not. You will see the purpose of this check in the end of this section.

3.2 Single valuedness

We still need to check view determination, but this is not easy. We cannot just execute all possible application of put function to verify our property, this would be is infinite. However, let's look at the reverse transformation of *put* (strictly speaking, we talk about more about reverse transformation of *put s*, that is why we won't represent the source argument).

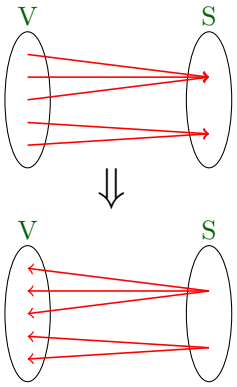
[1] set of free variables of a pattern

Satisfy view determination



single-valued

Do not satisfy view determination



not single-valued

I intuitively show that checking view determination of *put* is he same as checking the single-valuedness of the reverse transformation (that we will name R_{put}). In fact, a transformation written in PDL may be seen as tree-transducer and from [3], the single valuedness of a tree transducer is decidable in polynomial time. This is the good news. The bad news is the the paper is very dense, I had absolutely no idea what was a tree-transducer and the algorithm of this paper has never been implemented (according to the knowledge of a teacher). So I had to do it myself. Hopefully, after a fruitful discussion, my teacher told me that checking single valuedness was actually the same as checking confluence.

Why? a function that is not single-valued is a function that may output different values, in other words not deterministic. In our case, PDL is written as a rewriting system, with a set of rule and several rules may be applicable at same time (if we have overlapping pattern in our pattern matching). Confluence in a rewriting system means that, we may have several rules applicable at same time but they have to lead to the same result. Let's see a little example.

```
f []        -> False   (1)
f [B b]     -> False   (2)
f (B b):s   -> f s     (3)
f (A a):s   -> True    (4)
```

This example check if there is an A-value in a list. We may observe that if *f* apply to list with only one B-value, then the rule (2) and (3) may apply but both will at last return False

This rewriting system is convergent and so, single-valued. To check convergence, I used a tool called CSI. I chose this one because I could test it with a web interface, it is easy to use, easy to install with no additional tools needed. This might looks like publicity but there is a lot of confluence checking tools. Before this one, I tried Saigawa, ACP, TRS.tool and CSI is from far my favourite.

3.3 derive reverse transformation

We need to check confluence on the reverse transformation (R_{put}). For this, we have to derive reverse transformation. The key idea is just to switch the view pattern argument and the output expression, like this :

$$f\ p_s\ p_v \mathrel{\hat{=}} e \implies R_f\ e \mathrel{\hat{=}} p_v$$

Figure 3: reverse *put*

The only issue remains in function call in the expression that make no sense in left-hand side. A little variable change is needed to address it :

$$\text{Let } x'_s = f\ x_s\ x_v \implies x_v = R_f\ x'_s$$

Figure 4: remove function call in left-hand side

Let's clear things with an example :

```
putAs []          []          = []
putAs (ss @ [])   (v : vs)    = A v : putAs ss vs
putAs (A a : ss)  (vs @ [])   = putAs ss vs
putAs (A a : ss)  (v : vs)    = A v : putAs ss vs
putAs (B b : ss)  vs          = B b : putAs ss vs
```

This example update all values from view as A-value, forget A-values from source and update B-values like in this application :

```
putAs [ A 1, A 2, B 3, A 4, B 5]   [ 1, 3, 6]
    = [ A 1, A 3, B 3, A 6, B 5]
```

Then we reverse using our formula from figure 3

```
R_putAs []                    = []
R_putAs (A v : putAs ss vs)   = (v : vs)
R_putAs (A v : putAs ss vs)   = (v : vs)
R_putAs (B b : putAs ss vs)   = vs
```

Eventually, we can remove function call from left-hand side to put them on the other side, using variable change of 4. I removed one equation that was strictly identical to another.

```
R_putAs []          =   []
R_putAs A v : x'_s  =    v : R_putAs x'_s
R_putAs B b : x'_s  =    R_putAs x'_s
```

The final reversed function get A-values while removing constructor A and don't get B-values. We can finally check confluence of this function to see if *put* transformation satisfy view determination. If *put* also satisfy view determination, then R_f will become our *get*.

As I promised, I can finally explain why a variable of left-hand side can occur at most one in right-hand side. Let's see what happen if we don't respect this constraint.

$$f_1\ p_s\ p_v = (x_v, x_v) \qquad \Longrightarrow \qquad R_{f_1}\ (x_v, x_v) = p_v$$

If x_v appear in p_v, we won't know to which variable it refers. However, if the variable that is not linear comes from source,

$$f_1\ p_s\ p_v = (x_s, x_s) \qquad \Longrightarrow \qquad R_{f_1}\ (x_s, x_s) = p_v$$

There is no problem. That is why I think that this constraint should be relaxed and only apply to variables in view. Only this is already an extension of PDL.

There is also the problem of injectivity, why did why checked this? Because a non-injective rule cannot be reversed.

$$\text{putNoInj (A s) v} = \text{A s} \qquad \Longrightarrow \qquad R_{putNoInj}\ \text{A s} = \text{v}$$

In $R_{putNoInj}$, to what refer **v**?? Nothing. That is why practically, injectivity is needed.

4 Source Stability $\forall s, \exists v.\ put\ s\ v = s$

To satisfy Source stability, we need to find v. According to GetPut property ($put\ s\ (get\ s) = s$), this v is equal to $get\ s$. So, we need to check for every possible pattern in source p_s if

$$f\ p_s\ (R_f\ p_s) = p_s$$

Figure 5: check for source stability

This set of possible pattern p_s is potentially infinite. However, because we already checked totality, we know that all possible patterns are represented in pattern of put function. That is why, we only need to check our formula from figure 5 with every pattern used in source argument of *put* function. Let me show you how to check this property for one pattern : (A a : ss)

	putAs	(A a : ss)	R_{putAs} (A a : ss)	[R_{putAs} (A v : ss) = v : (R_{putAs} ss)]
=	putAs	(A a : ss)	(a : R_{putAs} ss)	[putAs (A a : ss) (v : vs) = A v : (putAs ss vs)]
=	A a :	putAs ss	(R_{putAs} ss)	[Inductive Hypothesis]
=	A a :	ss		

To sum up, we need to use rules of put function, rules of reversed function and an inductive hypothesis. Of course, we are only able to use an inductive hypothesis because we have a base case : [ ]

	putAs	[ ]	R_{putAs} [ ]	[R_{putAs} [ ] = [ ]]
=	putAs	[ ]	[ ]	[putAs [ ] [ ] = [ ]]
=	[ ]			no need for inductive hypothesis

Of course, this check has to be done automatically. This is the part that took me the biggest amount of time. First, I had to find a theorem prover. I used CITP (Constructor-based Inductive Theorem Prover) because Pdl is constructor-based and that I had to implement an automated inductive proof. This theorem prover works on *maude*, a language made to support rewriting logic specification. maude manual has about 500 pages, and CITP manual... does not exist, that's one reason why it took me so much time. Another reason is that CITP was not able to see more than one step ahead in a pattern. I mean that it could figure out that [ ] works as a base case for **h:s** but not for **A a : s**. To make CITP realise this, I had to make it prove :

putAs ss (R_{putAs} ss) = ss $\quad\Longrightarrow\quad$ putAs h : s (R_{putAs} h : s) = h : s

I use the implication to go one step further. Then I tell CITP to develop **h**. Because of type of **h** and constructors, CITP know that **h** may be develop as (A a) or (B a). Then, I use some *tactics* and implication is proved. I won't explain what are the tactics in details but I will just let you know the sequence of tactics is the same for all subcases of the variable, I just need to know how many subcases there are to know the good complete sequence.

After that, I just tell CITP to prove putAs ss (R_{putAs} ss) = ss. It will develop **ss** as [ ] or **h:s**. For the former, it is easy because it is a base case, for the latter, it can do the proof as I showed before because there is the implication and the base case.

Because a variable like **ss** is the parent of all possible pattern, source stability is proved.

Part II
Extend Pdl

5 An "else" problem

A rule is interpreted as : **if** p_s match the source and p_v match the view **then** e. But there is no way to say **else** e' and this is a big flaw for expressiveness. To demonstrate this and also take this opportunity to show how Pdl is used, I will show you an example. In this example, we try to get the number of "toto" in a list of name. First, we define constructors :

Source is a list of Name

```
noName : → NameList
consN  : String → NameList → NameList
```

A Character is either

```
a : → Char
⋮
z : → Char
```

A Name is a list of Character

```
sglS   : Char → String -- no empty string
consS  : Char → String → String
```

A Number is

```
zero   : → Number
succ   : Number → Number -- +1
```

The problem is the same and understanding get is easier than put, so I prefer to show you directly get (named R_put in PDL)

```
Rput noName  = zero

Rput consN(consS(t, consS(o, consS(t, sglS(o)))),s) = succ(Rput s)

Rput consN (?, s)  = Rput s -- ≠ of toto
```

The issue here is how to represent that a string is different of "toto"? If we do it naively, we will have 26^4 (26 is the number of possible letter and 4 the length of "toto") possibilities to enumerate. Of course, with longer word or if we consider for example all character of ASCII or UTF-8, this number explode.

5.1 An "else" solution

We cannot address the issue without changing something. Here, a good solution is to change constructors :

```
toto    : → Name
notToto : String → Name
```

Now, I can use the constant constructor toto to represent each string "toto" and notToto(s) to represent any other string.

Rather than showing you the updated get function with this trick which is quite trivial, I prefer to show you a put function that may be associated to this get. Of course, this example and all others I will show later are all successfully checked by the algorithm.

```
put   noName                  zero        = noName
put   consN(toto, ss)         succ(vs)    = consN(toto, put ss vs)
put   ss @ noName             succ(vs)    = consN(toto, put ss vs)
put   consN(toto, ss)         vs @ zero   = put ss vs
put   consN(notToto(n), ss)   vs          = consN(notToto(n), put ss vs)
```

This trick requires some preprocessing where all string are casted to toto and notToto. Moreover, this trick is only usable when we know to what we want to compare. If you want to compare source and view to each other, this is a whole other problem.

5.2 Compare source and view

5.2.1 Comparison

In fact the real problem do not lie in comparison. I want to write the name of view if they are the same and nothing if they are different :

```
cmp   ss                  noName        = noName
cmp   consN(a, ss)        consN(a, vs)  = consN(a, cmp ss vs)
  ⋮   26
cmp   consN(z, ss)        consN(z, vs)  = consN(z, cmp ss vs)
cmp   consN(a, ss)        consN(b, vs)  = noName
  ⋮   26^2 = 676
cmp   consN(z, ss)        consN(y, vs)  = noName
```

As you can see here, even if there is a lot of rules to write, about 700 rules are easily supported by computer. Still it is boring to write, so we may add some syntax to write them quickly with something like :

$\forall a, b \in Char, a \neq b$

```
cmp   consN(a, ss)        consN(b, vs)  = noName
```

5.2.2 To write or not to write?

As you may have seen, the function cmp is wrong. If a prefix of source name and view name is the same, then this prefix will be put before realising that the rest is different. We have to know whether we should put the operand of comparison or not before knowing the result of comparison. This is a big issue unresolvable without changing PDL a lot.

References

[1] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. Validity checking of putback transformations in bidirectional programming. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 1–15. Springer International Publishing, 2014.

[2] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421, 2007.

[3] Helmut Seidl. Single-valuedness of tree transducers is decidable in polynomial time. *Theoretical Computer Science*, 106:135–181, 1992.

[4] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin Heidelberg, 1988.