Inversion Laws for Specifications and Recursive Procedures

Wei Chen

In this paper, we continue the work on the formal approach to program inversion by presenting programming laws for specifying the inverse program according to the specification of its forward program, and for inverting recursive procedues. The formal establishment of these laws, once more, convinces us that program inversion has nice mathematical properties that can be used in formal program development. Some examples are included to illustrate the usage of the laws developed in this paper.

Inversion Laws for Specifications and Recursive Procedures

Wei Chen

WUCS-92-03

January 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Inversion Laws for Specifications and Recursive Procedures

Wei Chen

Department of Computer Science
Washington University
Campus Box 1045
St. Louis, MO 63130

January 2, 1992

Abstract

In this paper, we continue the work on the formal approach to program inversion by presenting programming laws for specifying the inverse program according to the specification of its forward program, and for inverting recursive procedures. The formal establishment of these laws, once more, convinces us that program inversion has nice mathematical properties that can be used in formal program development. Some examples are included to illustrate the usage of the laws developed in this paper.

1 Introduction

The concept of program inversion can be traced back to Dijkstra and Feijen [6]. Subsequently, it was somewhat explored by Gries in [7]. Since then, the concept has popped up every so often. In [4] and [2], a formal approach to program inversion was proposed and applied in developing some programs that are otherwise hard to prove. Later, Von Wright [13] incorporated this formal concept of program inversion into a general programming calculus called refinment calculus. In this paper we continue in this direction by formally establishing laws for specifying the inverse program and for inverting procedures.

In the next section, we introduce some notions and notations we will need in the rest of the paper. In section 3, we present laws to specify the inverse program. When applying program inversion in program construction, we aim at the inverse program. If we did not know what it would achieve, then the whole approach of program inversion to program construction would be questionable. To our best knowledge, most programs developed by program inversion are specified informally. Our contribution is to formally establish three specification laws that can be used to derive a specification for almost every interesting inverse program from the specification of its forward program. In this sense, we will have proved that program inversion is a correctness-preserving transformation.

In section 4, we give laws to invert a so-called specification statement. Although such laws are rarely needed, the symmetry between them and the laws in section 2 strikes us as interesting enough for the inclusion.

In section 5, an inversion law is supplied for the program substitution that is used to define the parameter mechanism of procedures. In section 6, we extend the definition of program inversion to the procedure declaration and discuss inversion laws for procedures that contain recursion.

Finally, for the purpose of illustration, we take an example reported in [12]. Small examples are also given throughout. A set of inversion laws is collected in an appendix.

2 Notions and notations

We consider programs in Dijkstra's guarded programming language. They satisfy the following healthiness conditions:

Law of the Excluded Miracle: $[\neg wp(S,F)]$.

Law of (Finite) Conjunctivity: $[wp(S, R \wedge Q) \equiv wp(S,R) \wedge wp(S,Q)]$.

Law of $\wedge$-Independence: $[wp(S, R \wedge Q) \equiv wp(S,R) \wedge (Q \vee wp(S,F))]$
when no program variable of S appears free in Q.

We also extend our program constructs by a specification statement of the form $x : [pre, post]$, which is defined in [9] as

$$wp(x\!:\![pre,post],R) \;\; \widehat{=} \;\; pre \wedge (\forall\, x : post : R).$$

We say that a program S_0 *refines* to S_1, written $S_0 \sqsubseteq S_1$, if $[wp(S_0,R) \Rightarrow wp(S_1,R)]$ holds for all R. We say that a program S is *totally correct* with respect to a pre/postcondition pair *pre* and *post*, if $[pre \Rightarrow wp(S,post)]$.

The specification statement does not necessarily satisfy the Law of the Excluded Miracle, thus a potential *miraculous* statement. We use a specification statement to specify a programming task and then refines it until we reach a program that contains no specification statements. The suitability of this approach is expressed in the following theorem.

Suitability Theorem

A program S is totally correct with respect to a pre/postcondition pair *pre* and *post* if and only if $x:[pre, post] \sqsubseteq S$ where x is the sole program variable of S.

We refer the reader to [1, 11, 9] for more details about the specification statement and refinement. We will abbreviate $\varepsilon:[P,T]$ as $\{P\}$, the usual assert statement. Note that $wp(\{P\}, R) = P \wedge R$.

Now the formalization of program inversion in [4] can be expressed as follows. Program Sw *inverts* to program Sv under precondition P if

$$\{P\} \sqsubseteq Sw;\ Sv.$$

A necessary condition of this is $[P \Rightarrow wp(Sw,T)]$. Taking the weakest possible P, we define an inversion relation $\sqsubseteq^{-1}$ by

$$Sw \ \sqsubseteq^{-1}\ Sv \ \widehat{=}\ \{wp(Sw,T)\} \sqsubseteq Sw; Sv.$$

This observation is due to Von Wright [13]. Note that every program has at least one inverse, executable or not. In fact, $\varepsilon:[true, false]$ is an inverse for any program.

3 Specifying the inverse program

Using program inversion in program construction, we not only need to invert a program, but also need to guarantee that the inverse obtained refines the specification statement that specifies the programming task we set out to resolve. The inverse specification is not simply the one that exchanges the precondition and postcondition of the forward specification.

Example 1 We have $x:[x = 1, x > 0] \sqsubseteq x := x + 1$ and $x := x + 1 \sqsubseteq^{-1} x := x - 1$, but not $x:[x > 0, x = 1] \sqsubseteq x := x - 1$.

The formalization of program inversion suggests that if a program terminates under a certain precondition, then its inverse takes its strongest postcondition as precondition and its precondition as postcondition.

The strongest postcondition of $x := x+1$ with respect to the precondition $x = 1$ is $x = 2$; thus, we have $x : [x = 2, x = 1] \sqsubseteq x := x - 1$. $\Box$

From the above example, we see that the key to specify the inverse program is to find the strongest postcondition of the forward program. In general, it is hard to formulate a strongest postcondition in a closed form. But, there are special cases frequently seen, in which a closed form can easily be formulated. For example, when a program takes a one-point predicate as postcondition, this predicate specifies exactly what a program can achieve and forms the core of its strongest postcondition. Specifically, we have the following specification laws for program inversion.

Law of Inversion Specification

For non-miraculous program Sw, i.e. $wp(Sw, F) = F$, we have

(1) (one-point)

$$\frac{Sw \sqsubseteq^{-1} Sv, \quad x : [P, x = E] \sqsubseteq Sw}{x : [x = E \wedge (\exists\, x :: P),\, P] \sqsubseteq Sv}$$

(2) (universal to existential)

$$\frac{Sw \sqsubseteq^{-1} Sv, \quad x : [P, (\forall\, i : B_i : x = E_i)] \sqsubseteq Sw}{x : [(\exists\, i : B_i : x = E_i) \wedge (\exists\, x :: P),\, P] \sqsubseteq Sv}$$

(3) (existential to universal)

$$\frac{Sw \sqsubseteq^{-1} Sv, \quad x : [P, (\exists\, i : B_i : x = E_i)] \sqsubseteq Sw}{x : [(\forall\, i : B_i : x = E_i) \wedge (\exists\, x :: P),\, P] \sqsubseteq Sv}$$

where x is the sole program variable for Sw and Sv, and x does not occur free in E, and B_i and E_i for any i.

Proof Obviously, (1) can be obtained by taking empty i and true B_i in (2) and (3). The proofs of (2) and (3) are similar, although (2) can be actually derived from (1) with a formal concept [5] of the logical constant. We will prove (3) in the following.

First, it should be clear that for any predicate we introduce can be considered as containing no free i, since we can always rename i to achieve this. Now we derive

$true$

$=$ { rewriting premises }

$(\forall R :: [R \wedge wp(Sw, T) \Rightarrow wp(Sw; Sv, R)]) \wedge [P \Rightarrow wp(Sw, (\exists i : B_i : x = E_i))]$

$\Rightarrow$ { instantiate R and monotonicity of wp }

$[P \Rightarrow wp(Sw, wp(Sv, P)) \wedge wp(Sw, (\exists i : B_i : x = E_i))]$

$=$ { conjunctivity of wp }

$[P \Rightarrow wp(Sw, wp(Sv, P) \wedge (\exists i : B_i : x = E_i))]$

$\Rightarrow$ { $wp(Sv, P)$ contains no free i }

$[P \Rightarrow wp(Sw, (\exists i : B_i : wp(Sv, P)^x_{E_i} \wedge x = E_i))]$

$\Rightarrow$ { calculus }

$[P \Rightarrow wp(Sw, (\exists i : B_i : wp(Sv, P)^x_{E_i}) \wedge (\exists i : B_i : x = E_i))]$

$=$ { $\wedge$-independence of wp, using no free x in B_i and E_i }

$[P \Rightarrow wp(Sw, (\exists i : B_i : x = E_i)) \wedge ((\exists i : B_i : wp(Sv, P)^x_{E_i}) \vee wp(Sw, F))]$

$=$ { second premise, and Sw is non-miraculous }

$[P \Rightarrow (\exists i : B_i : wp(Sv, P)^x_{E_i})]$

$=$ { no free x in the consequent }

$[(\exists x :: P) \Rightarrow (\exists i : B_i : wp(Sv, P)^x_{E_i})]$

$=$ { one-point rule }

$[(\exists x :: P) \Rightarrow (\exists i : B_i : (\forall x : x = E_i : wp(Sv, P)))]$

$\Rightarrow$ { no free x in B_i }

$[(\exists x :: P) \Rightarrow (\exists i :: (\forall x : B_i \Rightarrow x = E_i : wp(Sv, P)))]$

$\Rightarrow$ { $(\exists \forall) \Rightarrow (\forall \exists)$ }

$[(\exists x :: P) \Rightarrow (\exists i :: (B_i \Rightarrow x = E_i) \Rightarrow wp(Sv, P))]$

$\Rightarrow$ { no free i in $wp(Sv, P)$ }

$[(\exists x :: P) \wedge (\forall i : B_i : x = E_i) \Rightarrow wp(Sv, P)]$

$=$ { the suitability of the specification statement }

$x : [(\forall i : B_i : x = E_i) \wedge (\exists x :: P),\ P] \sqsubseteq Sv$

$\square$

Note that Sw is non-miraculous if all specification statements it contains are non-miraculous. The checking is straightforward from the definition.

In most cases, we won't stop refining Sw until it contains no specification statement.

There are many programming examples that can apply the one-point specification to specify the inverse program such as inversion count in [6], binary tree construction from traversals in [4], LU-multiplication in [2], and list compression in [13]. To illustrate the use of these laws, let us take a trivial example.

Example 2 Consider the following Sw and Sv.

$$Sw \;\widehat{=}\; \textbf{if}\, x = 0 \rightarrow x := 5 \,[\!]\, x = 1 \rightarrow x := 0\, \textbf{fi}$$

$$Sv \;\widehat{=}\; \textbf{if}\, x = 5 \rightarrow x := 0 \,[\!]\, x = 0 \rightarrow x := 1\, \textbf{fi}$$

Clearly, $Sw \sqsubseteq^{-1} Sv$, and

$$x\!:\![0 \leq x = X \leq 1, (X = 0 \Rightarrow x = 5) \wedge (X = 1 \Rightarrow x = 0)] \sqsubseteq Sw\ .$$

Then by the Law of Inversion Specification(2),

$$x\!:\![(X = 0 \wedge x = 5) \vee (X = 1 \wedge x = 0), 0 \leq x = X \leq 1] \sqsubseteq Sv$$

which is exactly the same as

$$x\!:\![x = X \wedge (x = 0 \vee x = 5), (X = 0 \Rightarrow x = 1) \wedge (X = 5 \Rightarrow x = 0)] \sqsubseteq Sv$$

when taking X for a logical constant. □

The above three specification laws also suggest a general programming strategy. When the precondition of a specification statement that specifies a programming task we are to fulfill contains the one-point predicate of the above kinds, we can try first to solve a new problem that is specified by essentially exchanging the pre/postcondition of the original specification, and then to invert the program for the new problem. The result of the inversion yields a program that satisfies the original specification.

4 Inverting the specification statement

To specify an inverse, we look for the strongest postcondition of the forward specification. We have found that if the postcondition contains a one-point predicate of some kind, then the strongest postcondition can be easily formulated. But when we invert a program, we are looking for its initial value.

This indicates that if the precondition of a specification statement contains a one-point predicate, we may easily invert this specification statement.

Law of Specification Inversion

(1) $x:[x = E \wedge P, R] \sqsubseteq^{-1} x:[R \wedge P, x = E]$

(2) $x:[(\forall\, i : B_i : x = E_i) \wedge P, R] \sqsubseteq^{-1} x:[R \wedge P, (\exists\, i : B_i : x = E_i)]$ and

(3) $x:[(\exists\, i : B_i : x = E_i) \wedge P, R] \sqsubseteq^{-1} x:[R \wedge P, (\forall\, i : B_i : x = E_i)]$

where no free x occurs in P, E, and B_i and E_i for any i.

Obviously, (1) is a corollary of (2) and (3). The proofs of (2) and (3) are similar, although (3) can be actually derived from (1) with a formal concept [5] of the logical constant. We will prove (2) in the following, but first we give two lemmas.

Lemma 1

$$\{(\forall\, i : B_i : x = E_i) \wedge P\} \sqsubseteq x:[(\forall\, i : B_i : x = E_i) \wedge P, (\exists\, i : B_i : x = E_i)]$$

where no free x occurs in P, and B_i and E_i for any i.

Proof We can assume that no free i appears in the predicates we introduce. For any predicate Q, assume $(\forall\, i : B_i : x = E_i) \wedge P \wedge Q$, i.e. $wp(\{(\forall\, i : B_i : x = E_i) \wedge P\}, Q)$. Then

$$\begin{array}{ll}
 & wp(x:[(\forall\, i : B_i : x = E_i) \wedge P, (\exists\, i : B_i : x = E_i)], Q) \\
= & \quad \{ \text{ semantics } \} \\
 & (\forall\, i : B_i : x = E_i) \wedge P \wedge (\forall\, x : (\exists\, i : B_i : x = E_i) : Q) \\
= & \quad \{ \text{ assumption } \} \\
 & (\forall\, x : (\exists\, i : B_i : x = E_i) : Q) \\
= & \quad \{ \text{ no free } i \text{ in } Q \text{ } \} \\
 & (\forall\, x :: (\forall\, i : B_i \wedge x = E_i : Q)) \\
= & \quad \{ \ \} \\
 & (\forall\, i :: (\forall\, x : B_i \wedge x = E_i : Q)) \\
= & \quad \{ \text{ one point rule, using no free } x \text{ in } B_i \text{ and } E_i \text{ } \} \\
 & (\forall\, i : B_i : Q^x_{E_i})
\end{array}$$

$\Leftarrow \quad \{\ \}$

$\quad (\forall i : B_i : x = E_i \land Q)$

$\Leftarrow \quad \{$ no free i in Q $\}$

$\quad (\forall i : B_i : x = E_i) \land Q$

$= \quad \{$ assumption $\}$

$\quad true$

Hence, by the refinement definition, the lemma holds. □

Lemma 2 $x:[pre \land P, post] = x:[pre \land P, post \land P]$ where P contains no free x.

Proof This is immediate from the definition of the specification statement; thus, omitted. □

Proof of Law of Specification Inversion(2)

$\quad \{wp(x:[P \land (\forall i : B_i : x = E_i), R], T)\}$

$= \quad \{$ semantics $\}$

$\quad \{P \land (\forall i : B_i : x = E_i)\}$

$\sqsubseteq \quad \{$ Lemma 1 $\}$

$\quad x:[P \land (\forall i : B_i : x = E_i), (\exists i : B_i : x = E_i)]$

$\sqsubseteq \quad \{$ introduce sequential composition $\}$

$\quad x:[P \land (\forall i : B_i : x = E_i), P \land R]; x:[P \land R, (\exists i : B_i : x = E_i)]$

$= \quad \{$ Lemma 2 $\}$

$\quad x:[P \land (\forall i : B_i : x = E_i), R]; x:[P \land R, (\exists i : B_i : x = E_i)]$

Hence, by the definition of program inversion, the law holds. □

When applying program inversion in program construction, we first develop a forward program and then invert it to a program we need. In this sense, the forward program is an intermediate program and need not executable. We may choose to leave some simple specification statements unrefined in the forward program, expecting their inverses to be easily implemented, although such cases should be rare.

Example 3 Let l be a logical constant, i.e. it cannot appear in the final program.

$$
\begin{array}{ll}
 & x:[x = f(l),\ x = l] \\
\sqsubseteq^{-1} & \{ \text{ Law of Specification Inversion(1) } \} \\
 & x:[x = l,\ x = f(l)] \\
= & \{ \text{ introducing assignment } \} \\
 & x := f(x)
\end{array}
$$

□

5 Inversion of program substitutions

In order to give inversion laws for the recursive procedure, we need first discuss the inversion of the program substitutions used to define the parameter mechanism of the procedure. For a program *prog*, $prog[f\backslash a]$ is another program obtained by substituting a for f in *prog*. Semantically, we distinguish two types of substitutions in this paper, namely, *value* and *value result*. We will separate one type from the other by a semicolon in the above order. Formally, we define

$$wp(prog[f_0; f_1\backslash A; a], R) \ \widehat{=}\ wp(prog, R^{a}_{f_1})^{f_0,f_1}_{A,a}$$

where f_0 and f_1 are mutually disjoint and do not occur free in R, and A is an expression.

This definition is taken from [10]. The requirement that R contain no free f's is a convenience, rather than a real limitation. We refer the interested reader to [10] for a detailed discussion.

Definition 1 We say that g is *global* in $prog[f_0; f_1\backslash A; a]$ if g appears free in *prog* but is different from f's, and that g is a *global program* variable of $prog[f_0; f_1\backslash A; a]$ if g is global in $prog[f_0; f_1\backslash A; a]$ and a program variable in *prog*. □

Law of Substitution Inversion

$$\frac{Sw \sqsubseteq^{-1} Sv}{Sw[f_0; f_1\backslash A; a] \sqsubseteq^{-1} Sv[f_0; f_1\backslash A; a]}$$

where a is not global in $Sv[f_0; f_1\backslash A; a]$, f_0 is not a program variable of Sw, and no global program variables of $Sw[f_0; f_1\backslash A; a]$ occur free in A.

Proof We can assume that no free f's occur in A. For a predicate R that contains no free f's, we derive

$$\begin{array}{ll}
 & wp(Sw[f_0; f_1 \backslash A; a]; Sv[f_0; f_1 \backslash A; a], R) \\
= & \quad \{ \text{ definitions, let } Q \ \widehat{=} \ R^a_{f_1} \ \} \\
 & wp(Sw[f_0; f_1 \backslash A; a], wp(Sv, Q)^{f_0, f_1}_{A,a}) \\
= & \quad \{ \text{ definition and no free } a \text{ in } wp(Sv, Q) \ \} \\
 & wp(Sw, wp(Sv, Q)^{f_0}_{A})^{f_0, f_1}_{A,a} \\
\Leftarrow & \quad \{ \text{ no free } f_0 \text{ in } A \ \} \\
 & wp(Sw, wp(Sv, Q) \wedge f_0 = A)^{f_0, f_1}_{A,a} \\
= & \quad \{ \text{ independence of } wp \ \} \\
 & (wp(Sw, wp(Sv, Q)) \wedge (f_0 = A \vee wp(Sw, false)))^{f_0, f_1}_{A,a} \\
= & \quad \{ \text{ substitution, no free } f\text{'s in } A \ \} \\
 & wp(Sw, wp(Sv, Q))^{f_0, f_1}_{A,a} \\
\Leftarrow & \quad \{ \text{ premise } \} \\
 & (wp(Sw, true) \wedge Q)^{f_0, f_1}_{A,a} \\
= & \quad \{ \text{ definition of } Q, \text{ no } f\text{'s in } R \ \} \\
 & wp(Sw, true)^{f_0, f_1}_{A,a} \wedge R \\
= & \quad \{ \text{ definition of program substitution } \} \\
 & wp(Sw[f_0; f_1 \backslash A; a], true) \wedge R
\end{array}$$

Hence, by the definition of program inversion, the law holds. □

Note that the free f's in A are different in nature from the free f's in Sw and Sv. We can rename f's in Sw and Sv but not in A. This is why we allow f's to occur free in A while in the proof we can assume no f's are free in A. The additional requirement that f_0 is not a program variable guarantees that when the inverse substitution starts, f_0 assumes the same value as the one when the forward substitution terminates. There are programs that take f_0 as a program variable. We can avoid it by introducing a local variable as f_0's proxy. We require that a should not be global in order to avoid the problem caused by aliasing or side effects, as the following example shows.

Example 4 We do not have

$$(f := f + a)[\varepsilon; f \backslash \varepsilon; a] \sqsubseteq^{-1} (f := f - a)[\varepsilon; f \backslash \varepsilon; a]$$

although in general we have $f := f + a \sqsubseteq^{-1} f := f - a$. This is because f is actually an alias of a and the normal inversion laws cannot apply. □

We have not mentioned the result substitution because we have no need nor capacity to recover the initial value of a result parameter.

6 Inversion of recursive procedures

We consider the following syntax for a procedure declaration.

$$\begin{array}{l} \textbf{proc}\ P(f_0; f_1) \bullet \\ \quad \mathit{Body}(P(A; a)) \\ \textbf{end} \end{array}$$

where *Body* is a sequence of statements and may contain recursive calls to P, and f_0 specifies the value parameter while f_1 specifies the value result parameter. If the *Body* part is not important, we abbreviate a procedure declaration to its head **proc** $P()$.

Semantically, a procedure declaration specifies a parametrized predicate transformer. It is given as the least fixed point of $\mathit{Body}(P[f_0; f_1 \backslash A; a])$ in P, written $(\mu P :: \mathit{Body}(P[f_0; f_1 \backslash A; a]))$. And every procedure call $P(B; b)$ is a predicate transformer of $(\mu P :: \mathit{Body}(P[f_0; f_1 \backslash A; a]))[f_0; f_1 \backslash B; b]$.

Definition 2 We say that **proc** $Pw()$ *inverts* to **proc** $Pv()$, written

$$\textbf{proc}\ Pw() \sqsubseteq^{-1} \textbf{proc}\ Pv()$$

if the parametrized predicate transformer denoted by **proc** $Pw()$ inverts to the parametrized predicate transformer denoted by **proc** $Pv()$ □

Law of Procedure Declaration Inversion

$$\frac{(\forall Xw, Xv : Xw \sqsubseteq^{-1} Xv : \mathit{Body}_w(Xw) \sqsubseteq^{-1} \mathit{Body}_v(Xv))}{\begin{array}{lcl} \textbf{proc}\ Pw(f_0; f_1) \bullet & & \textbf{proc}\ Pv(f_0; f_1) \bullet \\ \quad \mathit{Body}_w(Pw(A; a)) & \sqsubseteq^{-1} & \quad \mathit{Body}_v(Pv(A; a)) \\ \textbf{end} & & \textbf{end} \end{array}}$$

where a is not global of the procedure declaration Pv, A contains no global program variables of the procedure declaration Pw, and f_0 is not a program variable of the procedure declaration Pw.

Proof First we define

$$\begin{array}{rcl} Sw(Xw) & \widehat{=} & \mathit{Body}_w(Xw[f_0; f_1 \backslash A; a]) \\ Sv(Xv) & \widehat{=} & \mathit{Body}_v(Xv[f_0; f_1 \backslash A; a]) \end{array}$$

Then our proof obligation is to show that

$$(\mu Pw :: Sw(Pw)) \sqsubseteq^{-1} (\mu Pv :: Sv(Pv))$$

Since the least fixed point can be approximated from *abort* on ordinal numbers, it suffices to show $Sw^\lambda \sqsubseteq^{-1} Sv^\lambda$ for any ordinal λ.

We use induction to show this. The base case is trivial, since *abort* $\sqsubseteq^{-1}$ *abort*.

Step Case:

$$
\begin{array}{ll}
 & Sw^{\lambda+1} \\
= & \{ \text{ definition of approximation } \} \\
 & Sw(Sw^\lambda) \\
= & \{ \text{ definition of } Sw \text{ } \} \\
 & Body_w(Sw^\lambda[f_0; f_1 \backslash A; a]) \\
\sqsubseteq^{-1} & \{ \text{ I.H., Substitution Inversion, premise } \} \\
 & Body_v(Sv^\lambda[f_0; f_1 \backslash A; a]) \\
= & \{ \text{ definitions } \} \\
 & Sv^{\lambda+1}
\end{array}
$$

Limit Case:

$$
\begin{array}{ll}
 & \{wp(Sw^\lambda, true)\} \\
= & \{ \text{ approximation, using } \lambda \text{ is a limit } \} \\
 & \{(\exists \beta : \beta < \lambda : wp(Sw^\beta, true))\} \\
= & \{ \ \} \\
 & (\sqcup \beta : \beta < \lambda : \{wp(Sw^\beta, true)\}) \\
\sqsubseteq & \{ \text{ inductive hypothesis } \} \\
 & (\sqcup \beta : \beta < \lambda : Sw^\beta; Sv^\beta) \\
\sqsubseteq & \{ \text{ approximations are ascending } \} \\
 & (\sqcup \beta : \beta < \lambda : Sw^\beta; Sv^\lambda) \\
= & \{ \ \} \\
 & (\sqcup \beta : \beta < \lambda : Sw^\beta); Sv^\lambda \\
= & \{ \text{ approximation, using } \lambda \text{ is a limit } \} \\
 & Sw^\lambda; Sv^\lambda
\end{array}
$$

Hence, $Sw^\lambda \sqsubseteq^{-1} Sv^\lambda$ for any ordinal λ, then the law holds. □

This law tells us that in inverting a procedure declaration, we need to find another so-called inverse procedure declaration, and that such an inverse declaration can be obtained by inverting the body of the forward procedure declaration. During inversion, each recursive call in the forward procedure is simply renamed to a call to the inverse procedure declaration with the same parameters.

Usually, a specification statement is attached to a procedure declaration for the purpose of facilitating the (re)use of the procedure. Semantically, this specification statement refines to the parametrized predicate transformer denoted by the procedure declaration. Syntactically, we write

```
**proc** $P(f_0; f_1)$
  $- x, f_1 : [pre, post] \bullet$
      *Body*
**end**
```

where x contains all global program variables of P. Note that we have assumed that the value parameter f_0 is not a program variable.

To introduce a call to it in program development, we refine a specification substitution $x, f_1 : [pre, post][f_0; f_1 \backslash A; a]$ to $P(A; a)$. For example, given the following procedure declaration

```
**proc** $add(f_0; f_1)$
  $- f_1 : [f_1 = F, f_1 = f_0 + F] \bullet$
      $f_1 := f_0 + f_1$
**end**
```

then we can have the derivation

$$
\begin{array}{ll}
 & x : [x = X, x = X + 1] \\
= & \quad \{ \text{ rename the initial value } \} \\
 & x : [x = F, x = F + 1] \\
\sqsubseteq & \quad \{ \text{ introduce substitution } \} \\
 & f_1 : [f_1 = F, f_1 = F + f_0][f_0; f_1 \backslash 1; x] \\
\sqsubseteq & \quad \{ \text{ introduce procedure call } \} \\
 & add(1; x)
\end{array}
$$

By applying the Law of Inversion Specification, we can also derive a specification statement for an inverse procedure declaration.

Example 5 Consider inverting

```
proc $Pw(\varepsilon; f)$
  $- f : [f = F \geq 0, f = 2^F] \bullet$
    if $f = 0 \rightarrow f := 1$
    $[\!]\, f \neq 0 \rightarrow f := f - 1; Pw(\varepsilon, f); f = 2 * f$
    fi
end
```

We choose Pv for the name of an inverse declaration we will end with. So by the Law of Procedure Declaration Inversion, we need to invert the body of Pw under the assumption that $Pw(\varepsilon; f) \sqsubseteq^{-1} Pv(\varepsilon; f)$. In order to invert the **if** statement, we need to find mutually exclusive postconditions for the two guarded commands, which will be taken as the guards in the inverse program we are looking for. Obviously, the first guarded command ends with $f = 1$ while the second ends with $f \neq 1$.

We also need to invert the three assignments. We do not know how to directly invert $f := 1$ to an executable program, but this statement has a precondition of $f = 0$. It is straightforward that $\{f = 0\} f := 1 \sqsubseteq^{-1} f := 0$.

We will carry out the inversion process in a "calculational" style. For a program Sw, $(Sw)^{-1}$ denotes the set of its inverse programs. The similar notation is also used for the procedure declaration. An inversion process is to reduce such a set to a single program. (A formal treatment of this style will be reported elsewhere.)

```
  (  proc $Pw(\varepsilon; f)$                                        )^{-1}
  (   $- f : [f = F \geq 0, f = 2^F] \bullet$                          )
  (      if $f = 0 \rightarrow f := 1$                                 )
  (      $[\!]\, f \neq 0 \rightarrow f := f - 1; Pw(\varepsilon, f); f := 2 * f$ )
  (      fi                                                          )
  (  end                                                             )
$=$    { add assertions }
  (  proc $Pw(\varepsilon; f)$                                        )^{-1}
  (   $- f : [f = F \geq 0, f = 2^F] \bullet$                          )
  (      if $f = 0 \rightarrow \{f = 0\} f := 1 \quad \{f = 1\}$        )
  (      $[\!]\, f \neq 0 \rightarrow f := f - 1; Pw(\varepsilon, f); f := 2 * f \; \{f \neq 1\}$ )
  (      fi                                                          )
  (  end                                                             )
$\supseteq$    { introduce the name $Pv$, Inversion Specification(1) }
```

```
proc $Pv(\varepsilon; f)$
 $- f:[f = 2^F \wedge F \geq 0, f = F] \bullet$
   $\left( \begin{array}{l} \textbf{if}\ f = 0 \rightarrow \{f = 0\} f := 1 \quad \{f = 1\} \\ [\!]\ f \neq 0 \rightarrow f := f - 1; Pw(\varepsilon, f); f := 2 * f\ \{f \neq 1\} \\ \textbf{fi} \end{array} \right)^{-1}$
end
$\supseteq$  { Alternative Inversion }
proc $Pv(\varepsilon; f)$
 $- f:[f = 2^F \wedge F \geq 0, f = F] \bullet$
     if $f = 1 \rightarrow (\{f = 0\} f := 1)^{-1}$
     $[\!]\ f \neq 1 \rightarrow (f := f - 1; Pw(\varepsilon, f); f := 2 * f)^{-1}$
     fi
end
$\supseteq$  { $\{f = 0\} f := 1 \sqsubseteq^{-1} f := 0$, Semicolon Inversion }
proc $Pv(\varepsilon; f)$
 $- f:[f = 2^F \wedge F \geq 0, f = F] \bullet$
     if $f = 1 \rightarrow f := 0$
     $[\!]\ f \neq 1 \rightarrow (f := 2 * f)^{-1}; (Pw(\varepsilon, f))^{-1}; (f := f - 1)^{-1}$
     fi
end
$\supseteq$  { Assignment Inversion, assumption }
proc $Pv(\varepsilon; f)$
 $- f:[f = 2^F \wedge F \geq 0, f = F] \bullet$
     if $f = 1 \rightarrow f := 0$
     $[\!]\ f \neq 1 \rightarrow f := f/2; Pv(\varepsilon, f); f := f + 1$
     fi
end
```

Hence, **proc** $Pw() \sqsubseteq^{-1}$ **proc** $Pv()$. □

By applying the Law of Substitution Inversion, we have

Law of Procedure Call Inversion

$$\frac{\textbf{proc}\ Pw() \sqsubseteq^{-1} \textbf{proc}\ Pv()}{Pw(A; a) \sqsubseteq^{-1} Pv(A; a)}$$

where a is not global of the procedure declaration Pv, A contains no global program variables of the procedure declaration Pw, and the value parameter f_0 is not a program variable of the procedure declaration Pw.

Example 6 Given the following procedure declaration

```
proc $add_w(f_0; f_1)$
 $-\ f_1 : [f_1 = F, f_1 = f_0 + F] \bullet$
    $f_1 := f_1 + f_0$
end
```

then one inverse declaration of it is

```
proc $sub_v(f_0; f_1)$
 $-\ f_1 : [f_1 = f_0 + F, f_1 = F] \bullet$
    $f_1 := f_1 - f_0$
end
```

By the Law of Procedure Call Inversion, $add_w(1; x) \sqsubseteq^{-1} sub_v(1; x)$.

Since $x : [x = X, x = X + 1] \sqsubseteq add_w(1; x)$, by the Law of Specification Inversion(1), we obtain $x : [x = X + 1, x = X] \sqsubseteq sub_v(1; x)$. □

7 Inversion of recursive tree traversals

In this section, we invert a recursive procedure that generates the pre- and inorder traversals for a given labeled binary tree to yield another recursive procedure of constructing a labeled binary tree. Our main purpose is to illustrate applying the inversion laws in the development of a more interesting program, rather than the novelty of the program per se. Actually, the problem is first addressed by van de Snepscheut in [12]. In our opinion, there are some gaps that remain in the correctness arguments of his inverse program, since there is no inversion law given for procedures. In contrast, our derivation will be fully justified by a set of formally established inversion laws, together with the standard programming laws.

A labeled binary tree is either empty $\perp$ or a triple $\langle l, d, r\rangle$ with label d and left and right subtrees l and r. For a nonempty tree t, $t.d$, $t.l$, $t.r$, $in(t)$ and $pre(t)$ denote its label, left subtree, right subtree, inorder traversal and preorder traversal, respectively. We use $+\!\!+$ for catenating sequences.

Then for d of label type, t of tree and x and y of sequence, the following tree traversal procedure is given by van de Snepscheut in [12].

```
proc $traverse(d,t;\varepsilon)$
  $- x,y:[x,y = X,Y\ ,\ x,y = in(t){+\!\!+}d{+\!\!+}X, pre(t){+\!\!+}Y] \bullet$
     if $t = \perp \rightarrow x := d{+\!\!+}x$
     $[\!]\ t \neq \perp \rightarrow traverse(d,t.r);\ traverse(t.d,t.l);$
                  $y := t.d{+\!\!+}y$
     fi
end
```

Simply from the specification of this procedure, we observe that applying the Law of Specification Inversion does not yield a specification that describes a computation of constructing a labeled binary tree. So we change t into a value result parameter, and the procedure becomes

```
proc $traverse(d;t)$
  $- x,y,t:[x,y,t = X,Y,T\ ,$
              $x,y,t = in(T){+\!\!+}d{+\!\!+}X, pre(T){+\!\!+}Y, \perp] \bullet$
     if $t = \perp \rightarrow x := d{+\!\!+}x$
     $[\!]\ t \neq \perp \rightarrow traverse(d,t.r);\ traverse(t.d,t.l);$
                  $y := t.d{+\!\!+}y;\ t := \perp$
     fi
end
```

As usual, to invert the above procedure, we need to add some assertions. Note that a tree can be uniquely constructed only if all its labels are distinct, so by assuming this we annotate the above procedure as follows.

```
proc $traverse(d;t)$
  $- x,y,t:[(x,y,t = X,Y,T) \wedge distinct(d,t)\ ,$
              $x,y,t = in(T){+\!\!+}d{+\!\!+}X, pre(T){+\!\!+}Y, \perp] \bullet$
     if $t = \perp \rightarrow x := d{+\!\!+}x\ \ \{d = hd(x)\}$
     $[\!]\ t \neq \perp \rightarrow traverse(d,t.r);\ traverse(t.d,t.l);\ y := t.d{+\!\!+}y;$
                  $\{t = \langle \perp, hd(y), \perp \rangle\} t := \perp\ \ \{d \neq hd(x)\}$
     fi
end
```

which obviously inverts to

```
**proc** $tree(d;t)$
  $- x,y,t:[(x,y,t = in(T){+\!\!+}d{+\!\!+}X, pre(T){+\!\!+}Y, \bot)$
                                  $\wedge distinct(d,T)\,,\ \ x,y,t = X,Y,T] \bullet$
      **if** $d = hd(x) \rightarrow x := tl(x)$
      $[\!]\ d \neq hd(x) \rightarrow t := \langle \bot, hd(y), \bot \rangle;\ y := tl(y);$
                         $tree(t.d, t.l);\ tree(d, t.r)$
      **fi**
**end**
```

This procedure can be used to develop a program of constructing a labeled binary tree from its pre- and inorder traversals. But note that the initial condition of $t =\bot$ is not needed in the second guarded command and retained at the end of the first guarded command, so we can further transform the above procedure by removing the initial condition of $t =\bot$:

```
**proc** $tree(d;t)$
  $- x,y,t:[(x,y = in(T){+\!\!+}d{+\!\!+}X, pre(T){+\!\!+}Y)$
                                  $\wedge distinct(d,T)\,,\ \ x,y,t = X,Y,T] \bullet$
      **if** $d = hd(x) \rightarrow x := tl(x);\ t :=\bot$
      $[\!]\ d \neq hd(x) \rightarrow t.d := hd(y);\ y := tl(y);$
                         $tree(t.d, t.l);\ tree(d, t.r)$
      **fi**
**end**
```

The above procedure is given in [12] as an inverse of the procedure at the beginning of this section. Clearly, this is not the case with our formalism.

8 Conclusion

We have presented laws to specify the inverse program. These laws indicate that program inversion is a correctness-preserving transformation. The correctness of the inverse program can be derived from that of its forward program. We have also established some laws to invert procedures that can contain recursion. Since a recursive solution usually contains fewer explicit states than its repetitive counterpart while the process of program inversion is essentially to recover the changes on the state space, it is very likely that a recursive solution is easier to invert. This is confirmed by the problem of constructing a tree from its traversals. We invite the reader to compare the inversion process of this paper with the one in [4] that gives a repetitive solution. It is our hope that the formal establishment of the procedure inversion laws will expand the application domain of program inversion.

In general, programming is a hard task; anything that makes it easier should be welcome. In [8], nearly one hundred laws of programming are given. Although these laws are claimed to constitute a complete set in the sense that some clearly defined subset of truths about programming can be deduced, it is also mentioned that we are still a long way from knowing how to apply them in program construction on the scale required by modern technology and that deeper and more specific techniques are desperately needed. We believe that program inversion is becoming one of such needed techniques.

9 Appendix: inversion laws

9.1 Definition

$$Sw \sqsubseteq^{-1} Sv \quad \widehat{=} \quad \{wp(Sw, T)\} \sqsubseteq Sw; Sv.$$

9.2 Primitives

Law of Assignment Inversion

$$\frac{P \Rightarrow def(Ev)^{x}_{Ew} \wedge x = Ev^{x}_{Ew}}{\{P\}x := Ew \sqsubseteq^{-1} x := Ev}$$

□

Law of Assertion Inversion

$$\{P\} \sqsubseteq^{-1} \{P\}$$

□

Law of Specification Inversion

(1) $x : [x = E \wedge P, R] \sqsubseteq^{-1} x : [R \wedge P, x = E]$

(2) $x : [(\forall i : B_i : x = E_i) \wedge P, R] \sqsubseteq^{-1} x : [R \wedge P, (\exists i : B_i : x = E_i)]$ and

(3) $x : [(\exists i : B_i : x = E_i) \wedge P, R] \sqsubseteq^{-1} x : [R \wedge P, (\forall i : B_i : x = E_i)]$

where no free x occurs in P, E, and B_i and E_i for any i. □

9.3 Program constructors

Law of Semicolon Inversion

$$\frac{Sw_0 \sqsubseteq^{-1} Sv_0, \quad Sw_1 \sqsubseteq^{-1} Sv_1}{Sw_0;\ Sw_1 \ \sqsubseteq^{-1}\ Sv_1; Sv_0}$$

□

Law of Alternation Inversion

$$\frac{(\forall\, i,j : i \neq j : Bv_i \wedge Bv_j = F),\ (\forall\, i :: Sw_i \ \sqsubseteq^{-1}\ Sv_i)}{\mathbf{if}\, i :: Bw_i \rightarrow Sw_i\, \{Bv_i\}\, \mathbf{fi} \ \sqsubseteq^{-1}\ \mathbf{if}\, i :: Bv_i \rightarrow Sv_i\, \mathbf{fi}}$$

□

Law of Block Inversion

$$\frac{Sw \ \sqsubseteq^{-1}\ Sv}{|[\mathbf{var}\, l; S; Sw\, \{l = E\}]| \ \sqsubseteq^{-1}\ |[\mathbf{var}\, l; l := E; Sv]|}$$

provided that l is the sole program variable of S and no l appears in E. □

Law of Iteration Inversion

$$\frac{Sw \ \sqsubseteq^{-1}\ Sv}{\{\neg Bv\}\mathbf{do}\, Bw \rightarrow Sw\, \{Bv\}\, \mathbf{od} \ \sqsubseteq^{-1}\ \mathbf{do}\, Bv \rightarrow Sv\, \mathbf{od}}$$

□

Law of Substitution Inversion

$$\frac{Sw \sqsubseteq^{-1} Sv}{Sw[f_0; f_1 \backslash A; a] \sqsubseteq^{-1} Sv[f_0; f_1 \backslash A; a]}$$

where a is not global in $Sv[f_0; f_1\backslash A; a]$, f_0 is not a program variable of Sw, and no global program variables of $Sw[f_0; f_1\backslash A; a]$ occur free in A. □

Law of Procedure Declaration Inversion

$$\frac{(\forall\, Xw, Xv : Xw \sqsubseteq^{-1} Xv : Body_w(Xw) \sqsubseteq^{-1} Body_v(Xv))}{\begin{array}{lcl} \mathbf{proc}\ Pw(f_0; f_1) \bullet & & \mathbf{proc}\ Pv(f_0; f_1) \bullet \\ \quad Body_w(Pw(A; a)) & \sqsubseteq^{-1} & \quad Body_v(Pv(A; a)) \\ \mathbf{end} & & \mathbf{end} \end{array}}$$

where a is not global of the procedure declaration Pv, A contains no global program variables of the procedure declaration Pw, and f_0 is not a program variable of the procedure declaration Pw. □

Law of Procedure Call Inversion

$$\frac{\textbf{proc } Pw() \sqsubseteq^{-1} \textbf{proc } Pv()}{Pw(A;a) \ \sqsubseteq^{-1} \ Pv(A;a)}$$

where a is not global of the procedure declaration Pv, A contains no global program variables of the procedure declaration Pw, and the value parameter f_0 is not a program variable of the procedure declaration Pw. □

9.4 Miscellaneous

Law of Straightforward Inversion

$$\{x = E\}Sw \ \sqsubseteq^{-1} \ x := E$$

where x is the sole program variable of Sw and E contains no free x. □

Law of Inverse Simplification

$$\frac{Sw \sqsubseteq^{-1} Sv_0, \quad Sv_0 \sqsubseteq Sv_1}{Sw \ \sqsubseteq^{-1} \ Sv_1}$$

□

9.5 Specification

Law of Inversion Specification

For non-miraculous program Sw, i.e. $wp(Sw, F) = F$, we have

(1) (one-point)

$$\frac{Sw \ \sqsubseteq^{-1} \ Sv, \quad x : [P, x = E] \ \sqsubseteq \ Sw}{x : [x = E \wedge (\exists\, x :: P),\, P] \ \sqsubseteq \ Sv}$$

(2) (universal to existential)

$$\frac{Sw \ \sqsubseteq^{-1} \ Sv, \quad x : [P, (\forall\, i : B_i : x = E_i)] \ \sqsubseteq \ Sw}{x : [(\exists\, i : B_i : x = E_i) \wedge (\exists\, x :: P),\, P] \ \sqsubseteq \ Sv}$$

(3) (existential to universal)

$$\frac{Sw \sqsubseteq^{-1} Sv, \quad x : [P, (\exists\, i : B_i : x = E_i)] \sqsubseteq Sw}{x : [(\forall\, i : B_i : x = E_i) \land (\exists\, x :: P),\ P] \sqsubseteq Sv}$$

where x is the sole program variable for Sw and Sv, and x does not occur free in E, and B_i and E_i for any i. $\square$

9.6 Notes

1 All the proofs of the above laws can be found in either [3] or this paper.

2 All the above laws except probably the Law of Inverse Simplification can be written in the calculational style, e.g. the Law of Semicolon Inversion can be written as $(S_0;\, S_1)^{-1} \supseteq (S_1)^{-1}\, ;\, (S_0)^{-1}$.

References

[1] Back, R.J.R.: "A calculus of refinements for program derivations", *Acta Informatica* **25**, 593-624, 1988.

[2] Chen, W.: "A formal approach to program inversion", in: *Proc. of 1990 ACM 18th Ann. Comp. Sci. Conf.*, Washington, DC, 398-403, 1990.

[3] Chen, W.: "Programming by transformation – theory and methods", D.Sc. Dissertation, Washington University (St. Louis). May 1991.

[4] Chen, W. and Udding, J.T.: "Program inversion: more than fun!", *Sci. of Comp. Prog.* **15**, 1-13, 1990.

[5] Chen, W. and Udding, J.T.: "The specification statement refined", WUCS-89-37, Washington University (St. Louis), 1989. Under revision.

[6] Dijkstra, E.W. and Feijen, W.H.J.: *A Method of Programming*, Addison-Wesley Publishing Company, Reading, MA, 1988.

[7] Gries, D.: *The Science of Programming*, Springer-Verlag, New York, NY, 1981.

[8] Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M. and Sufrin, B.A.: "Laws of programming", *CACM* **30**, 672-686, 1987.

[9] Morgan, C.C.: "The specification statement", *ACM TOPLAS* **10**, 403-419, 1988.

[10] Morgan, C.C.: "Procedures, parameters, and abstraction: separate concerns", *Sci. of Comp. Prog.* **11**, 17-27, 1988.

[11] Morris, J.M.: "A theoretical basis for stepwise refinement and the programming calculus", *Sci. of Comp. Prog.* **9**, 287-306, 1987.

[12] Snepscheut, J.L.A. van de: "Inversion of a recursive tree traversal", *IPL* **39**, 265-267, 1991.

[13] Wright, J. von: "Program inversion in the refinement calculus", to appear in *IPL*.