Chair for Software Systems
Institute for Informatics
LMU Munich

Bachelor Thesis in Computer Science:

# Specifying Loops With Contracts

## Reasoning about loops as recursive procedures

Gregor Cassian Alexandru

August 8, 2019

I dedicate this work to the memory of Prof. Martin Hofmann, who was an amazing teacher and inspiration, and left us all too soon.

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

................................................ (Unterschrift des Kandidaten)
München, den 8. August 2019

## Abstract

Recursive procedures can be inductively verified to fulfill some contract, by assuming that contract for recursive calls. Since loops correspond essentially to linear recursion, they ought to be verifiable in an analogous manner. Rules that allow this have been proposed, but are incomplete in some aspect, or do not lend themselves to straightforward implementation. We propose some implementation-oriented derivatives of the existing verification rules and further illustrate the advantages of contract-based loop specification by means of examples.

# Contents

# 1 Introduction

Writing correct software is hard. If we want to be certain of a program's correctness, we need to do two things: First specify how it ought to behave, then verify that it fulfills that specification. When writing programs that should perform some finite computation, this is usually done using pre- and postconditions, which together form the program's *contract*. The postcondition often relates the final to the initial state.

The process of finding logical formulae that, if valid, prove the correctness of the implementation of a contract (assuming some semantics for the language used), is called *verification condition generation*. The *VCG* for a program that contains no loops or procedure calls may be completely automated using *predicate transformers* [Dij76].

Of course, in the course of writing a program to implement some contract, we might need to do iteration, perhaps using a loop. Unrolling the loop will not terminate in general, even in simple cases. For example, if we wanted to prove such a simple statement as $\{i \geq 0\}$**while** $i \neq 0$ **do** $i := i - 1$ **done** $\{i = 0\}$, we could not do it using loop-unrolling, since $i$ has no upper bound.

If we put aside for a moment the possibility of nontermination, we can prove some property of a loop by induction over the number of its iterations. The most studied such inductive property since [Hoa69] has been the *invariant assertion*, briefly known as the "invariant": An assertion that holds before and after each step in the iteration.

We would be similarly justified in wanting to specify a loop by its relation of the pre- to the post-iteration state. This can also be phrased as an inductive property, but requires alternative ways of reasoning about loops. Tuerk [Tue10] states an axiomatic rule that allows this, but uses higher-order logic to do so, making the rule hard to implement directly and overshooting the rule's intention a bit in generality.

Hehner [HG99] proposes a refinement semantics for loops which allows inductively proving a loop fulfills some "specification", which by his definition may explicitly be a relation of pre- and post-execution states. Since he doesn't give these a predicate transformer semantics though, the approach is not integrated.

To give an overview, in this thesis we:

- Recognize the usefulness of the specification statement [Mor88] for using and verifying loops with contracts, especially as a reification of the induction hypothesis in the inductive step of the verification proof.
- Define a rule for inductive verification of while-loops for partial correctness based on this, in predicate transformer semantics (Section 4.2), as well as as an axiomatic rule (Section 5.2).
- Prove the soundness of our weakest-precondition rule in the refinement calculus, using refinement semantics for loops [HG99] (Section 4.3)
- Describe our prototypical implementation of the rule in a verification tool for a model language, written in `Scala` (Section 6).
- Specify and verify some textbook algorithms using our rules, observing how it compares to the invariant (Section 7).

## 2 Motivating Example

**Note on the code** The following code snippets will all use a Python-like pseudocode, annotated with specifications in *[slanted brackets]*. We use Python slice notation for sublists in the specification. We use separation logic[Rey02] to talk about the state of the heap.

We want to prove the correctness of a procedure which concatenates two lists:

```
[pre: list(xs,xdata)*list(ys,ydata)]
def concat(xs,ys):
   if xs == None: xs = ys
   else:
      zs = xs
      while zs.next != None:
         zs = zs.next
      zs.next = ys
[post: list(xs,xdata++ydata)]
```

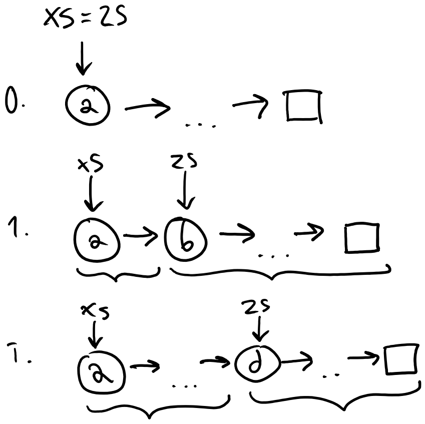
The lists we are dealing with here are singly-linked, `None`-terminated heap-allocated reference types. The precise definition of the `list()` predicate is given in Figure 1.

$$\boxed{\begin{aligned} list(p,[\,]) &\Leftrightarrow p = None \\ list(p, x :: xs) &\Leftrightarrow \exists q.\ (p.data \mapsto x)\ *\ (p.next \mapsto q)\ *\ list(q, xs) \end{aligned}}$$

Figure 1: `list()` Definition

### 2.1 Verification using an Invariant

We will verify this iterative implementation of the algorithm by specifying the loop with an invariant. To find the invariant, we visualize the iteration happening over the linked list `xs`:



We find the invariant to be that in every iteration, xs is split at zs into an initial *segment*, containing data1, and the remaining list zs, containing data2, so that data1++data2==xdata. At the end of iteration, we have lseg(xs,xdata[:-1],zs) and list(zs,xdata[-1:]). With the redirection of zs.next to ys *after the loop*, we have the desired postcondition, list(xs,xdata++ydata).

```
[pre: list(xs,xdata)*list(ys,ydata)]
def concat(xs,ys):
   if xs == None: xs = ys
   else:
      zs = xs
      while zs.next != None:
      [inv:∃ data1,data2. xdata=data1++data2*
       lseg(xs,data1,zs)*list(zs,data2)]
         zs = zs.next
      zs.next = ys
[post: list(xs,xdata++ydata)]
```

We notice two things:

- Because the computation is only completed (that is, the pointer is redirected) *after* the loop has ended, the invariant only ensures that our program is in the desired *state* at the end of iteration.
- We therefore talk about the states left by incomplete computations. This requires us to introduce the new predicate lseg().

## 2.2 Using a local Recursive Procedure

We want to find an approach where we only need to specify complete computations. This means we need to treat the loop and subsequent statement as a single block. We can rewrite our program to this intent by locally defining a recursive procedure loop(zs : List), which we immediately apply:

```
[pre: list(xs,xdata)*list(ys,ydata)]
def concat(xs,ys):
   if xs == None: xs = ys
   else:
      def loop(zs):
         if zs.next == None:
             zs.next = ys
         else:
            loop(zs.next)
      loop(xs)
[post: list(xs,xdata++ydata)]
```

We now need to annotate the local recursive procedure with pre- and postconditions. Our proof obligation will be to show:

- That the specification gives us the desired result at the place where the procedure is called (`loop(xs)`)
- That the procedure fulfills its specification.

We will try to deduce the specification:

- To avoid a `None`-dereference with `zs.next`, we need to ensure that `zs != None`. This will be in the precondition of `loop()`.
- In order for the method's postcondition of `list(xs,xdata++ydata)` to hold after the call `loop(xs)`, the postcondition of `loop()` must be `list(zs,zdata++ydata)` and `list(zs,zdata)*list(ys,ydata)` must be added to the precondition accordingly.

The specification is therefore:

```
[pre: zs != None, list(zs,zdata)*list(ys,ydata)]
def loop(zs):
    if zs.next == None:
        zs.next = ys
    else:
        loop(zs.next)
[post: list(zs,zdata++ydata)]
```

We use Hoare's axiomatic rules for recursive procedures [Hoa71] to draw the proof outline for verifying the call `loop(xs)` in Figure 2.

$$\begin{aligned} P &:= \texttt{zs!=None,list(zs,zdata)*list(ys,ydata)} \\ Q &:= \texttt{list(zs,zdata++ydata)} \\ \sigma &:= [\texttt{zs}\backslash\texttt{xs}, \texttt{zdata}\backslash\texttt{xdata}] && \text{(Parameter substitution)} \\ \text{IH} &:= \{P\}\texttt{loop(zs)}\{Q\} && \text{(Induction Hypothesis)} \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{Induction base}}{\vdash \{P \wedge \texttt{zs.next==None}\}\texttt{zs.next=ys}\{Q\}}\ \vdots \qquad \dfrac{\text{Inductive step}}{\text{IH} \vdash \{P \wedge \texttt{zs.next!=None}\}\texttt{loop(zs.next)}\{Q\}}\ \vdots}{\text{IH} \vdash \{P\}\ \textbf{if}\ \texttt{zs.next==None}\ \textbf{:}\ \texttt{zs.next=ys}\ \textbf{else:}\ \texttt{loop(zs.next)}\{Q\}}\ \text{\textsc{Cond}}}{\vdash \{P\}\texttt{loop(zs)}\{Q\}}\ \text{Recursive Invocation}}{\vdash \{P\sigma\}\texttt{loop(xs)}\{Q\sigma\}}\ \text{Parameter Substitution}$$

Figure 2: Recursive Invocation Proof Tree

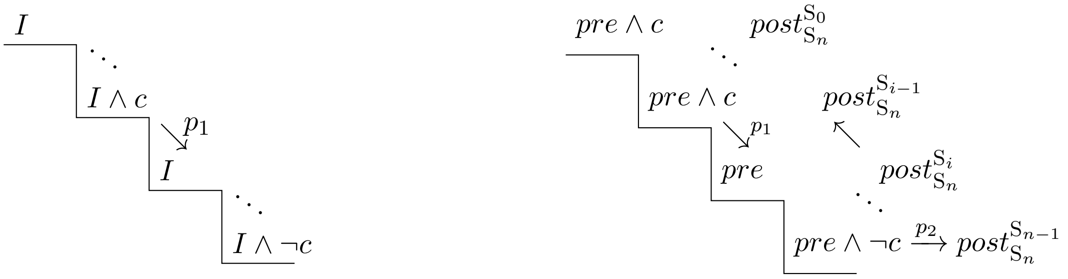
To prove the specification, it's enough that in the inductive case `zs.next` is not redirected, since in the base case the last pointer of the chain is redirected to `ys`, which causes the lists to be concatenated, which result is propagated up the call stack due to the recursive definition of the datatype.

## 2.3 Comparison

At this point we should pause and reflect on the differences between the approaches. The invariant talks about states. pre- and postconditions talk about a change in state. The invariant talks about incomplete computations. A recursive procedure talks about a complete computation in its postcondition, possibly by relying on its recursive calls to establish the postcondition in their respective state.

In the particular case of linear iteration over linked datastructures, inductive verification gives us the additional benefit that it follows the recursive definition of the type (compare Definition 1). This is a consequence of the fact that iteration with loops corresponds to linear recursion, and linear recursive functions are *hylomorphisms* of cons-lists [MFP91, p. 4].

Following graphic might serve to illustrate the differences:



The specification with an invariant assertion doesn't take full advantage of the power of an inductive proof. If we want to prove the loop establishes some relation *post* between the state before iteration, $S_0$, and the final state $S_n$, we can show inductively that it holds in the base case ($post_{S_n}^{S_{n-1}}$) and inductively that $post_{S_n}^{S_i} \rightarrow post_{S_n}^{S_{i-1}}$. We can see that *pre* acts like an invariant everywhere except for in the base case, so we could describe the invariant as an eternal precondition, a description that fits well with the "incomplete computations" we already related it to.

# 3 Preliminaries

We concern ourselves with a simple *while*-language:

$$
\begin{aligned}
\mathit{Prog} ::= \;& \mathit{skip} && \text{(do nothing)}\\
\mid\;& \mathit{Var}_\tau := \mathit{Expr}_\tau && \text{(assignment)}\\
\mid\;& \{\mathit{Prog};\mathit{Prog}\} && \text{(sequential composition)}\\
\mid\;& \mathbf{if}\ \mathit{Expr}_{\mathit{Bool}}\ \mathbf{then}\ \mathit{Prog}\ \mathbf{else}\ \mathit{Prog} && \text{(conditional composition)}\\
\mid\;& \mathbf{while}\ \mathit{Expr}_{\mathit{Bool}}\ \mathbf{do}\ \mathit{Prog}\ \mathbf{done}\ \mathit{Prog} && \text{(while loop)}\\
\mid\;& \vec{\mathit{Var}} : [\mathit{Assertion}, \mathit{Assertion}] && \text{(specification statement)}
\end{aligned}
$$

- The *specification statement* is introduced in section 4.1.
- $\mathit{Var}_\tau, \mathit{Expr}_\tau$ denote variables/expressions of type $\tau$. We won't go into detail on the grammar of these expressions, suffice it to say we assume the usual operations on integer and Boolean expressions are available.
- We, like Tuerk [Tue10], consider loops of the form **while** $c$ **do** $p_1$ **done** $p_2$, that is, that have a rest program which executes the base case of the iteration. We saw in the motivating example a case where this was vital; in general it draws a nicer correspondence to conditional statements, which are also of the form **if** $c$ **then** $p_1$ **else** $p_2$. Hehner does not consider this in [HG99], so we have slightly altered his refinement semantics rule to accommodate for it, in Section 4.1.
- **while**-loops without **done** branches or **if**-statements without **else**-branches may be written as **while** $c$ **do** $p$ and **if** $c$ **then** $p$, respectively.

### Note on Notation:

- With $P[\vec{x}\backslash\vec{y}]$ we mean "$P$, where all free occurrences of $\vec{x}$ have been replaced by $\vec{y}$, avoiding capture of any free variables in $\vec{y}$".
- In general we will take $f$ or $\vec{f}$ to stand for a fresh (vector) of variables.

Even though the motivating example uses separation logic, we will not concern ourselves with its peculiarities in the further examination, as we show in Section 5.2 that our rule allows local reasoning just like Tuerk's.

# 4 Predicate transformer semantics

## 4.1 Prerequisites

As stated in the introduction, we want to find a rule with predicate transformer semantics that will allow us to inductively verify a loop specified by how it changes the program state. The two transformers, *strongest postcondition* and *weakest precondition*, are dual, so we will first express ourselves only in *weakest precondition*, to avoid redundancy. The applicable rules are found in Figure 6.

Second, we will need the notion of refinement, and will need a way for “specifications”, that can also relate states, to take the place place of code for the purpose of verification condition generation, i. e., they must have a predicate transformer semantics.

Third we will need some refinement semantics for loops that will let us inductively verify they fulfill a specification.

$$\{P\}S\{Q\} := P \rightarrow wp\langle S \bullet Q\rangle \tag{1a}$$

Relation to Hoare-Triples

$$wp\langle skip \bullet R\rangle := R \tag{1b}$$

Skip

$$wp\langle p_1; p_2 \bullet R\rangle := wp\langle p_1 \bullet wp\langle p_2 \bullet R\rangle\rangle \tag{1c}$$

Sequential composition

$$wp\langle x := E \bullet R\rangle := \forall y.\ y = E \rightarrow R[x\backslash y] \tag{1d}$$

Assignment – Expanded

$$wp\langle x := E \bullet R\rangle := R[x\backslash E] \tag{1e}$$

Assignment

$$wp\langle \textbf{if } c \textbf{ then } p_1 \textbf{ else } p_2 \bullet R\rangle := c \wedge wp\langle p_1 \bullet R\rangle \ \vee\ \neg c \wedge wp\langle p_2 \bullet R\rangle \tag{1f}$$

Conditional

$$\begin{aligned} wp\langle \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2 \bullet R\rangle &:= c \wedge wp\langle p_1 \bullet wp\langle \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2 \bullet R\rangle\rangle \\ &\quad \vee \neg c \wedge wp\langle p_2 \bullet R\rangle \end{aligned} \tag{1g}$$

Loop-Unrolling

Figure 3: *wp*-Rules – Since *wp* is used inductively for rule (1g), we need to define what that means: We define an inductive predicate to be the smallest relation verifying the given clauses.

### Refinement

**Definition 4.1** (Refinement)**.** *An (abstract) program $P$ is* refined *by $Q$ (denoted $P \sqsubseteq Q$),* *iff Q satisfies any specification satisfied by P. More precisely:*

$$P \sqsubseteq Q :\Leftrightarrow \forall R.\ wp\langle P \bullet R\rangle \to wp\langle Q \bullet R\rangle$$

Refinement allows us to view implementations as fulfilling “more” than the specifications of a contract that they implement, seen itself as an abstract program. Conversely, it allows us to abstract away from an implementation to its contract, if only the contract is of interest to us. (The “more” specifications the implementation fulfills are the implementation details that are frequently not of interest)

**Theorem 4.1** (Monotonicity of Refinement)**.** *If $S[T]$ is a program with subprogram $T$, and $T \sqsubseteq T'$ then $S[T] \sqsubseteq S[T']$ (see [Mor88])*

Monotonicity is what makes refinement modular: We can switch out components of a program for components that fulfill more specifications, and retain a program that fulfills (now, more than) its original specifications.

**Specifications as programs** Morgan [Mor88] proposes a way to reason about specifications as programs: The specification statement. It is a statement of the form $\vec{w} : [pre, post]$ meaning: “A program that, given the condition *pre* holds, establishes condition *post*, while modifying variables $\vec{w}$”.
Crucially, he allows for relating the pre- and post-execution states using a modifier ($old()$) on variables in the postcondition which denotes they should be evaluated in the pre-execution state. He encodes all this in following semantics:

**Definition 4.2.**

$$wp\langle \vec{w}\!:[pre, post] \bullet R\rangle := pre \wedge (\forall \vec{w}.\ post[\overrightarrow{old(w)}\backslash\vec{f}] \to R)[\vec{f}\backslash\vec{w}]$$

**Lemma 4.2.** *If $R$ contains no $old()$-designated variables:*

$$wp\langle \vec{w}\!:[pre, post] \bullet R\rangle = pre \wedge (\forall \vec{w}.\ post \to R)[\overrightarrow{old(w)}\backslash\vec{w}]$$

Specification statements are extremely useful in their own right. For example, we can model an assignment $x := E$ as the specification statement $x\!:[true, x = E[x\backslash old(x)]]$. Or we can model a type of generalized assignment where we don’t have an equational relation describing the assigned value, e.g. $x : [x > 0]$.(We will from now on omit a precondition of *true*).

**Referring to initial state** Morgan doesn’t define a rule for $old()$ appearing in a general postcondition $R$ in $wp\langle p \bullet R\rangle$. Since we will need some defined behavior for this in the following, and since we find $old()$ practical to use in postconditions in general, we propose following modified weakest precondition transformer, written **wp**, which we allows us to reference the states of variables before execution of some program $p$:

**Definition 4.3** (Referring to initial state)**.**

$$\boxed{\begin{aligned} \mathbf{wp}\langle p \bullet R\rangle &:= \vec{x} = \vec{f} \rightarrow wp\langle p \bullet R[\vec{old(x)}\backslash\vec{f}]\rangle \\ \Leftrightarrow \mathbf{wp}\langle p \bullet R\rangle &:= wp\langle p \bullet R[\vec{old(x)}\backslash\vec{f}]\rangle[\vec{f}\backslash\vec{x}] \\ \Leftrightarrow \mathbf{wp}\langle p \bullet R\rangle &:= wp\langle p \bullet R\rangle[\vec{old(x)}\backslash\vec{x}] \end{aligned}}$$

*where $\vec{x}$ are all variables designated with $old()$ in $R$.*

We give an example of referring to initial state in this way with a sequence of assignments modeled as specification statements, in Figure 4.

Reference names for syntactic transformations:

$$\text{Rename bound variables to avoid capture} \qquad \text{(2a)}$$

$$\text{Perform Substitution} \qquad \text{(2b)}$$

Calculate $\mathbf{wp}\langle x := x + 2; x := x + 1 \bullet x = old(x) + 3\rangle$:

$$\begin{aligned}
&\mathbf{wp}\langle x\!:[x = old(x) + 2]; x\!:[x = old(x) + 1] \bullet x = old(x) + 3\rangle && \text{(4.3)}\\
\rightsquigarrow\ & wp\langle x\!:[x = old(x) + 2]; x\!:[x = old(x) + 1] \bullet x = f + 3\rangle[f\backslash x] && \text{(1c),(4.2)}\\
\rightsquigarrow\ & wp\langle x\!:[x = old(x) + 2] \bullet (\forall x.x = g + 1 \rightarrow x = f + 3)[g\backslash x]\rangle[f\backslash x] && \text{(2a),(2b)}
\end{aligned}$$

Here we have to rename $x$, as bound by the quantifier, so that the free $x$ will not be captured when we substitute it for $g$

$$\begin{aligned}
=\ & wp\langle x\!:[x = old(x) + 2] \bullet (\forall y.y = x + 1 \rightarrow y = f + 3)\rangle[f\backslash x] && \text{(4.2)}\\
\rightsquigarrow\ & (\forall x.x = g + 2 \rightarrow \forall y.y = x + 1 \rightarrow y = f + 3)[g\backslash x][f\backslash x] && \text{(2a),(2b)}\times 2\\
=\ & \forall z.z = x + 2 \rightarrow \forall y.y = z + 1 \rightarrow y = x + 3
\end{aligned}$$

Figure 4: At any step in the evaluation, $x$ is the current value of $x$. As soon as we step back, $old(x)$ becomes the current $x$ and $x$ must be renamed as it is now the $x$ in the following state.

**Refinement Semantics for loops** The standard least fixed-point semantics for loops states:

$$\begin{aligned}
\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2 \quad &= \quad \textbf{if } c \textbf{ then } p_1; \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2 \textbf{ else } p_2\\
W = \textbf{if } c \textbf{ then } p_1; W \textbf{ else } p_2 \quad &\rightarrow \quad W \sqsupseteq \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2
\end{aligned}$$

Hehner [HG99] proposes following alternative refinement semantics:

$$W \sqsubseteq \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2 := \ W \sqsubseteq \textbf{if } c \textbf{ then } p_1; W \textbf{ else } p_2$$

That means, if we want to prove that a loop refines some program $W$, we can use $W$ instead of the loop after one unfolding[1]. This corresponds in essence to the Recursive Invocation Rule [Hoa71, p. 109] for verifying procedures. Note that these semantics can only be used to prove partial correctness. A *variant* must be provided to prove termination, if total correctness is to be shown.

We can use this semantics, in particular, to show that a loop fulfills some specification $[\mathit{pre}, \mathit{post}]$, by taking as $W$ $\vec{w}\colon[\mathit{pre}, \mathit{post}]$, where $\vec{w}$ are the variables modified by the loop.

## 4.2 Loop specification rule for weakest precondition

We want to define $\mathit{wp}\langle[\mathit{pre}]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[\mathit{post}] \bullet R\rangle$, that is, "The weakest precondition so that our loop, which we assume satisfies *post* given *pre* (modifying variables $\vec{w}$), fulfills $R$". That means, for verification we let $\vec{w}\colon[\mathit{pre}, \mathit{post}]$ take the place of the loop. The monotonicity law of refinement (4.1) then requires that the loop refine $\vec{w}\colon[\mathit{pre}, \mathit{post}]$. The first version of the rule therefore states:

$$\begin{aligned}
&\mathit{wp}\langle[\mathit{pre}]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[\mathit{post}] \bullet R\rangle := \\
&\qquad \mathit{wp}\langle\vec{w}\colon[\mathit{pre}, \mathit{post}] \bullet R\rangle \wedge \\
&\qquad \vec{w}\colon[\mathit{pre}, \mathit{post}] \sqsubseteq \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2
\end{aligned}$$

We propose following premises for the refinement condition, which we prove to be sufficient in Section 4.3:

$$\begin{aligned}
&\forall\vec{v}.\ \mathit{pre} \to \mathbf{wp}\langle\textbf{if } c \textbf{ then } p_1; \vec{w}\colon[\mathit{pre}, \mathit{post}] \textbf{ else } p_2 \bullet \mathit{post}\rangle \\
\Leftrightarrow\ &\forall\vec{v}.\ \mathit{pre} \wedge \neg c \to \mathit{wp}\langle p_2 \bullet \mathit{post}\rangle[\overrightarrow{old(w)}\backslash\vec{w}] && \text{Induction Base} \\
\wedge &\forall\vec{v}.\ \mathit{pre} \wedge c \to \mathit{wp}\langle p_1; \vec{w}\colon[\mathit{pre}, \mathit{post}] \bullet \mathit{post}\rangle[\overrightarrow{old(w)}\backslash\vec{w}] && \text{Inductive Step}
\end{aligned}$$

(where $\vec{v}$ are all free variables in $p_1, p_2$.)

Our proposed rule is therefore:

$$\boxed{\begin{aligned}
&\mathit{wp}\langle[\mathit{pre}]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[\mathit{post}] \bullet R\rangle := \\
&\qquad \mathit{wp}\langle\vec{w}\colon[\mathit{pre}, \mathit{post}] \bullet R\rangle \wedge \\
&\qquad \forall\vec{v}.\ \mathit{pre} \wedge \neg c \to \mathit{wp}\langle p_2 \bullet \mathit{post}\rangle[\overrightarrow{old(w)}\backslash\vec{w}] \wedge && \text{Induction Base} \\
&\qquad \forall\vec{v}.\ \mathit{pre} \wedge c \to \mathit{wp}\langle p_1; \vec{w}\colon[\mathit{pre}, \mathit{post}] \bullet \mathit{post}\rangle[\overrightarrow{old(w)}\backslash\vec{w}] && \text{Inductive Step}
\end{aligned}} \tag{3}$$

where $\vec{w}$ are all variables modified by, and $\vec{v}$ all variables free in $p_1, p_2$.

## 4.3 Soundness proof

For this proof we will need following lemma [Dij76]:

[1] We could not do this with fixed-point semantics, since using $W$ there would require $W \sqsupseteq \textbf{while}[\ldots]$, due to monotonicity

**Lemma 4.3** (Monotonicity of *wp*)**.** $(R \to S) \to wp\langle p \bullet R\rangle \to wp\langle p \bullet S\rangle$

**Theorem 4.4** (Soundness of Contract While-Rule)**.** *Let* ***while*** $c$ ***do*** $p_1$ ***done*** $p_2$ *be a loop,* $\vec{w}$ *all the variables modified by,* $\vec{v}$ *all free in* $p_1, p_2$*. Then* $\vec{w} : [\mathit{pre}, \mathit{post}] \sqsubseteq$ ***while*** $c$ ***do*** $p_1$ ***done*** $p_2$ *if* (Induction Base) *and* (Inductive Step) *hold, where:*

$$\forall \vec{v}.\ \mathit{pre} \wedge \neg c \to wp\langle p_2 \bullet \mathit{post}\rangle [\vec{old(w)} \backslash \vec{w}] \qquad \text{(Induction Base)}$$

$$\forall \vec{v}.\ \mathit{pre} \wedge c \to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet \mathit{post}\rangle [\vec{old(w)} \backslash \vec{w}] \qquad \text{(Inductive Step)}$$

*Proof.*

$$\vec{w} : [\mathit{pre}, \mathit{post}] \sqsubseteq \textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2$$

(loop refinement semantics)

$$\Leftrightarrow \vec{w} : [\mathit{pre}, \mathit{post}] \sqsubseteq \textbf{if } c \textbf{ then } p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \textbf{ else } p_2$$

(definition of refinement (4.1))

$$\Leftrightarrow \forall R.\ wp\langle \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle \to wp\langle \textbf{if } c \textbf{ then } p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \textbf{ else } p_2 \bullet R\rangle$$

(condidional rule (1f))

$$\Leftrightarrow \forall R.\ wp\langle \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle \to (c \wedge wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle \ \vee\ \neg c \wedge wp\langle p_2 \bullet R\rangle)$$

(equational rearrangement, choose arbitrary $R$)

$$\Leftrightarrow\ ((c \wedge wp\langle \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle) \to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle)$$
$$\wedge ((\neg c \wedge wp\langle \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle) \to wp\langle p_2 \bullet R\rangle)$$

(Lemma 4.2)

$$\Leftrightarrow\ \Big(\Big(c \wedge \mathit{pre} \wedge (\forall \vec{w}.\ \mathit{post} \to R)\, [\vec{old(w)} \backslash \vec{w}]\Big) \to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle\Big)$$
$$\wedge \Big(\Big(\neg c \wedge \mathit{pre} \wedge (\forall \vec{w}.\ \mathit{post} \to R)\, [\vec{old(w)} \backslash \vec{w}]\Big) \to wp\langle p_2 \bullet R\rangle\Big)$$

(use (Induction Base),(Inductive Step))

$$\Leftrightarrow\ \Big(\Big(wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet \mathit{post}\rangle [\vec{old(w)} \backslash \vec{w}] \wedge (\forall \vec{w}.\ \mathit{post} \to R)\, [\vec{old(w)} \backslash \vec{w}]\Big)$$
$$\to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle\Big)$$
$$\wedge \Big(\Big(wp\langle p_2 \bullet \mathit{post}\rangle [\vec{old(w)} \backslash \vec{w}] \wedge (\forall \vec{w}.\ \mathit{post} \to R)\, [\vec{old(w)} \backslash \vec{w}]\Big) \to wp\langle p_2 \bullet R\rangle\Big)$$

(unify substitutions)

$$\Leftrightarrow\ ((wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet \mathit{post}\rangle \wedge (\forall \vec{w}.\ \mathit{post} \to R)) \to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle)\, [\vec{old(w)} \backslash \vec{w}]$$
$$\wedge ((wp\langle p_2 \bullet \mathit{post}\rangle \wedge (\forall \vec{w}.\ \mathit{post} \to R)) \to wp\langle p_2 \bullet R\rangle)\, [\vec{old(w)} \backslash \vec{w}]$$

(use Lemma (4.3), given that $p_1, p_2$ modify only variables in $\vec{w}$, per definition)

$$\Rightarrow\ (wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle \to wp\langle p_1; \vec{w} : [\mathit{pre}, \mathit{post}] \bullet R\rangle)\, [\vec{old(w)} \backslash \vec{w}]$$
$$\wedge (wp\langle p_2 \bullet R\rangle \to wp\langle p_2 \bullet R\rangle)\, [\vec{old(w)} \backslash \vec{w}]$$

□

## 4.4 Context-aware rule

Previously we showed refinement independently of context – that is, $pre$ has to suffice as precondition for the inductive base & step. We forced independence from context by quantifying over all variables $\vec{v}$ free in $p_1, p_2$.

We can always strengthen the precondition from the context though: For the variables modified in $p_1$,$\vec{w_1}$, we must assume an arbitrary value – all others remain unchanged by each iteration of the loop.

$$\begin{aligned}
&wp\langle [pre]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[post] \bullet R\rangle := \\
&\qquad wp\langle \vec{w}\!:[pre, post] \bullet R\rangle \wedge \\
&\qquad \forall \vec{w_1}.\ pre \wedge \neg c \rightarrow wp\langle p_2 \bullet post\rangle[\overrightarrow{old(w)}\backslash\vec{w}] \wedge && \text{Induction Base} \\
&\qquad \forall \vec{w_1}.\ pre \wedge c \rightarrow wp\langle p_1; \vec{w}\!:[pre, post] \bullet post\rangle[\overrightarrow{old(w)}\backslash\vec{w}] && \text{Inductive Step} \\
&\qquad\quad \text{where } \vec{w} \text{ are all variables modified by } p_1, p_2 \text{ and } \vec{w_1} \text{ by } p_1.
\end{aligned} \tag{5}$$

This version of the rule is practical if we want to avoid having to state a lot of information from the context in the precondition of the loop. E.g. it would allow following program to be verified: $(i := 42; [true]\textbf{while } false \textbf{ do } skip \textbf{ done } j := i[j = 42])$, whereas the other would not. It is a matter of taste which to use – the first is more modular, while this might spare one needless repetition.

## 4.5 Rules in strongest postcondition

For completeness's sake, and because we will need them later, as our implementation is done on the basis of symbolic execution, we introduce the counterparts of the rules we used/defined for weakest precondition in strongest postcondition.

**Rules Used** We first note some of the standard $sp$ Rules:

$$\{P\}S\{Q\} := sp\langle P \bullet S\rangle \rightarrow Q \tag{6a}$$

Relation to Hoare-Triples

$$sp\langle P \bullet p_1; p_2\rangle := sp\langle sp\langle P \bullet p_1\rangle \bullet p_2\rangle \tag{6b}$$

Sequential composition

$$sp\langle P \bullet \textbf{if } c \textbf{ then } p_1 \textbf{ else } p_2\rangle := (sp\langle c \wedge P \bullet p_1\rangle) \vee (sp\langle \neg c \wedge P \bullet p_2\rangle) \tag{6c}$$

Conditional

Figure 5: $sp$-Rules

**Specification statement** Morgan does not define a *sp*-rule for the specification statement, since it is not needed for theoretical considerations and is readily derived from the *wp*-rule, again due to the duality of the two. We define the rule here:

$$\boxed{sp\langle P \bullet \vec{w}\colon [pre, post]\rangle := (P \rightarrow pre) \rightarrow (\exists \vec{f}.\ P[\vec{w}\backslash\vec{f}] \wedge post[\vec{old(w)}\backslash\vec{f}])} \tag{7}$$

As we can't add *pre* to the precondition as in the *wp*-rule, we make $P \rightarrow pre$ a precondition of the rest of the postcondition. Only have we proven it may we assume $\exists \vec{f}.\ P[\vec{w}\backslash\vec{f}] \wedge post[\vec{old(w)}\backslash\vec{f}]$.

### While-Contract Rule

$$\boxed{\begin{aligned} & sp\langle P \bullet [pre]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[post]\rangle := \\ & \qquad\qquad sp\langle P \bullet \vec{w}\colon [pre, post]\rangle \wedge \\ & \qquad\qquad sp\langle pre \wedge \neg c \wedge \vec{w} = \vec{f} \bullet p_2\rangle \rightarrow post[\vec{old(w)}\backslash\vec{f}] \wedge && \text{Base} \\ & \qquad\qquad sp\langle pre \wedge c \wedge \vec{w} = \vec{f} \bullet p_1; \vec{w}\colon [pre, post]\rangle \rightarrow post[\vec{old(w)}\backslash\vec{f}] && \text{Step} \end{aligned}} \tag{8}$$

We need to add $\vec{w} = \vec{f}$ to the precondition to "capture" the values of $\vec{w}$ before execution of $p_1, p_2$ respectively.

The context-aware version of the rule is:

$$\boxed{\begin{aligned} & sp\langle P \bullet [pre]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[post]\rangle := \\ & \qquad\qquad sp\langle P \bullet \vec{w}\colon [pre, post]\rangle \wedge \\ & \qquad\qquad sp\langle P[\vec{w_1}\backslash\vec{g}] \wedge pre \wedge \neg c \wedge \vec{w} = \vec{f} \bullet p_2\rangle \rightarrow post[\vec{old(w)}\backslash\vec{f}] \wedge \\ & \qquad\qquad sp\langle P[\vec{w_1}\backslash\vec{g}] \wedge pre \wedge c \wedge \vec{w} = \vec{f} \bullet p_1; \vec{w}\colon [pre, post]\rangle \rightarrow post[\vec{old(w)}\backslash\vec{f}] \end{aligned}} \tag{9}$$

where $w_1$ are the variables modified by $p_1$ and $\vec{g}$ is a fresh vector of variables.

# 5 Rules in Hoare Logic

Having stated our rule in predicate transformer semantics, we want to state them in the alternative and perhaps more familiar form of axiomatic rules in Hoare logic.

## 5.1 Preliminaries

The standard Hoare-logic rules we will refer to can be found in Figure 6.

$$\frac{}{\{Q\}skip\{Q\}}\ \textsc{Skip} \tag{10a}$$

$$\frac{\{P\}p_1\{R\} \quad \{R\}p_2\{Q\}}{\{P\}p_1; p_2\{Q\}}\ \textsc{Seq} \tag{10b}$$

$$\frac{\{P \wedge c\}p_1 Q \quad \{P \wedge \neg p_2\{Q\}\}}{\{P\}\textbf{if } c \textbf{ then } p_1 \textbf{ else } p_2\{Q\}}\ \textsc{Cond} \tag{10c}$$

$$\frac{}{\{Q[x\backslash E]\}x := E\{Q\}}\ \textsc{Assign} \tag{10d}$$

$$\frac{P \rightarrow P' \quad \{P'\}p\{Q'\} \quad Q' \rightarrow Q}{\{P\}p\{Q\}}\ \textsc{Conseq} \tag{10e}$$

Figure 6: Hoare-Logic Rules

Since Morgan does not define Hoare Logic rules for the specification statement, we will first define them, as well as a notation that will allow us to use *old*() in postconditions. We start with *old*(): We introduce a modified notation:

$$\frac{\{P \wedge \vec{w} = \vec{f}\}S\{Q[\vec{old(w)}\backslash \vec{f}]\}}{\llparenthesis P \rrparenthesis S \llparenthesis Q \rrparenthesis}\ \textsc{Old} \tag{11}$$

Such triples are useful anywhere we want to use *old*() in the postcondition, e.g. when $S$ is the body of a procedure and $P$, $Q$ its contract. They do the job of introducing shadow variables for referencing the initial value of program variables.

The rule for the specification statement is:

$$\frac{P \rightarrow pre \qquad P[\vec{w}\backslash \vec{f}] \wedge post[\vec{old(w)}\backslash \vec{f}] \rightarrow Q}{\{P\}\vec{w}\colon[pre, post]\{Q\}}\ \textsc{Spec} \tag{12}$$

The fresh variables $\vec{f}$ replacing $\vec{w}$ represent the original values of those variables, before their indefinite modification by the statement. The postcondition may still talk about $\vec{w}$, possibly relating them to $\vec{old(w)}$ – now $\vec{f}$.

To gain some more confidence in this rule, we use it to derive the assignment rule (10d), using the usual translation of assignment to specification statement, in Figure 7.

$$\dfrac{\dfrac{}{Q[x\backslash E] \to true} \qquad \dfrac{\dfrac{}{Q[x\backslash E[x\backslash f]] \land x = E[x\backslash f] \to Q}}{Q[x\backslash E][x\backslash f] \land x = E[x\backslash old(x)][old(x)\backslash f] \to Q}\ \text{Subst. comp}}{\{Q[x\backslash E]\}x\colon[true, x = E[x\backslash old(x)]]\{Q\}}\ \textsc{Spec}$$

Figure 7: Compatibility of assignment with Spec rule

## 5.2 Loop specification rule in Hoare logic

The Hoare logic version has three parts again (we refer to them as Use, Base, Step).

$$\boxed{\dfrac{\begin{array}{c}\{P\}\vec{w}\colon[pre, post]\{Q\} \\ \llparenthesis pre \land c\rrparenthesis p_1; \vec{w}\colon[pre, post]\llparenthesis post\rrparenthesis \qquad \llparenthesis pre \land \neg c\rrparenthesis p_2\llparenthesis post\rrparenthesis\end{array}}{\{P\}[pre]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[post]\{Q\}}\ \textsc{While-Contr}} \tag{13}$$

where $\vec{w}$ are the variables modified by $p_1, p_2$.

We now see why local reasoning can be used in the inductive step: The general frame rule of separation logic states that:

$$\dfrac{\{P\}S\{Q\}}{\{F * P\}S\{F * Q\}} \tag{14}$$

if $S$ does not modify any variable in $F$. We know $\vec{w}\colon[pre, post]$ modifies only variables $\vec{w}$, so we can frame out unrelated frames. In the loop rule the opportunity to frame out arises in the inductive step where the specification statement is used as the "induction hypothesis".

**Context-aware rule** As in predicate transformer semantics, we can define a version of the rule that uses the context to strengthen the precondition in the inductive proof.

$$\boxed{\dfrac{\begin{array}{c}\{P\}\vec{w}\colon[pre, post]\{Q\} \\ \llparenthesis P[\vec{w_1}\backslash\vec{f}] \land pre \land c\rrparenthesis p_1; \vec{w}\colon[pre, post]\llparenthesis post\rrparenthesis \qquad \llparenthesis P[\vec{w_1}\backslash\vec{f}] \land pre \land \neg c\rrparenthesis p_2\llparenthesis post\rrparenthesis\end{array}}{\{P\}[pre]\textbf{while } c \textbf{ do } p_1 \textbf{ done } p_2[post]\{Q\}}} \tag{15}$$

where $\vec{w}$ are the variables modified by $p_1, p_2$, $\vec{w_1}$ by $p_1$.

# 6 Implementation

$$\frac{\{P \land c\}p_1; rQ \quad \{P \land \neg p_2\}; r\{Q\}}{\{P\}\textbf{if } c \textbf{ then } p_1 \textbf{ else } p_2; r\{Q\}} \textsc{Cond-SE} \tag{16a}$$

$$\frac{\{x = E[x\backslash f] \land P[x\backslash f]\} r\{Q\}}{\{P\} x := E; r\{Q\}} \textsc{Assign-SE} \tag{16b}$$

Figure 8: Hoare Rules for Symbolic Execution

We implemented the context-aware version of the rule, in a tool for verification condition generation using symbolic execution, for programs in a DSL (Domain-Specific Language) with the features described in Section 3. The implementation is in `Scala`.

The symbolic execution is done following the rules of, essentially, strongest postcondition, but interpreted as Hoare rules (examples in Figure 8), meaning we always have a postcondition in mind. This makes it simpler to deal with *old*().

We present the code in slightly simplified form to abstract away inessential details, in particular the parts concerned with separation logic.

The signature of the method that does the heavy lifting is:

```
def _hoare(pre: Expr, st0: Subst, st1: Subst, progs: List[Prog], post: Expr)
```

- `Expr` is the type for expressions, in both the DSL and the assertions.
- `Var` is the type for program- and symbolic variables.
- `Subst` is just a type synonym for `Map[Var, Expr]`, which maps program variables to their symbolic values in a particular state.
- `Prog` is the type of an instruction in the Language.

`pre` and `post` are both `Expr`s of type `Type.bool`. `pre` never contains free occurrences of program variables, only symbolic variables that represent the value of program variables at a particular state in the execution (compare Figure 4). `post` on the other hand knows nothing about symbolic variables and predicates only over program variables.

`st1` is the current symbolic state of the program, which is updated as execution progresses. `st0` is the state before execution, which can be referred to in the postcondition by *old*()-designated variables.

**Treatment of *old*()** `_hoare` is defined as a pattern match over `progs`. We examine the case where `progs == Nil` (that is, there are no remaining instructions). These cases correspond to the branches of our VCG tree. Let's look at the code:

```
case Nil =>
   val _post = post eval (st0, st1)
   prove(pre ==> _post)
```

`prove` can be configured to either pass the argument, which should be an $Expr_{Bool}$, to an SMT-Solver (we use Z3), or print it to the console.

`eval` is defined for all subtypes of `Expr` and replaces program variables with their symbolic values. For `Old(expr: Expr)` it is defined as:

```
def eval(st0: Subst, st1: Subst) = expr eval st0
```

That is, the expression `expr` is evaluated in state `st0` using the overloaded `eval` method with a single parameter. The definition for `Var(…)` is:

```
def eval(st0: Subst, st1: Subst) = st1 (this)
def eval(st: Subst) = st (this)
```

That is, we return the symbolic value of `this` in the current state.

Since `pre` only predicates about the symbolic values of variables, this implements our rule for *old*() (11). The implementation ensures that `st0` is the correct state.

**Specification statement** Let's consider how we handle the specification statement.

```
0: case Spec(spre, mod, spost) :: rest =>
1:   val _spre = spre eval (st0,st1)
2:   val re = Ren.fresh(mod)
3:   val st2 = st1 ++ re
4:   val _spost = spost eval (st1,st2)
5:   prove(pre ==> _spre)
6:   hoare(_spost && pre, st0, st2, rest, post)
```

Since our implementation essentially does SE, we would do best to compare this rule to our definition for *sp* in Section 4.5.

1. we evaluate SPEC's precondition in the current state, making it predicate no longer over program variables, but their symbolic values.

2. We generate fresh variables as symbolic values for the variables `mod` modified by SPEC.

3. We define define a new state `st2`, after execution of SPEC, where we overwrite the symbolic values of `mod` in `st1` with the fresh variables.

4. We evaluate `spost` in the state `st2` with *old*() state `st1`, to add it to the precondition in the following.

5. To assume `_spost` and `pre`, we have to prove that SPEC's precondition is met in `st1`. We don't have to perform any substitution on `pre` like in the rule because `pre` talks only about symbolic values, not program variables.

6. We continue with the new precondition and `rest`, as the rule for *sp* of sequential composition (6b) states. Note that `st1` was only locally the *old*() state, for evaluation of `spost`. The `&&` here is not `Scala`'s `&&`, but overloaded as an Operator in the DSL.

### While-Rule

```
0: case WhileContract(lpre, While(c, p1), p2, lpost) :: rest =>
1:   val spec = Spec(lpre, p1.mod ++ p2.mod, lpost)
2:   val re = Ren.fresh(p1.mod)
3:   val st2 = st1 ++ re
4:   val _c = c eval st2
5:   val _lpre = lpre eval (st0,st2)
6:   hoare(pre, st0, st1, spec :: rest, post)
7:   hoare(_c && _lpre && pre, st2, st2, p1 ++ List(spec), lpost)
8:   hoare(!_c && _lpre && pre, st2, st2, p2, lpost)
```

1. We generate the specification that will replace the loop in both the inductive proof, as the inductive hypothesis, and the the program. `mod` gives us the variables modified by a program.

3. We define a new state `st2`, in which we will verify the induction base & step.

6. We continue SE of the program , replacing the loop with the specification we now have to prove it refines.

7. We prove the inductive step. The fact that we can just append `spec` to `p1` and treat it as a normal program is what makes our rule so advantageous.

With so many states at play, a bit of a clarification might be in order. We will look at the inductive step part of the proof and apply the specification statement. We have three states to consider: I before execution of the loop body, II after the loop body and III after the remaining iterations and $p_2$, represented by the specification statement:

$$\mathrm{I}\ p_1;\ \mathrm{II}\ \vec{w} : [pre, post]\ \mathrm{III}$$

Using the *sp*-while rule (8), the proof obligation for the inductive step is:

$$sp\langle pre \wedge c \wedge \vec{w} = \vec{f} \bullet p_1; \vec{w} : [pre, post]\rangle \to post[\overrightarrow{old(w)} \backslash \vec{f}]$$

In the implementation, referencing the old state is done with maps, not fresh introduction of variables. We therefore now want to consider only the state in which each assertion should be evaluated, without performing substitutions. We mark the current state in the subscript, the old state, if applicable, in the superscript. Thus expanding the application of the Spec rule we get:

$$\text{let } mid = sp\langle (pre \wedge c)_{\mathrm{I}} \bullet p_1\rangle \text{ in: } ((mid_{\mathrm{II}} \to pre_{\mathrm{II}}) \to mid_{\mathrm{II}} \wedge post^{\mathrm{II}}_{\mathrm{III}}) \to post^{\mathrm{I}}_{\mathrm{III}}$$

# 7 Example Application to Textbook Algorithms

We will specify the iterative versions of two popular textbook algorithms, fast exponentiation and Euclid's algorithm, annotating the loops with contracts.We will find that we can easily derive the invariants, which will make them seem to be quite a roundabout way of doing things.

## 7.1 Euclid's algorithm

Following code is an iterative implementation of Euclid's algorithm for finding the greatest common divisor (gcd) of two integers:

```
[pre: m>=0,n>=0]
def euclid(m,n):
   while m != n:
      if m > n: m = m-n
      else:     n = n-m
   return n
[post:n=gcd(old(m),old(n))]
```

Since the program consists of essentially only the loop, we have no choice but to annotate it thus:

```
[pre: m>=0,n>=0]
def euclid(m,n):
   [pre: m>=0,n>=0]
   while m != n:
      if m > n: m = m-n
      else:     n = n-m
   [post:n=gcd(old(m),old(n))]
   return n
[post:n=gcd(old(m),old(n))]
```

We prove this specification using the Hoare rules for symbolic execution 8, in Figure 10. We need some properties of the gcd for the proof, which we state as axioms in Figure 9.

$$\operatorname{gcd}(m,m) = m \tag{17a}$$

$$\operatorname{gcd}(m,n) = \operatorname{gcd}(n,m) \tag{17b}$$

$$m \geq n \to \operatorname{gcd}(m,n) = \operatorname{gcd}(m-n,n) \tag{17c}$$

Figure 9: gcd Axioms

We use $_0$ instead of $old()$ to mark old variables in this proof.

$$W := \texttt{while m != n: } B$$

$$B := \texttt{if m>n: m = m-n else: n=n-m}$$

$$\dfrac{\dfrac{\textsc{Use}\quad\textsc{Base}\quad\textsc{Step}}{\{a \geq 0, b \geq 0, m = a, n = b\} W \{n = \gcd(a, b)\}}\ \textsc{While-Contr}}{\llparenthesis m \geq 0, n \geq 0 \rrparenthesis W \llparenthesis n = \gcd(m_0, n_0)\rrparenthesis}\ \textsc{Old}$$

Use:

$$\dfrac{\dfrac{}{\{a, b \geq 0, m = a, n = b\} \to m, n \geq 0}\quad \dfrac{\dfrac{}{(a, b \geq 0, c = a, d = b) \wedge (n = \gcd(c, d)) \to n = \gcd(a, b)}}{(a, b \geq 0, m = a, n = b)[m, n\backslash c, d] \wedge (n = \gcd(m_0, n_0))[m_0, n_0\backslash c, d] \to n = \gcd(a, b)}\ \text{Subst}}{\{a, b \geq 0, m = a, n = b\} n, m\colon [m, n \geq 0, n = \gcd(m_0, n_0)]\{n = \gcd(a, b)\}}\ \textsc{Spec} \tag{18a}$$

Base:

$$\dfrac{\dfrac{\dfrac{}{e = f, n = f \to n = \gcd(e, f)}\ (17a)}{\{e, f \geq 0, e = f, m = e, n = f\}\mathit{skip}\{n = \gcd(e, f)\}}\ \textsc{Skip}}{\llparenthesis m, n \geq 0, \wedge m = n\rrparenthesis \mathit{skip} \llparenthesis n = \gcd(m_0, n_0)\rrparenthesis}\ \text{Subst}, \textsc{Old} \tag{18b}$$

Step:

$$\dfrac{\dfrac{\textsc{Left}\quad\textsc{Right}}{\{e, f \geq 0, e \neq f, m = e, n = f\}\texttt{if m>n: m = m-n else: n=n-m}; m, n\colon [m \wedge n \geq 0, n = \gcd(m_0, n_0)]\{n = \gcd(e, f)\}}\ \textsc{Cond-SE}}{\llparenthesis m, n \geq 0, \wedge m \neq n\rrparenthesis P; m, n\colon [m \wedge n \geq 0, n = \gcd(m_0, n_0)]\llparenthesis n = \gcd(m_0, n_0)\rrparenthesis}\ \text{Subst}, \textsc{Old} \tag{18c}$$

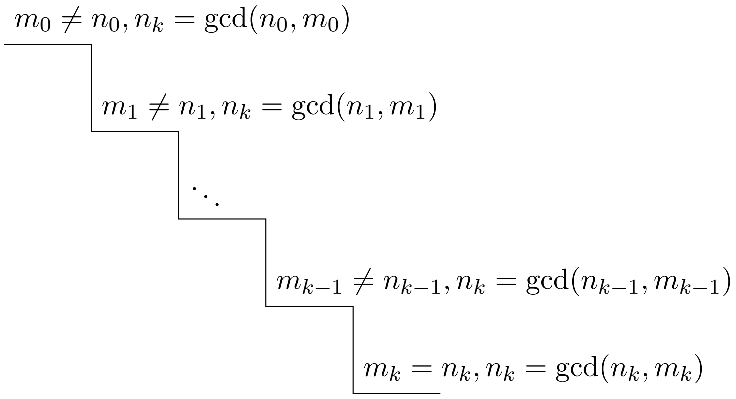
Left:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{n = f \geq 0}\quad\dfrac{}{g = e \geq 0, g > n \to m = g - n \geq 0}}{e, f \geq 0, g = e, n = f, g > n, m = g - n \to m, n \geq 0}\quad \dfrac{\dfrac{\dfrac{}{e, f \geq 0, e = g > i = f, n = \gcd(e - f, f) \to n = \gcd(e, f)}\ (17c)}{e, f \geq 0, e \neq f, g = e, i = f, g > i, h = g - i, n = \gcd(h, i) \to n = \gcd(e, f)}\ =}{(e, f \geq 0, e \neq f, g = e, n = f, g > n, m = g - n)[m, n\backslash h, i] \wedge (n = \gcd(m_0, n_0))[m_0, n_0\backslash h, i] \to n = \gcd(e, f)}\ \text{Subst}}{\{e, f \geq 0, e \neq f, g = e, n = f, g > n, m = g - n\} m, n\colon [m \wedge n \geq 0, n = \gcd(m_0, n_0)]\{n = \gcd(e, f)\}}\ \textsc{Spec}}{\{e, f \geq 0, e \neq f, m = e, n = f, m > n\}\texttt{m = m-n}; m, n\colon [m \wedge n \geq 0, n = \gcd(m_0, n_0)]\{n = \gcd(e, f)\}}\ \textsc{Assgt-SE} \tag{18d}$$

Right: proof analogous to Left

Figure 10: Euclid's algorithm specification proof

**Deducing the Invariant** We can deduce the invariant from the specification. First off, the precondition will always be part of the invariant, as it is already treated like one in the way it is verified in all the contract-loop rules. Second, we will see that in some cases, the invariant is quite easy to deduce from the postcondition. Let's return to our concrete example. We will first take `old(old())` to refer to the values of variables before any iteration. Now we know that `n=gcd(old(old(n)),old(old(m)))` must hold at the end of iteration. With the way the inductive proof works, this is the actually the last result proven, after having proved the corresponding instances for all intermediary results. But since `n` in the postcondition always refers to the value of `n` at the end of *all* iterations, all the intermediate results share this connection → an invariant. We substitute `gcd(old(old(n)),old(old(m)))` for this `n` and get `gcd(old(old(n)),old(old(m))=gcd(old(n),old(m))`.


$m_0 \neq n_0, n_k = \text{gcd}(n_0, m_0)$
$m_1 \neq n_1, n_k = \text{gcd}(n_1, m_1)$
$\ddots$
$m_{k-1} \neq n_{k-1}, n_k = \text{gcd}(n_{k-1}, m_{k-1})$
$m_k = n_k, n_k = \text{gcd}(n_k, m_k)$

Since it doesn't matter in the invariant if we refer to variables at the start or end of an iteration, we can define `old()` to always refer to the state before *any* iteration in invariants, and get rid of `old(old())`: `gcd(old(n),old(m))=gcd(n,m)`. Together with the part of the invariant that is the precondition we get:

```
[pre:m>=0,n>=0]
def euclid(m,n):
   while m != n:
   [inv: m>=0,n>=0, gcd(old(m),old(n))=gcd(m,n)]
      if m > n: m = m-n
      else:     n = n-m
   return n
[post:n=gcd(old(m),old(n))]
```

The fact that we're specifying the state of *incomplete computations* forces us to view all iterations at once, instead of inductively. We will look at another algorithm, *fast exponentiation*, and make a very similar observation.

## 7.2 Fast Exponentiation

Following code is an iterative implementation of exponentiation by squaring (fast exponentiation):

```
[pre: i>=0]
def fastExp(x : Real,i : Int):
   r = 1
   while i != 0:
      if i odd: r = x*r
      x=x*x
      i=i div 2
   return r
[post:r=old(x)^old(i)]
```

The algorithm is based on the equalities: $x^{2n} = (x^2)^n, x^{2n+1} = x \cdot (x^2)^n$. We choose as precondition $i \geq 0$, as postcondition $r = \mathit{old}(r) \cdot \mathit{old}(x)^{\mathit{old}(i)}$. We prove using the while-rule for strongest postcondition (8) that the loop fulfills this specification:

IB Show $(r = r_0, x = x_0 \wedge i = i_0 \wedge i \geq 0 \wedge i = 0) \rightarrow (r = r_0 \cdot x_0^{i_0})$:
Assume $(r = r_0, x = x_0 \wedge i = i_0 \wedge i \geq 0 \wedge i = 0)$.
Then $r_0 \cdot x_0^{i_0} = r \cdot x_0^{i} = r \cdot x_0^{0} = r \cdot 1 = r$ $\quad\square$

IB Show $\mathit{sp}\langle r = r_0 \wedge x = x_0 \wedge i = i_0 \wedge i \geq 0 \wedge i \neq 0 \bullet \textbf{if } i \text{ odd } \textbf{then } r := x \cdot r; x = x \cdot x; i = i \text{ div } 2; (r, x, i)\colon[i \geq 0, r = \mathit{old}(r) \cdot \mathit{old}(x)^{\mathit{old}(i)}]\rangle \rightarrow r = r_0 \cdot x_0^{i_0}$:

$$\begin{aligned}
&\mathit{sp}\langle r = r_0 \wedge x = x_0 \wedge i = i_0 \wedge i > 0 \bullet \textbf{if } i \text{ odd } \textbf{then } r := x \cdot r\rangle = \\
&\quad (x = x_0 \wedge i = i_0 \wedge i > 0 \wedge \exists k \in \mathbb{N}.\ i = 2k+1, r = x \cdot r_0) \\
&\quad \vee (r = r_0 \wedge x = x_0 \wedge i = i_0 \wedge i > 0 \wedge \exists k \in \mathbb{N}.\ i = 2k)
\end{aligned}$$

$i$ odd:

$$\begin{aligned}
&\mathit{sp}\langle (x = x_0 \wedge i = i_0 \wedge i > 0 \wedge \exists k \in \mathbb{N}.\ i = 2k+1 \wedge r = x \cdot r_0) \bullet \\
&\quad x := x \cdot x; i = i \text{ div } 2\rangle \rightarrow \\
&(\exists k \in \mathbb{N}.\ i_0 = 2k+1 \wedge r = x_0 \cdot r_0 \wedge x = x_0 \cdot x_0 \wedge i = k)
\end{aligned}$$

$$\begin{aligned}
&\mathit{sp}\langle (\exists k \in \mathbb{N}.\ i_0 = 2k+1 \wedge r = x_0 \cdot r_0 \wedge x = x_0 \cdot x_0 \wedge i = k) \bullet \\
&\quad (r, x, i)\colon[i \geq 0, r = \mathit{old}(r) \cdot \mathit{old}(x)^{\mathit{old}(i)}]\rangle \rightarrow \\
&((\exists k \in \mathbb{N}.\ i = k \rightarrow i \geq 0) \rightarrow \\
&\quad ((\exists k \in \mathbb{N}.\ i_0 = 2k+1 \wedge r_1 = x_0 \cdot r_0 \wedge x_1 = x_0 \cdot x_0 \wedge i_1 = k) \wedge (r = r_1 \cdot x_1^{i_1}))) \rightarrow \\
&(r = x_0 \cdot r_0 \cdot (x_0 \cdot x_0)^{(i_0-1)/2}) = \\
&(r = r_0 \cdot x_0 \cdot x_0^{i_0-1}) = \\
&(r = r_0 \cdot x_0^{i_0}) \qquad \square
\end{aligned}$$

$i$ even:

$$\begin{aligned}
&sp\langle (r = r_0 \land x = x_0 \land i = i_0 \land i > 0 \land \exists k \in \mathbb{N}.\ i = 2k) \bullet \\
&\quad x := x \cdot x; i = i \text{ div } 2\rangle \rightarrow \\
&(\exists k \in \mathbb{N}.\ i_0 = 2k \land r = r_0 \land x = x_0 \cdot x_0 \land i = k)
\end{aligned}$$

$$\begin{aligned}
&sp\langle (\exists k \in \mathbb{N}.\ i_0 = 2k \land r = r_0 \land x = x_0 \cdot x_0 \land i = k) \bullet \\
&\quad (r, x, i) : [i \geq 0, r = old(r) \cdot old(x)^{old(i)}]\rangle \rightarrow \\
&((\exists k \in \mathbb{N}.\ i = k \rightarrow i \geq 0) \rightarrow \\
&\quad ((\exists k \in \mathbb{N}.\ i_0 = 2k \land r_1 = r_0 \land x_1 = x_0 \cdot x_0 \land i_1 = k) \land (r = r_1 \cdot x_1^{i_1}))) \rightarrow \\
&(r = r_0 \cdot (x_0 \cdot x_0)^{i_0/2}) = \\
&(r = r_0 \cdot x_0^{i_0}) \qquad \square
\end{aligned}$$

We can again use transitivity of equality to deduce the invariant: The value of $r$ at the end of the iteration will be $old(r) \cdot old(x)^{old(i)}$, $old()$ here again referring to before any iteration. The invariant is therefore $i \geq 0 \land r \cdot x^i = old(r) \cdot old(x)^{old(i)}$. Again, it doesn't state what we are doing, but rather what we are *not* doing.

# 8 Related Work

Hehner [Heh05] [HG99] comes closest in spirit to our stated rule, although he does not account for a rest program $p_2$ serving as the base case for the iteration. His specifications are single predicates, with $pre$, $post$ relationship encoded by implication. He doesn't give them weakest precondition semantics, so our adaption of his refinement semantics for loops to weakest precondition using Morgan's specification statement[Mor88] is new.

Very similar to Hehner's work with regard to the specifications is [Lou+11], where "invariant relations" are used to specify loops as relations between pre- and post-execution state. They use a *relational semantics* of programs for this that is very different from the predicate transformer semantics that we define.

Earlier work are the rules for procedure calls due to Hoare[Hoa71]. Due to the procedures' call-by-name parameters, the rules could very easily be applied to loops that modify local variables, and we find the only aspect in which they are lacking to be that he doesn't allow for parameters to be passed both by name and by value, to circumvent the problem of the original value of a variable no longer being referenceable in the postcondition. We allow this using the $old()$ specifier.

Tuerk [Tue10] states a version of the rule in Hoare logic by parameterizing pre- and postconditions. This is essentially to allow referring to old values, but this fact is rather obfuscated by the way the rule is formulated. The observation that this type of loop specification makes verification of iteration over linked datastructures easier to verify, because one does not have to talk about partial datastructures, is also due to him, and he does account for a rest program $p_2$.

Tuerk's rule is implemented in the tool Verifast [Jac+11] but without the provision for $p_2$, which is vital if that code is the base case of an otherwise *incomplete computation*. Regarding Tuerk's work, our biggest contribution is probably to make the rule and its implications more easily understood and, by defining a weakest precondition semantics for it, make it easier to implement correctly, since we reason only about the aspects relevant for actual use cases.

# 9 Conclusion

We have defined a practical rule for inductive verification of loops specified with contracts. We hope the comparison of inductive verification to that with invariants which we have offered in the various example algorithms we examined convince the reader at least that the invariant method is lacking in some aspects, if not that it should be replaced entirely by the (as we have argued) more natural specification of loops with contracts.

Our utilization of past work that gives well-defined semantics to specifications as programs, Morgan's specification statement[Mor88], and a semantics for loops that allow inductive verification, Hehner's refinement semantics [HG99], allowed us to prove our rule sound in the predicate transformer semantics of the refinement calculus.

Our prototypical implementation shows that the rule is not only practical for theoretical considerations, but also straightforward to implement.

## References

[Dij76] E.W. Dijkstra. *A discipline of programming.* English. Prentice-Hall series in automatic computation. Prentice-Hall, 1976. ISBN: 0-13-215871-X.

[Heh05] Eric C. R. Hehner. "Specified Blocks". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions.* 2005, pp. 384–391. DOI: 10.1007/978-3-540-69149-5_41.

[HG99] Eric C. R. Hehner and Andrew M. Gravel. "Refinement semantics and loop rules". In: *FM'99 — Formal Methods.* Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Springer, 1999, pp. 1497–1510. ISBN: 978-3-540-48118-8.

[Hoa69] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hoa71] C. A. R. Hoare. "Procedures and parameters: An axiomatic approach". In: *Symposium on Semantics of Algorithmic Languages.* Ed. by E. Engeler. Springer, 1971, pp. 102–116. ISBN: 978-3-540-36499-3.

[Jac+11] Bart Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods.* Ed. by Mihaela Bobaru et al. Springer, 2011, pp. 41–55. ISBN: 978-3-642-20398-5.

[Lou+11] Asma Louhichi et al. "Invariant relations: an automated tool to analyze loops". In: *Proceedings of the Fifth international conference on Verification and Evaluation of Computer and Communication Systems.* BCS Learning & Development Ltd. 2011, pp. 84–95.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture.* Ed. by John Hughes. Springer, 1991, pp. 124–144. ISBN: 978-3-540-47599-6.

[Mor88] Carroll Morgan. "The Specification Statement". In: *ACM Trans. Program. Lang. Syst.* 10.3 (July 1988), pp. 403–419. ISSN: 0164-0925. DOI: 10.1145/44501.44503.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science.* LICS '02. IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. URL: http://dl.acm.org/citation.cfm?id=645683.664578.

[Tue10] Thomas Tuerk. "Local Reasoning about While-Loops". In: *VSTTE 2010. Workshop Proceedings.* Ed. by Rajeev Joshi et al. ETH Zürich, 2010, pp. 29–39.