

# Pop-Refinement

Alessandro Coglio

Kestrel Institute  
<http://www.kestrel.edu/~coglio>

December 14, 2021

## Abstract

Pop-refinement is an approach to stepwise refinement, carried out inside an interactive theorem prover by constructing a monotonically decreasing sequence of predicates over deeply embedded target programs. The sequence starts with a predicate that characterizes the possible implementations, and ends with a predicate that characterizes a unique program in explicit syntactic form.

Compared to existing refinement approaches, pop-refinement enables more requirements (e.g. program-level and non-functional) to be captured in the initial specification and preserved through refinement. Security requirements expressed as hyperproperties (i.e. predicates over sets of traces) are always preserved by pop-refinement, unlike the popular notion of refinement as trace set inclusion.

After introducing the concept of pop-refinement, two simple examples in Isabelle/HOL are presented, featuring program-level requirements, non-functional requirements, and hyperproperties. General remarks about pop-refinement follow. Finally, related and future work are discussed.

# Contents

<b>1</b>	<b>Definition</b>	<b>3</b>
<b>2</b>	<b>First Example</b>	<b>4</b>
2.1	Target Programming Language . . . . .	4
2.1.1	Syntax . . . . .	4
2.1.2	Static Semantics . . . . .	5
2.1.3	Dynamic Semantics . . . . .	5
2.1.4	Performance . . . . .	7
2.2	Requirement Specification . . . . .	7
2.3	Stepwise Refinement . . . . .	8
2.3.1	Step 1 . . . . .	8
2.3.2	Step 2 . . . . .	9
2.3.3	Step 3 . . . . .	10
2.3.4	Step 4 . . . . .	10
2.3.5	Step 5 . . . . .	11
2.3.6	Step 6 . . . . .	12
2.3.7	Step 7 . . . . .	13
<b>3</b>	<b>Second Example</b>	<b>14</b>
3.1	Hyperproperties . . . . .	14
3.2	Target Programming Language . . . . .	15
3.2.1	Syntax . . . . .	16
3.2.2	Static Semantics . . . . .	16
3.2.3	Dynamic Semantics . . . . .	17
3.3	Requirement Specification . . . . .	20
3.3.1	Input/Output Variables . . . . .	20
3.3.2	Low Processing . . . . .	20
3.3.3	High Processing . . . . .	21
3.3.4	All Requirements . . . . .	21
3.3.5	Generalized Non-Interference . . . . .	21
3.4	Stepwise Refinement . . . . .	22
3.4.1	Step 1 . . . . .	23
3.4.2	Step 2 . . . . .	24
3.4.3	Step 3 . . . . .	26

3.4.4	Step 4	27
3.4.5	Step 5	28
3.4.6	Step 6	28
<b>4</b>	<b>General Remarks</b>	<b>30</b>
4.1	Program-Level Requirements	30
4.2	Non-Functional Requirements	30
4.3	Links with High-Level Requirements	31
4.4	Non-Determinism and Under-Specification	31
4.5	Specialized Formalisms	32
4.6	Strict and Non-Strict Refinement Steps	32
4.7	Final Predicate	32
4.8	Proof Coverage	33
4.9	Generality and Flexibility	34
<b>5</b>	<b>Related Work</b>	<b>35</b>
<b>6</b>	<b>Future Work</b>	<b>37</b>
6.1	Populating the Framework	37
6.2	Automated Transformations	37
6.3	Other Kinds of Design Objects	37

# Chapter 1

## Definition

In stepwise refinement [4, 18], a program is derived from a specification via a sequence of intermediate specifications.

Pop-refinement (where ‘pop’ stands for ‘predicates over programs’) is an approach to stepwise refinement, carried out inside an interactive theorem prover (e.g. Isabelle/HOL, HOL4, Coq, PVS, ACL2) as follows:

1. Formalize the syntax and semantics of (the needed subset of) the target programming language (and libraries), as a deep embedding.
2. Specify the requirements by defining a predicate over programs that characterizes the possible implementations.
3. Refine the specification stepwise by defining monotonically decreasing predicates over programs (decreasing with respect to inclusion, i.e. logical implication), according to decisions that narrow down the possible implementations.
4. Conclude the derivation with a predicate that characterizes a unique program in explicit syntactic form, from which the program text is readily obtained.

## Chapter 2

# First Example

Pop-refinement is illustrated via a simple derivation, in Isabelle/HOL, of a program that includes non-functional aspects.

### 2.1 Target Programming Language

In the target language used in this example, a program consists of a list of distinct variables (the parameters of the program) and an arithmetic expression (the body of the program). The body is built out of parameters, non-negative integer constants, addition operations, and doubling (i.e. multiplication by 2) operations. The program is executed by supplying non-negative integers to the parameters and evaluating the body to obtain a non-negative integer result.

For instance, executing the program

```
prog (a,b) {3 + 2 * (a + b)}
```

with 5 and 7 supplied to `a` and `b` yields 27. The syntax and semantics of this language are formalized as follows.

#### 2.1.1 Syntax

Variables are identified by names.

**type-synonym** *name* = *string*

Expressions are built out of constants, variables, doubling operations, and addition operations.

**datatype** *expr* = *Const nat* | *Var name* | *Double expr* | *Add expr expr*

A program consists of a list of parameter variables and a body expression.

**record**  $prog =$   
 $para :: name\ list$   
 $body :: expr$

### 2.1.2 Static Semantics

A context is a set of variables.

**type-synonym**  $ctxt = name\ set$

Given a context, an expression is well-formed iff all its variables are in the context.

**fun**  $wfe :: ctxt \Rightarrow expr \Rightarrow bool$   
**where**  
 $wfe\ \Gamma\ (Const\ c) \longleftrightarrow True\ |$   
 $wfe\ \Gamma\ (Var\ v) \longleftrightarrow v \in \Gamma\ |$   
 $wfe\ \Gamma\ (Double\ e) \longleftrightarrow wfe\ \Gamma\ e\ |$   
 $wfe\ \Gamma\ (Add\ e_1\ e_2) \longleftrightarrow wfe\ \Gamma\ e_1 \wedge wfe\ \Gamma\ e_2$

The context of a program consists of the parameters.

**definition**  $ctxt :: prog \Rightarrow ctxt$   
**where**  $ctxt\ p \equiv set\ (para\ p)$

A program is well-formed iff the parameters are distinct and the body is well-formed in the context of the program.

**definition**  $wfp :: prog \Rightarrow bool$   
**where**  $wfp\ p \equiv distinct\ (para\ p) \wedge wfe\ (ctxt\ p)\ (body\ p)$

### 2.1.3 Dynamic Semantics

An environment associates values (non-negative integers) to variables.

**type-synonym**  $env = name \rightarrow nat$

An environment matches a context iff environment and context have the same variables.

**definition**  $match :: env \Rightarrow ctxt \Rightarrow bool$   
**where**  $match\ \mathcal{E}\ \Gamma \equiv dom\ \mathcal{E} = \Gamma$

Evaluating an expression in an environment yields a value, or an error (*None*) if the expression contains a variable not in the environment.

**definition**  $mul-opt :: nat\ option \Rightarrow nat\ option \Rightarrow nat\ option$  (**infixl**  $\otimes$  70)  
— Lifting of multiplication to *nat option*.

**where**  $U_1 \otimes U_2 \equiv$   
*case*  $(U_1, U_2)$  *of*  $(Some\ u_1, Some\ u_2) \Rightarrow Some\ (u_1 * u_2) \mid - \Rightarrow None$

**definition**  $add-opt :: nat\ option \Rightarrow nat\ option \Rightarrow nat\ option$  (**infixl**  $\oplus$  65)  
 — Lifting of addition to *nat option*.

**where**  $U_1 \oplus U_2 \equiv$   
*case*  $(U_1, U_2)$  *of*  $(Some\ u_1, Some\ u_2) \Rightarrow Some\ (u_1 + u_2) \mid - \Rightarrow None$

**fun**  $eval :: env \Rightarrow expr \Rightarrow nat\ option$

**where**

$eval\ \mathcal{E}\ (Const\ c) = Some\ c \mid$   
 $eval\ \mathcal{E}\ (Var\ v) = \mathcal{E}\ v \mid$   
 $eval\ \mathcal{E}\ (Double\ e) = Some\ 2 \otimes eval\ \mathcal{E}\ e \mid$   
 $eval\ \mathcal{E}\ (Add\ e_1\ e_2) = eval\ \mathcal{E}\ e_1 \oplus eval\ \mathcal{E}\ e_2$

Evaluating a well-formed expression never yields an error, if the environment matches the context.

**lemma**  $eval-wfe$ :

$wfe\ \Gamma\ e \Longrightarrow match\ \mathcal{E}\ \Gamma \Longrightarrow eval\ \mathcal{E}\ e \neq None$   
 $\langle proof \rangle$

The environments of a program are the ones that match the context of the program.

**definition**  $envs :: prog \Rightarrow env\ set$

**where**  $envs\ p \equiv \{\mathcal{E}. match\ \mathcal{E}\ (ctxt\ p)\}$

Evaluating the body of a well-formed program in an environment of the program never yields an error.

**lemma**  $eval-wfp$ :

$wfp\ p \Longrightarrow \mathcal{E} \in envs\ p \Longrightarrow eval\ \mathcal{E}\ (body\ p) \neq None$   
 $\langle proof \rangle$

Executing a program with values supplied to the parameters yields a non-negative integer result, or an error (*None*) if the parameters are not distinct, the number of supplied values differs from the number of parameters, or the evaluation of the body yields an error.

**definition**  $supply :: prog \Rightarrow nat\ list \Rightarrow env\ option$

**where**  $supply\ p\ us \equiv$

*let*  $vs = para\ p$  *in*  
*if*  $distinct\ vs \wedge length\ us = length\ vs$   
*then*  $Some\ (map-of\ (zip\ vs\ us))$   
*else*  $None$

**definition**  $exec :: prog \Rightarrow nat\ list \Rightarrow nat\ option$

**where**  $exec\ p\ us \equiv$



*case supply p us of Some  $\mathcal{E} \Rightarrow eval \mathcal{E} (body p) \mid None \Rightarrow None$*

Executing a well-formed program with the same number of values as the number of parameters never yields an error.

**lemma** *supply-wfp*:

*wfp p  $\implies$   
length us = length (para p)  $\implies$   
 $\exists \mathcal{E} \in envs p. supply p us = Some \mathcal{E}$*   
*<proof>*

**lemma** *exec-wfp*:

*wfp p  $\implies$  length us = length (para p)  $\implies exec p us \neq None$*   
*<proof>*

### 2.1.4 Performance

As a non-functional semantic aspect, the cost (e.g. time and power) to execute a program is modeled as the number of doubling and addition operations.

**fun** *coste* :: *expr*  $\Rightarrow$  *nat*

**where**

*coste (Const c) = 0* |  
*coste (Var v) = 0* |  
*coste (Double e) = 1 + coste e* |  
*coste (Add e<sub>1</sub> e<sub>2</sub>) = 1 + coste e<sub>1</sub> + coste e<sub>2</sub>*

**definition** *costp* :: *prog*  $\Rightarrow$  *nat*

**where** *costp p*  $\equiv$  *coste (body p)*

## 2.2 Requirement Specification

The target program must:

1. Be well-formed.
2. Have exactly the two parameters "*x*" and "*y*", in this order.
3. Produce the result *f x y* when *x* and *y* are supplied to "*x*" and "*y*", where *f* is defined below.
4. Not exceed cost 3.

**definition** *f* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*

**where** *f x y*  $\equiv$  *3 \* x + 2 \* y*

**definition**  $spec_0 :: prog \Rightarrow bool$

**where**  $spec_0 p \equiv$

$$\begin{aligned} & wfp\ p \wedge \\ & para\ p = [\"x\", \"y\"] \wedge \\ & (\forall x\ y. exec\ p\ [x, y] = Some\ (f\ x\ y)) \wedge \\ & costp\ p \leq 3 \end{aligned}$$

$f$  is used by  $spec_0$  to express a functional requirement on the execution of the program.  $spec_0$  includes the non-functional requirement  $costp\ p \leq 3$  and the syntactic interface requirement  $para\ p = [\"x\", \"y\"]$ , which are not expressed by  $f$  alone and are expressible only in terms of programs.  $f$  can be computed by a program with cost higher than 3 and with more or different parameters; it can also be computed by programs in different target languages.

## 2.3 Stepwise Refinement

It is not difficult to write a program that satisfies  $spec_0$  and to prove that it does. But with more complex target languages and requirement specifications, writing a program and proving that it satisfies the requirements is notoriously difficult. Stepwise refinement decomposes the proof into manageable pieces, constructing the implementation along the way. The following sequence of refinement steps may be overkill for obtaining an implementation of  $spec_0$ , but illustrates concepts that should apply to more complex cases.

### 2.3.1 Step 1

The second conjunct in  $spec_0$  determines the parameters, leaving only the body to be determined. That conjunct also reduces the well-formedness of the program to the well-formedness of the body, and the execution of the program to the evaluation of the body.

**abbreviation**  $\Gamma_{xy} :: txt$

**where**  $\Gamma_{xy} \equiv \{\"x\", \"y\"\}$

**abbreviation**  $\mathcal{E}_{xy} :: nat \Rightarrow nat \Rightarrow env$

**where**  $\mathcal{E}_{xy}\ x\ y \equiv [\"x\" \mapsto x, \"y\" \mapsto y]$

**lemma** *reduce-prog-to-body*:

$$\begin{aligned} & para\ p = [\"x\", \"y\"] \implies \\ & wfp\ p = wfe\ \Gamma_{xy}\ (body\ p) \wedge \\ & exec\ p\ [x, y] = eval\ (\mathcal{E}_{xy}\ x\ y)\ (body\ p) \end{aligned}$$

*<proof>*

Using lemma *reduce-prog-to-body*, and using the definition of  $costp$  to reduce the cost of the program to the cost of the body,  $spec_0$  is refined as follows.

**definition**  $spec_1 :: prog \Rightarrow bool$

**where**  $spec_1 p \equiv$

$$\begin{aligned} & wfe \Gamma_{xy} (body\ p) \wedge \\ & para\ p = [\'x\ \prime, \'y\ \prime] \wedge \\ & (\forall x\ y. eval\ (\mathcal{E}_{xy}\ x\ y)\ (body\ p) = Some\ (f\ x\ y)) \wedge \\ & coste\ (body\ p) \leq 3 \end{aligned}$$

**lemma** *step-1-correct*:

$$spec_1\ p \Longrightarrow spec_0\ p$$

*<proof>*

$spec_1$  and  $spec_0$  are actually equivalent, but the definition of  $spec_1$  is “closer” to the implementation than the definition of  $spec_0$ : the latter states constraints on the whole program, while the former states simpler constraints on the body, given that the parameters are already determined. The proof of *step-1-correct* can also be used to prove the equivalence of  $spec_1$  and  $spec_0$ , but in general proving inclusion is easier than proving equivalence. Some of the following refinement steps yield non-equivalent predicates.

### 2.3.2 Step 2

The third conjunct in  $spec_1$  says that the body computes  $f\ x\ y$ , which depends on both  $x$  and  $y$ , and which yields an odd result for some values of  $x$  and  $y$ . Thus the body cannot be a constant, a variable, or a double, leaving a sum as the only option. Adding  $\exists e_1\ e_2. body\ p = Add\ e_1\ e_2$  as a conjunct to  $spec_1$  and re-arranging the other conjuncts, moving some of them under the existential quantification so that they can be simplified in the next refinement step,  $spec_1$  is refined as follows.

**definition**  $spec_2 :: prog \Rightarrow bool$

**where**  $spec_2 p \equiv$

$$\begin{aligned} & para\ p = [\'x\ \prime, \'y\ \prime] \wedge \\ & (\exists e_1\ e_2. \\ & \quad body\ p = Add\ e_1\ e_2 \wedge \\ & \quad wfe \Gamma_{xy} (body\ p) \wedge \\ & \quad (\forall x\ y. eval\ (\mathcal{E}_{xy}\ x\ y)\ (body\ p) = Some\ (f\ x\ y)) \wedge \\ & \quad coste\ (body\ p) \leq 3) \end{aligned}$$

**lemma** *step-2-correct*:

$$spec_2\ p \Longrightarrow spec_1\ p$$

*<proof>*

This refinement step is guided by an analysis of the constraints in  $spec_1$ .

### 2.3.3 Step 3

The fact that the body is a sum reduces the well-formedness, evaluation, and cost of the body to the well-formedness, evaluation, and cost of the addends.

**lemma** *reduce-body-to-addends*:

$$\begin{aligned}
& \text{body } p = \text{Add } e_1 \ e_2 \implies \\
& \text{wfe } \Gamma_{xy} (\text{body } p) = (\text{wfe } \Gamma_{xy} \ e_1 \wedge \text{wfe } \Gamma_{xy} \ e_2) \wedge \\
& \text{eval } (\mathcal{E}_{xy} \ x \ y) (\text{body } p) = \text{eval } (\mathcal{E}_{xy} \ x \ y) \ e_1 \oplus \text{eval } (\mathcal{E}_{xy} \ x \ y) \ e_2 \wedge \\
& \text{coste } (\text{body } p) = 1 + \text{coste } e_1 + \text{coste } e_2 \\
& \langle \text{proof} \rangle
\end{aligned}$$

Using *reduce-body-to-addends* and arithmetic simplification,  $\text{spec}_2$  is refined as follows.

**definition**  $\text{spec}_3 :: \text{prog} \Rightarrow \text{bool}$

**where**  $\text{spec}_3 \ p \equiv$

$$\begin{aligned}
& \text{para } p = [\"x\", \"y\"] \wedge \\
& (\exists e_1 \ e_2. \\
& \quad \text{body } p = \text{Add } e_1 \ e_2 \wedge \\
& \quad \text{wfe } \Gamma_{xy} \ e_1 \wedge \\
& \quad \text{wfe } \Gamma_{xy} \ e_2 \wedge \\
& \quad (\forall x \ y. \text{eval } (\mathcal{E}_{xy} \ x \ y) \ e_1 \oplus \text{eval } (\mathcal{E}_{xy} \ x \ y) \ e_2 = \text{Some } (f \ x \ y)) \wedge \\
& \quad \text{coste } e_1 + \text{coste } e_2 \leq 2)
\end{aligned}$$

**lemma** *step-3-correct*:

$$\text{spec}_3 \ p \implies \text{spec}_2 \ p$$

$\langle \text{proof} \rangle$

This refinement step defines the top-level structure of the body, reducing the constraints on the body to simpler constraints on its components.

### 2.3.4 Step 4

The second-to-last conjunct in  $\text{spec}_3$  suggests to split  $f \ x \ y$  into two addends to be computed by  $e_1$  and  $e_2$ .

The addends  $3 * x$  and  $2 * y$  suggested by the definition of  $f$  would lead to a blind alley, where the cost constraints could not be satisfied—the resulting  $\text{spec}_4$  would be always false. The refinement step would be “correct” (by strict inclusion) but the refinement sequence could never reach an implementation. It would be necessary to backtrack to  $\text{spec}_3$  and split  $f \ x \ y$  differently.

To avoid the blind alley, the definition of  $f$  is rephrased as follows.

**lemma** *f-rephrased*:

$$f \ x \ y = x + (2 * x + 2 * y)$$

$\langle proof \rangle$

This rephrased definition of  $f$  does not use the multiplication by 3 of the original definition, which is not (directly) supported by the target language; it only uses operations supported by the language.

Using  $f$ -rephrased,  $spec_3$  is refined as follows.

**definition**  $spec_4 :: prog \Rightarrow bool$

**where**  $spec_4 p \equiv$

$para p = [\"x\", \"y\"] \wedge$

$(\exists e_1 e_2.$

$body p = Add e_1 e_2 \wedge$

$wfe \Gamma_{xy} e_1 \wedge$

$wfe \Gamma_{xy} e_2 \wedge$

$(\forall x y. eval (\mathcal{E}_{xy} x y) e_1 = Some x) \wedge$

$(\forall x y. eval (\mathcal{E}_{xy} x y) e_2 = Some (2 * x + 2 * y)) \wedge$

$coste e_1 + coste e_2 \leq 2)$

**lemma** *step-4-correct*:

$spec_4 p \implies spec_3 p$

$\langle proof \rangle$

This refinement step reduces the functional constraint on the body to simpler functional constraints on the addends. The functional constraint can be decomposed in different ways, some of which are incompatible with the non-functional cost constraint: blind alleys are avoided by taking the non-functional constraint into account.

### 2.3.5 Step 5

The term  $x$  in the third-to-last conjunct in  $spec_4$  is a shallow embedding of the program expression  $x$ , whose deep embedding is the term  $Var \"x\"$ . Using the latter as  $e_1$ , the third-to-last conjunct in  $spec_4$  is satisfied; the expression is well-formed and has cost 0.

**lemma** *first-addend*:

$e_1 = Var \"x\" \implies$

$eval (\mathcal{E}_{xy} x y) e_1 = Some x \wedge$

$wfe \Gamma_{xy} e_1 \wedge$

$coste e_1 = 0$

$\langle proof \rangle$

Adding  $e_1 = Var \"x\"$  as a conjunct to  $spec_4$  and simplifying,  $spec_4$  is refined as follows.

**definition**  $spec_5 :: prog \Rightarrow bool$

**where**  $spec_5 p \equiv$

$$\begin{aligned}
\text{para } p &= [{}''x'', {}''y''] \wedge \\
(\exists e_2. & \\
\text{body } p &= \text{Add } (\text{Var } {}''x'') e_2 \wedge \\
\text{wfe } \Gamma_{xy} e_2 &\wedge \\
(\forall x y. \text{eval } (\mathcal{E}_{xy} x y) e_2 &= \text{Some } (2 * x + 2 * y)) \wedge \\
\text{coste } e_2 &\leq 2)
\end{aligned}$$

**lemma** *step-5-correct*:

$$\text{spec}_5 p \implies \text{spec}_4 p$$

*<proof>*

This refinement step determines the first addend of the body, leaving only the second addend to be determined.

### 2.3.6 Step 6

The term  $2 * x + 2 * y$  in the second-to-last conjunct of  $\text{spec}_5$  is a shallow embedding of the program expression  $2 * x + 2 * y$ , whose deep embedding is the term  $\text{Add } (\text{Double } (\text{Var } {}''x'')) (\text{Double } (\text{Var } {}''y''))$ . Using the latter as  $e_2$ , the second-to-last conjunct in  $\text{spec}_5$  is satisfied, but the last conjunct is not. The following factorization of the shallowly embedded expression leads to a reduced cost of the corresponding deeply embedded expression.

**lemma** *factorization*:

$$(2::\text{nat}) * x + 2 * y = 2 * (x + y)$$

*<proof>*

The deeply embedded expression  $\text{Double } (\text{Add } (\text{Var } {}''x'') (\text{Var } {}''y''))$ , which corresponds to the shallowly embedded expression  $2 * (x + y)$ , satisfies the second-to-last conjunct of  $\text{spec}_5$ , is well-formed, and has cost 2.

**lemma** *second-addend*:

$$\begin{aligned}
e_2 = \text{Double } (\text{Add } (\text{Var } {}''x'') (\text{Var } {}''y'')) &\implies \\
\text{eval } (\mathcal{E}_{xy} x y) e_2 = \text{Some } (2 * x + 2 * y) &\wedge \\
\text{wfe } \Gamma_{xy} e_2 &\wedge \\
\text{coste } e_2 = 2 &
\end{aligned}$$

*<proof>*

Adding  $e_2 = \text{Double } (\text{Add } (\text{Var } {}''x'') (\text{Var } {}''y''))$  as a conjunct to  $\text{spec}_5$  and simplifying,  $\text{spec}_5$  is refined as follows.

**definition**  $\text{spec}_6 :: \text{prog} \Rightarrow \text{bool}$

**where**  $\text{spec}_6 p \equiv$

$$\begin{aligned}
\text{para } p &= [{}''x'', {}''y''] \wedge \\
\text{body } p &= \text{Add } (\text{Var } {}''x'') (\text{Double } (\text{Add } (\text{Var } {}''x'') (\text{Var } {}''y'')))
\end{aligned}$$

**lemma** *step-6-correct*:

$spec_6 p \implies spec_5 p$   
 $\langle proof \rangle$

This refinement step determines the second addend of the body, leaving nothing else to be determined.

This and the previous refinement step turn semantic constraints on the program components  $e_1$  and  $e_2$  into syntactic definitions of such components.

### 2.3.7 Step 7

$spec_6$ , which defines the parameters and body, is refined to characterize a unique program in explicit syntactic form.

**abbreviation**  $p_0 :: prog$

**where**  $p_0 \equiv$

$(para = [\'x\', \'y\'],$   
 $body = Add (Var \'x\') (Double (Add (Var \'x\') (Var \'y\'))))$

**definition**  $spec_7 :: prog \Rightarrow bool$

**where**  $spec_7 p \equiv p = p_0$

**lemma** *step-7-correct*:

$spec_7 p \implies spec_6 p$   
 $\langle proof \rangle$

The program satisfies  $spec_0$  by construction. The program witnesses the consistency of the requirements, i.e. the fact that  $spec_0$  is not always false.

**lemma**  *$p_0$ -sat- $spec_0$* :

$spec_0 p_0$   
 $\langle proof \rangle$

From  $p_0$ , the program text

`prog (x,y) {x + 2 * (x + y)}`

is easily obtained.

# Chapter 3

## Second Example

Pop-refinement is illustrated via a simple derivation, in Isabelle/HOL, of a non-deterministic program that satisfies a hyperproperty.

### 3.1 Hyperproperties

Hyperproperties are predicates over sets of traces [3]. Hyperproperties capture security policies like non-interference [5], which applies to deterministic systems, and generalized non-interference (GNI) [11], which generalizes non-interference to non-deterministic systems.

The formulation of GNI in [3], which is derived from [12], is based on:

- A notion of traces as infinite streams of abstract states.
- Functions that map each state to low and high inputs and outputs, where ‘low’ and ‘high’ have the usual security meaning (e.g. ‘low’ means ‘unclassified’ and ‘high’ means ‘classified’). These functions are homomorphically extended to map each trace to infinite streams of low and high inputs and outputs.

The following formulation is slightly more general, because the functions that return low and high inputs and outputs operate directly on abstract traces.

GNI says that for any two traces  $\tau_1$  and  $\tau_2$ , there is always a trace  $\tau_3$  with the same high inputs as  $\tau_1$  and the same low inputs and low outputs as  $\tau_2$ . Intuitively, this means that a low observer (i.e. one that only observes low inputs and low outputs of traces) cannot gain any information about high inputs (i.e. high inputs cannot interfere with low outputs) because observing a trace  $\tau_2$  is indistinguishable from observing some other trace  $\tau_3$  that has the same high inputs as an arbitrary trace  $\tau_1$ .



**locale** *generalized-non-interference* =  
**fixes** *low-in* :: 'τ ⇒ 'i — low inputs  
**fixes** *low-out* :: 'τ ⇒ 'o — low outputs  
**fixes** *high-in* :: 'τ ⇒ 'i — high inputs  
**fixes** *high-out* :: 'τ ⇒ 'o — high outputs

**definition** (in *generalized-non-interference*) *GNI* :: 'τ set ⇒ bool

**where** *GNI*  $\mathcal{T} \equiv$

$\forall \tau_1 \in \mathcal{T}. \forall \tau_2 \in \mathcal{T}. \exists \tau_3 \in \mathcal{T}.$

$high-in \tau_3 = high-in \tau_1 \wedge low-in \tau_3 = low-in \tau_2 \wedge low-out \tau_3 = low-out \tau_2$

## 3.2 Target Programming Language

In the target language used in this example,<sup>1</sup> a program consists of a list of distinct state variables and a body of statements. The statements modify the variables by deterministically assigning results of expressions and by non-deterministically assigning random values. Expressions are built out of non-negative integer constants, state variables, and addition operations. Statements are combined via conditionals, whose tests compare expressions for equality, and via sequencing. Each variable stores a non-negative integer. Executing the body in a state yields a new state. Because of non-determinism, different new states are possible, i.e. executing the body in the same state may yield different new states at different times.

For instance, executing the body of the program

```

prog {
  vars {
    x
    y
  }
  body {
    if (x == y + 1) {
      x = 0;
    } else {
      x = y + 3;
    }
    randomize y;
    y = y + 2;
  }
}

```

in the state where *x* contains 4 and *y* contains 7, yields a new state where *x* always contains 10 and *y* may contain any number in  $\{2, 3, \dots\}$ .

<sup>1</sup>Even though this language has many similarities with the language in Section 2.1, the two languages are defined separately to keep Chapter 2 simpler.

### 3.2.1 Syntax

Variables are identified by names.

**type-synonym**  $name = string$

Expressions are built out of constants, variables, and addition operations.

**datatype**  $expr = Const\ nat \mid Var\ name \mid Add\ expr\ expr$

Statements are built out of deterministic assignments, non-deterministic assignments, conditionals, and sequencing.

**datatype**  $stmt =$   
 $Assign\ name\ expr \mid$   
 $Random\ name \mid$   
 $IfEq\ expr\ expr\ stmt\ stmt \mid$   
 $Seq\ stmt\ stmt$

A program consists of a list of state variables and a body statement.

**record**  $prog =$   
 $vars :: name\ list$   
 $body :: stmt$

### 3.2.2 Static Semantics

A context is a set of variables.

**type-synonym**  $ctxt = name\ set$

Given a context, an expression is well-formed iff all its variables are in the context.

**fun**  $wfe :: ctxt \Rightarrow expr \Rightarrow bool$   
**where**  
 $wfe\ \Gamma\ (Const\ c) \longleftrightarrow True \mid$   
 $wfe\ \Gamma\ (Var\ v) \longleftrightarrow v \in \Gamma \mid$   
 $wfe\ \Gamma\ (Add\ e_1\ e_2) \longleftrightarrow wfe\ \Gamma\ e_1 \wedge wfe\ \Gamma\ e_2$

Given a context, a statement is well-formed iff its deterministic assignments assign well-formed expressions to variables in the context, its non-deterministic assignments operate on variables in the context, and its conditional tests compare well-formed expressions.

**fun**  $wfs :: ctxt \Rightarrow stmt \Rightarrow bool$   
**where**  
 $wfs\ \Gamma\ (Assign\ v\ e) \longleftrightarrow v \in \Gamma \wedge wfe\ \Gamma\ e \mid$   
 $wfs\ \Gamma\ (Random\ v) \longleftrightarrow v \in \Gamma \mid$   
 $wfs\ \Gamma\ (IfEq\ e_1\ e_2\ s_1\ s_2) \longleftrightarrow wfe\ \Gamma\ e_1 \wedge wfe\ \Gamma\ e_2 \wedge wfs\ \Gamma\ s_1 \wedge wfs\ \Gamma\ s_2 \mid$

$$wfs \Gamma (Seq s_1 s_2) \longleftrightarrow wfs \Gamma s_1 \wedge wfs \Gamma s_2$$

The context of a program consists of the state variables.

**definition**  $ctxt :: prog \Rightarrow ctxt$   
**where**  $ctxt p \equiv set (vars p)$

A program is well-formed iff the variables are distinct and the body is well-formed in the context of the program.

**definition**  $wfp :: prog \Rightarrow bool$   
**where**  $wfp p \equiv distinct (vars p) \wedge wfs (ctxt p) (body p)$

### 3.2.3 Dynamic Semantics

A state associates values (non-negative integers) to variables.

**type-synonym**  $state = name \rightarrow nat$

A state matches a context iff state and context have the same variables.

**definition**  $match :: state \Rightarrow ctxt \Rightarrow bool$   
**where**  $match \sigma \Gamma \equiv dom \sigma = \Gamma$

Evaluating an expression in a state yields a value, or an error (*None*) if the expression contains a variable not in the state.

**definition**  $add-opt :: nat option \Rightarrow nat option \Rightarrow nat option$  (**infixl**  $\oplus$  65)  
 — Lifting of addition to *nat option*.

**where**  $U_1 \oplus U_2 \equiv$   
 $case (U_1, U_2) of (Some u_1, Some u_2) \Rightarrow Some (u_1 + u_2) \mid - \Rightarrow None$

**fun**  $eval :: state \Rightarrow expr \Rightarrow nat option$   
**where**

$eval \sigma (Const c) = Some c \mid$   
 $eval \sigma (Var v) = \sigma v \mid$   
 $eval \sigma (Add e_1 e_2) = eval \sigma e_1 \oplus eval \sigma e_2$

Evaluating a well-formed expression never yields an error, if the state matches the context.

**lemma** *eval-wfe*:

$wfe \Gamma e \Longrightarrow match \sigma \Gamma \Longrightarrow eval \sigma e \neq None$   
 $\langle proof \rangle$

Executing a statement in a state yields a new state, or an error (*None*) if the evaluation of an expression yields an error or if an assignment operates on a variable not in the state. Non-determinism is modeled via a relation between old states and results, where a result is either a new state or an error.

**inductive**  $exec :: stmt \Rightarrow state \Rightarrow state option \Rightarrow bool$

$(- \triangleright - \rightsquigarrow - [50, 50, 50] 50)$

**where**

*ExecAssignNoVar:*

$v \notin \text{dom } \sigma \implies \text{Assign } v \ e \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecAssignEvalError:*

$\text{eval } \sigma \ e = \text{None} \implies \text{Assign } v \ e \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecAssignOK:*

$v \in \text{dom } \sigma \implies$

$\text{eval } \sigma \ e = \text{Some } u \implies$

$\text{Assign } v \ e \triangleright \sigma \rightsquigarrow \text{Some } (\sigma(v \mapsto u)) \mid$

*ExecRandomNoVar:*

$v \notin \text{dom } \sigma \implies \text{Random } v \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecRandomOK:*

$v \in \text{dom } \sigma \implies \text{Random } v \triangleright \sigma \rightsquigarrow \text{Some } (\sigma(v \mapsto u)) \mid$

*ExecCondEvalError1:*

$\text{eval } \sigma \ e_1 = \text{None} \implies \text{IfEq } e_1 \ e_2 \ s_1 \ s_2 \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecCondEvalError2:*

$\text{eval } \sigma \ e_2 = \text{None} \implies \text{IfEq } e_1 \ e_2 \ s_1 \ s_2 \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecCondTrue:*

$\text{eval } \sigma \ e_1 = \text{Some } u_1 \implies$

$\text{eval } \sigma \ e_2 = \text{Some } u_2 \implies$

$u_1 = u_2 \implies$

$s_1 \triangleright \sigma \rightsquigarrow \varrho \implies$

$\text{IfEq } e_1 \ e_2 \ s_1 \ s_2 \triangleright \sigma \rightsquigarrow \varrho \mid$

*ExecCondFalse:*

$\text{eval } \sigma \ e_1 = \text{Some } u_1 \implies$

$\text{eval } \sigma \ e_2 = \text{Some } u_2 \implies$

$u_1 \neq u_2 \implies$

$s_2 \triangleright \sigma \rightsquigarrow \varrho \implies$

$\text{IfEq } e_1 \ e_2 \ s_1 \ s_2 \triangleright \sigma \rightsquigarrow \varrho \mid$

*ExecSeqError:*

$s_1 \triangleright \sigma \rightsquigarrow \text{None} \implies \text{Seq } s_1 \ s_2 \triangleright \sigma \rightsquigarrow \text{None} \mid$

*ExecSeqOK:*

$s_1 \triangleright \sigma \rightsquigarrow \text{Some } \sigma' \implies s_2 \triangleright \sigma' \rightsquigarrow \varrho \implies \text{Seq } s_1 \ s_2 \triangleright \sigma \rightsquigarrow \varrho$

The execution of any statement in any state always yields a result.

**lemma** *exec-always:*

$\exists \varrho. s \triangleright \sigma \rightsquigarrow \varrho$

$\langle \text{proof} \rangle$

Executing a well-formed statement in a state that matches the context never yields an error and always yields states that match the context.

**lemma** *exec-wfs-match:*

$\text{wfs } \Gamma \ s \implies \text{match } \sigma \ \Gamma \implies s \triangleright \sigma \rightsquigarrow \text{Some } \sigma' \implies \text{match } \sigma' \ \Gamma$

$\langle \text{proof} \rangle$

**lemma** *exec-wfs-no-error*:

$wfs \Gamma s \implies match \sigma \Gamma \implies \neg (s \triangleright \sigma \rightsquigarrow None)$   
*<proof>*

**lemma** *exec-wfs-always-match*:

$wfs \Gamma s \implies match \sigma \Gamma \implies \exists \sigma'. s \triangleright \sigma \rightsquigarrow Some \sigma' \wedge match \sigma' \Gamma$   
*<proof>*

The states of a program are the ones that match the context of the program.

**definition** *states* :: *prog*  $\Rightarrow$  *state set*

**where** *states* *p*  $\equiv \{\sigma. match \sigma (ctxt p)\}$

Executing the body of a well-formed program in a state of the program always yields some state of the program, and never an error.

**lemma** *exec-wfp-no-error*:

$wfp p \implies \sigma \in states p \implies \neg (body p \triangleright \sigma \rightsquigarrow None)$   
*<proof>*

**lemma** *exec-wfp-in-states*:

$wfp p \implies \sigma \in states p \implies body p \triangleright \sigma \rightsquigarrow Some \sigma' \implies \sigma' \in states p$   
*<proof>*

**lemma** *exec-wfp-always-in-states*:

$wfp p \implies \sigma \in states p \implies \exists \sigma'. body p \triangleright \sigma \rightsquigarrow Some \sigma' \wedge \sigma' \in states p$   
*<proof>*

Program execution can be described in terms of the trace formalism in [3]. Every possible (non-erroneous) execution of a program can be described by a trace of two states—initial and final. In this definition, erroneous executions do not contribute to the traces of a program; only well-formed programs are of interest, which, as proved above, never execute erroneously. Due to non-determinism, there may be traces with the same initial state and different final states.

**record** *trace* =

*initial* :: *state*

*final* :: *state*

**inductive-set** *traces* :: *prog*  $\Rightarrow$  *trace set*

**for** *p::prog*

**where** [*intro!*]:

$\sigma \in states p \implies$

$body p \triangleright \sigma \rightsquigarrow Some \sigma' \implies$

$(initial = \sigma, final = \sigma') \in traces p$

The finite traces of a program could be turned into infinite traces by infinitely stuttering the final state, obtaining the ‘executions’ defined in [3]. However, such

infinite traces carry no additional information compared to the finite traces from which they are derived: for programs in this language, the infinite executions of [3] are modeled as finite traces of type *trace*.

### 3.3 Requirement Specification

The target program must process low and high inputs to yield low and high outputs, according to constraints that involve both non-determinism and under-specification, with no information flowing from high inputs to low outputs.<sup>2</sup>

#### 3.3.1 Input/Output Variables

Even though the language defined in Section 3.2 has no explicit features for input and output, an external agent could write values into some variables, execute the program body, and read values from some variables. Thus, variables may be regarded as holding inputs (in the initial state) and outputs (in the final state).

In the target program, four variables are required:

- A variable *"lowIn"* to hold low inputs.
- A variable *"lowOut"* to hold low outputs.
- A variable *"highIn"* to hold high inputs.
- A variable *"highOut"* to hold high outputs.

Other variables are allowed but not required.

**definition** *io-vars* :: *prog*  $\Rightarrow$  *bool*  
**where** *io-vars* *p*  $\equiv$  *ctxt p*  $\supseteq$  {*"lowIn"*, *"lowOut"*, *"highIn"*, *"highOut"*}

#### 3.3.2 Low Processing

If the low input is not 0, the low output must be 1 plus the low input. That is, for every possible execution of the program where the initial state's low input is not 0, the final state's low output must be 1 plus the low input. If there are multiple possible final states for the same initial state due to non-determinism, all of them must have the same required low output. Thus, processing of non-0 low inputs must be deterministic.

**definition** *low-proc-non0* :: *prog*  $\Rightarrow$  *bool*  
**where** *low-proc-non0* *p*  $\equiv$

<sup>2</sup>As in Section 3.1, 'low' and 'high' have the usual security meaning, e.g. 'low' means 'unclassified' and 'high' means 'classified'.

$$\begin{aligned}
& \forall \sigma \in \text{states } p. \forall \sigma'. \\
& \text{the } (\sigma \text{ "lowIn"}) \neq 0 \wedge \\
& \text{body } p \triangleright \sigma \rightsquigarrow \text{Some } \sigma' \longrightarrow \\
& \text{the } (\sigma' \text{ "lowOut"}) = \text{the } (\sigma \text{ "lowIn"}) + 1
\end{aligned}$$

If the low input is 0, the low output must be a random value. That is, for every possible initial state of the program whose low input is 0, and for every possible value, there must exist an execution of the program whose final state has that value as low output. Executions corresponding to all possible values must be possible. Thus, processing of the 0 low input must be non-deterministic.

**definition** *low-proc-0* :: *prog*  $\Rightarrow$  *bool*

**where** *low-proc-0* *p*  $\equiv$

$$\begin{aligned}
& \forall \sigma \in \text{states } p. \forall u. \\
& \text{the } (\sigma \text{ "lowIn"}) = 0 \longrightarrow \\
& (\exists \sigma'. \text{body } p \triangleright \sigma \rightsquigarrow \text{Some } \sigma' \wedge \text{the } (\sigma' \text{ "lowOut"}) = u)
\end{aligned}$$

### 3.3.3 High Processing

The high output must be at least as large as the sum of the low and high inputs. That is, for every possible execution of the program, the final state's high output must satisfy the constraint. If there are multiple possible final states for the same initial state due to non-determinism, all of them must contain a high output that satisfies the constraint. Since different high outputs may satisfy the constraint given the same inputs, not all the possible final states from a given initial state must have the same high output. Thus, processing of high inputs is under-specified; it can be realized deterministically or non-deterministically.

**definition** *high-proc* :: *prog*  $\Rightarrow$  *bool*

**where** *high-proc* *p*  $\equiv$

$$\begin{aligned}
& \forall \sigma \in \text{states } p. \forall \sigma'. \\
& \text{body } p \triangleright \sigma \rightsquigarrow \text{Some } \sigma' \longrightarrow \\
& \text{the } (\sigma' \text{ "highOut"}) \geq \text{the } (\sigma \text{ "lowIn"}) + \text{the } (\sigma \text{ "highIn"})
\end{aligned}$$

### 3.3.4 All Requirements

Besides satisfying the above requirements on input/output variables, low processing, and high processing, the target program must be well-formed.

**definition** *spec<sub>0</sub>* :: *prog*  $\Rightarrow$  *bool*

**where** *spec<sub>0</sub>* *p*  $\equiv$

$$\text{wfp } p \wedge \text{io-vars } p \wedge \text{low-proc-non0 } p \wedge \text{low-proc-0 } p \wedge \text{high-proc } p$$

### 3.3.5 Generalized Non-Interference

The parameters of the GNI formulation in Section 3.1 are instantiated according to the target program under consideration. In an execution of the program:

- The value of the variable *"lowIn"* in the initial state is the low input.
- The value of the variable *"lowOut"* in the final state is the low output.
- The value of the variable *"highIn"* in the initial state is the high input.
- The value of the variable *"highOut"* in the final state is the high output.

**definition** *low-in* :: *trace*  $\Rightarrow$  *nat*  
**where** *low-in*  $\tau \equiv$  *the* (*initial*  $\tau$  *"lowIn"*)

**definition** *low-out* :: *trace*  $\Rightarrow$  *nat*  
**where** *low-out*  $\tau \equiv$  *the* (*final*  $\tau$  *"lowOut"*)

**definition** *high-in* :: *trace*  $\Rightarrow$  *nat*  
**where** *high-in*  $\tau \equiv$  *the* (*initial*  $\tau$  *"highIn"*)

**definition** *high-out* :: *trace*  $\Rightarrow$  *nat*  
**where** *high-out*  $\tau \equiv$  *the* (*final*  $\tau$  *"highOut"*)

**interpretation**

*Target: generalized-non-interference low-in low-out high-in high-out*  $\langle$ *proof* $\rangle$

**abbreviation** *GNI* :: *trace set*  $\Rightarrow$  *bool*  
**where** *GNI*  $\equiv$  *Target.GNI*

The requirements in *spec<sub>0</sub>* imply that the set of traces of the target program satisfies GNI.

**lemma** *spec<sub>0</sub>-GNI*:  
*spec<sub>0</sub> p*  $\implies$  *GNI* (*traces p*)  
 $\langle$ *proof* $\rangle$

Since GNI is implied by *spec<sub>0</sub>* and since every pop-refinement of *spec<sub>0</sub>* implies *spec<sub>0</sub>*, GNI is preserved through every pop-refinement of *spec<sub>0</sub>*. Pop-refinement differs from the popular notion of refinement as inclusion of sets of traces (e.g. [1]), which does not preserve GNI [3].

### 3.4 Stepwise Refinement

The remark at the beginning of Section 2.3 applies here as well: the following sequence of refinement steps may be overkill for obtaining an implementation of *spec<sub>0</sub>*, but illustrates concepts that should apply to more complex cases.



### 3.4.1 Step 1

The program needs no other variables besides those prescribed by *io-vars*. Thus, *io-vars* is refined to a stronger condition that constrains the program to contain exactly those variables, in a certain order.

**abbreviation**  $vars_0 :: \text{name list}$   
**where**  $vars_0 \equiv ["lowIn", "lowOut", "highIn", "highOut"]$   
 — The order of the variables in the list is arbitrary.

**lemma** *vars<sub>0</sub>-correct*:  
 $vars\ p = vars_0 \implies io\text{-}vars\ p$   
 $\langle \text{proof} \rangle$

The refinement of *io-vars* reduces the well-formedness of the program to the well-formedness of the body.

**abbreviation**  $\Gamma_0 :: \text{ctxt}$   
**where**  $\Gamma_0 \equiv \{"lowIn", "lowOut", "highIn", "highOut"\}$

**lemma** *reduce-wf-prog-to-body*:  
 $vars\ p = vars_0 \implies wfp\ p \longleftrightarrow wfs\ \Gamma_0\ (\text{body}\ p)$   
 $\langle \text{proof} \rangle$

The refinement of *io-vars* induces a simplification of the processing constraints: since the context of the program is now defined to be  $\Gamma_0$ , the  $\sigma \in \text{states}\ p$  conditions are replaced with *match*  $\sigma\ \Gamma_0$  conditions.

**definition** *low-proc-non0<sub>1</sub>* ::  $\text{prog} \Rightarrow \text{bool}$   
**where** *low-proc-non0<sub>1</sub>*  $p \equiv$   
 $\forall \sigma\ \sigma'.$   
 $\text{match}\ \sigma\ \Gamma_0 \wedge$   
 $\text{the}\ (\sigma\ \text{"lowIn"}) \neq 0 \wedge$   
 $\text{body}\ p \triangleright \sigma \rightsquigarrow \text{Some}\ \sigma' \longrightarrow$   
 $\text{the}\ (\sigma'\ \text{"lowOut"}) = \text{the}\ (\sigma\ \text{"lowIn"}) + 1$

**lemma** *low-proc-non0<sub>1</sub>-correct*:  
 $vars\ p = vars_0 \implies \text{low-proc-non0}_1\ p \longleftrightarrow \text{low-proc-non0}\ p$   
 $\langle \text{proof} \rangle$

**definition** *low-proc-0<sub>1</sub>* ::  $\text{prog} \Rightarrow \text{bool}$   
**where** *low-proc-0<sub>1</sub>*  $p \equiv$   
 $\forall \sigma\ u.$   
 $\text{match}\ \sigma\ \Gamma_0 \wedge$   
 $\text{the}\ (\sigma\ \text{"lowIn"}) = 0 \longrightarrow$   
 $(\exists \sigma'. \text{body}\ p \triangleright \sigma \rightsquigarrow \text{Some}\ \sigma' \wedge \text{the}\ (\sigma'\ \text{"lowOut"}) = u)$

**lemma** *low-proc-0<sub>1</sub>-correct*:

$vars\ p = vars_0 \implies low\text{-}proc\text{-}0_1\ p \longleftrightarrow low\text{-}proc\text{-}0\ p$   
 $\langle proof \rangle$

**definition**  $high\text{-}proc_1 :: prog \Rightarrow bool$

**where**  $high\text{-}proc_1\ p \equiv$

$\forall \sigma\ \sigma'.$   
 $match\ \sigma\ \Gamma_0 \wedge$   
 $body\ p \triangleright \sigma \rightsquigarrow Some\ \sigma' \longrightarrow$   
 $the\ (\sigma'\ "highOut'') \geq the\ (\sigma\ "lowIn'') + the\ (\sigma\ "highIn'')$

**lemma**  $high\text{-}proc_1\text{-}correct:$

$vars\ p = vars_0 \implies high\text{-}proc_1\ p \longleftrightarrow high\text{-}proc\ p$   
 $\langle proof \rangle$

The refinement of  $spec_0$  consists of the refinement of  $io\text{-}vars$  and of the simplified constraints.

**definition**  $spec_1 :: prog \Rightarrow bool$

**where**  $spec_1\ p \equiv$

$vars\ p = vars_0 \wedge$   
 $wfs\ \Gamma_0\ (body\ p) \wedge$   
 $low\text{-}proc\text{-}non0_1\ p \wedge$   
 $low\text{-}proc\text{-}0_1\ p \wedge$   
 $high\text{-}proc_1\ p$

**lemma**  $step\text{-}1\text{-}correct:$

$spec_1\ p \implies spec_0\ p$   
 $\langle proof \rangle$

### 3.4.2 Step 2

The body of the target program is split into two sequential statements—one to compute the low output and one to compute the high output.

**definition**  $body\text{-}split :: prog \Rightarrow stmt \Rightarrow stmt \Rightarrow bool$

**where**  $body\text{-}split\ p\ s_L\ s_H \equiv body\ p = Seq\ s_L\ s_H$

— The order of the two statements in the body is arbitrary.

The splitting reduces the well-formedness of the body to the well-formedness of the two statements.

**lemma**  $reduce\text{-}wf\text{-}body\text{-}to\text{-}stmts:$

$body\text{-}split\ p\ s_L\ s_H \implies wfs\ \Gamma_0\ (body\ p) \longleftrightarrow wfs\ \Gamma_0\ s_L \wedge wfs\ \Gamma_0\ s_H$   
 $\langle proof \rangle$

The processing predicates over programs are refined to predicates over the statements  $s_L$  and  $s_H$ . Since  $s_H$  follows  $s_L$ :

- $s_H$  must not change the low output, which is computed by  $s_L$ .
- $s_L$  must not change the low and high inputs, which are used by  $s_H$ .

**definition**  $low\text{-}proc\text{-}non0_2 :: stmt \Rightarrow bool$

**where**  $low\text{-}proc\text{-}non0_2 s_L \equiv$

$\forall \sigma \sigma'.$   
 $match \sigma \Gamma_0 \wedge$   
 $the (\sigma \text{ "lowIn"}) \neq 0 \wedge$   
 $s_L \triangleright \sigma \rightsquigarrow Some \sigma' \longrightarrow$   
 $the (\sigma' \text{ "lowOut"}) = the (\sigma \text{ "lowIn"}) + 1$

**definition**  $low\text{-}proc\text{-}0_2 :: stmt \Rightarrow bool$

**where**  $low\text{-}proc\text{-}0_2 s_L \equiv$

$\forall \sigma u.$   
 $match \sigma \Gamma_0 \wedge$   
 $the (\sigma \text{ "lowIn"}) = 0 \longrightarrow$   
 $(\exists \sigma'. s_L \triangleright \sigma \rightsquigarrow Some \sigma' \wedge the (\sigma' \text{ "lowOut"}) = u)$

**definition**  $low\text{-}proc\text{-}no\text{-}input\text{-}change :: stmt \Rightarrow bool$

**where**  $low\text{-}proc\text{-}no\text{-}input\text{-}change s_L \equiv$

$\forall \sigma \sigma'.$   
 $match \sigma \Gamma_0 \wedge$   
 $s_L \triangleright \sigma \rightsquigarrow Some \sigma' \longrightarrow$   
 $the (\sigma' \text{ "lowIn"}) = the (\sigma \text{ "lowIn"}) \wedge$   
 $the (\sigma' \text{ "highIn"}) = the (\sigma \text{ "highIn"})$

**definition**  $high\text{-}proc_2 :: stmt \Rightarrow bool$

**where**  $high\text{-}proc_2 s_H \equiv$

$\forall \sigma \sigma'.$   
 $match \sigma \Gamma_0 \wedge$   
 $s_H \triangleright \sigma \rightsquigarrow Some \sigma' \longrightarrow$   
 $the (\sigma' \text{ "highOut"}) \geq the (\sigma \text{ "lowIn"}) + the (\sigma \text{ "highIn"})$

**definition**  $high\text{-}proc\text{-}no\text{-}low\text{-}output\text{-}change :: stmt \Rightarrow bool$

**where**  $high\text{-}proc\text{-}no\text{-}low\text{-}output\text{-}change s_H \equiv$

$\forall \sigma \sigma'.$   
 $match \sigma \Gamma_0 \wedge$   
 $s_H \triangleright \sigma \rightsquigarrow Some \sigma' \longrightarrow$   
 $the (\sigma' \text{ "lowOut"}) = the (\sigma \text{ "lowOut"})$

**lemma**  $proc_2\text{-}correct:$

**assumes**  $Body: body\text{-}split p s_L s_H$

**assumes**  $WfLow: wfs \Gamma_0 s_L$

**assumes**  $WfHigh: wfs \Gamma_0 s_H$

**assumes**  $LowNon0: low\text{-}proc\text{-}non0_2 s_L$

**assumes** *Low0*: *low-proc-0<sub>2</sub>*  $s_L$   
**assumes** *LowSame*: *low-proc-no-input-change*  $s_L$   
**assumes** *High*: *high-proc<sub>2</sub>*  $s_H$   
**assumes** *HighSame*: *high-proc-no-low-output-change*  $s_H$   
**shows** *low-proc-non0<sub>1</sub>*  $p \wedge$  *low-proc-0<sub>1</sub>*  $p \wedge$  *high-proc<sub>1</sub>*  $p$   
 $\langle$ *proof* $\rangle$

The refined specification consists of the splitting of the body into the two sequential statements and the refined well-formedness and processing constraints.

**definition** *spec<sub>2</sub>* :: *prog*  $\Rightarrow$  *bool*

**where** *spec<sub>2</sub>*  $p \equiv$   
 $vars\ p = vars_0 \wedge$   
 $(\exists s_L\ s_H.$   
 $\quad body\_split\ p\ s_L\ s_H \wedge$   
 $\quad wfs\ \Gamma_0\ s_L \wedge$   
 $\quad wfs\ \Gamma_0\ s_H \wedge$   
 $\quad low\_proc\_non0_2\ s_L \wedge$   
 $\quad low\_proc\_0_2\ s_L \wedge$   
 $\quad low\_proc\_no\_input\_change\ s_L \wedge$   
 $\quad high\_proc_2\ s_H \wedge$   
 $\quad high\_proc\_no\_low\_output\_change\ s_H)$

**lemma** *step-2-correct*:

$spec_2\ p \implies spec_1\ p$   
 $\langle$ *proof* $\rangle$

### 3.4.3 Step 3

The processing constraints *low-proc-non0<sub>2</sub>* and *low-proc-0<sub>2</sub>* on  $s_L$  suggest the use of a conditional that randomizes "*lowOut*" if "*lowIn*" is 0, and stores 1 plus "*lowIn*" into "*lowOut*" otherwise.

**abbreviation**  $s_{L0}$  :: *stmt*

**where**  $s_{L0} \equiv$   
 $IfEq$   
 $\quad (Var\ "lowIn")$   
 $\quad (Const\ 0)$   
 $\quad (Random\ "lowOut")$   
 $\quad (Assign\ "lowOut"\ (Add\ (Var\ "lowIn")\ (Const\ 1)))$

**lemma** *wfs-s<sub>L0</sub>*:

$wfs\ \Gamma_0\ s_{L0}$   
 $\langle$ *proof* $\rangle$

**lemma** *low-proc-non0-s<sub>L0</sub>*:

$low\_proc\_non0_2\ s_{L0}$

$\langle proof \rangle$

**lemma** *low-proc-0-s<sub>L0</sub>*:

*low-proc-0<sub>2</sub> s<sub>L0</sub>*

$\langle proof \rangle$

**lemma** *low-proc-no-input-change-s<sub>L0</sub>*:

*low-proc-no-input-change s<sub>L0</sub>*

$\langle proof \rangle$

The refined specification is obtained by simplification using the definition of *s<sub>L</sub>*.

**definition** *spec<sub>3</sub> :: prog ⇒ bool*

**where** *spec<sub>3</sub> p ≡*

*vars p = vars<sub>0</sub> ∧*

*(∃ s<sub>H</sub>.*

*body-split p s<sub>L0</sub> s<sub>H</sub> ∧*

*wfs Γ<sub>0</sub> s<sub>H</sub> ∧*

*high-proc<sub>2</sub> s<sub>H</sub> ∧*

*high-proc-no-low-output-change s<sub>H</sub>)*

**lemma** *step-3-correct*:

*spec<sub>3</sub> p ⇒ spec<sub>2</sub> p*

$\langle proof \rangle$

The non-determinism required by *low-proc-0* cannot be pop-refined away. In particular, *s<sub>L</sub>* cannot be defined to copy the high input to the low output when the low input is 0, which would lead to a program that does not satisfy GNI.

### 3.4.4 Step 4

The processing constraint *high-proc<sub>2</sub>* on *s<sub>H</sub>* can be satisfied in different ways. A simple way is to pick the sum of the low and high inputs: *high-proc<sub>2</sub>* is refined by replacing the inequality with an equality.

**definition** *high-proc<sub>4</sub> :: stmt ⇒ bool*

**where** *high-proc<sub>4</sub> s<sub>H</sub> ≡*

*∀ σ σ'.*

*match σ Γ<sub>0</sub> ∧*

*s<sub>H</sub> ▷ σ ⇝ Some σ' →*

*the (σ' "highOut") = the (σ "lowIn") + the (σ "highIn")*

**lemma** *high-proc<sub>4</sub>-correct*:

*high-proc<sub>4</sub> s<sub>H</sub> ⇒ high-proc<sub>2</sub> s<sub>H</sub>*

$\langle proof \rangle$

The refined specification is obtained by substituting the refined processing constraint on *s<sub>H</sub>*.

**definition**  $spec_4 :: prog \Rightarrow bool$

**where**  $spec_4 p \equiv$

$vars\ p = vars_0 \wedge$   
 $(\exists s_H.$   
   $body-split\ p\ s_{L0}\ s_H \wedge$   
   $wfs\ \Gamma_0\ s_H \wedge$   
   $high-proc_4\ s_H \wedge$   
   $high-proc-no-low-output-change\ s_H)$

**lemma** *step-4-correct*:

$spec_4\ p \implies spec_3\ p$   
 $\langle proof \rangle$

### 3.4.5 Step 5

The refined processing constraint  $high-proc_4$  on  $s_H$  suggest the use of an assignment that stores the sum of "lowIn" and "highIn" into "highOut".

**abbreviation**  $s_{H0} :: stmt$

**where**  $s_{H0} \equiv Assign\ "highOut"\ (Add\ (Var\ "lowIn")\ (Var\ "highIn"))$

**lemma** *wfs-s<sub>H0</sub>*:

$wfs\ \Gamma_0\ s_{H0}$   
 $\langle proof \rangle$

**lemma** *high-proc<sub>4</sub>-s<sub>H0</sub>*:

$high-proc_4\ s_{H0}$   
 $\langle proof \rangle$

**lemma** *high-proc-no-low-output-change-s<sub>H0</sub>*:

$high-proc-no-low-output-change\ s_{H0}$   
 $\langle proof \rangle$

The refined specification is obtained by simplification using the definition of  $s_H$ .

**definition**  $spec_5 :: prog \Rightarrow bool$

**where**  $spec_5 p \equiv vars\ p = vars_0 \wedge body-split\ p\ s_{L0}\ s_{H0}$

**lemma** *step-5-correct*:

$spec_5\ p \implies spec_4\ p$   
 $\langle proof \rangle$

### 3.4.6 Step 6

$spec_5$ , which defines the variables and the body, is refined to characterize a unique program in explicit syntactic form.

**abbreviation**  $p_0 :: prog$   
**where**  $p_0 \equiv (\langle vars = vars_0, body = Seq s_{L0} s_{H0} \rangle)$

**definition**  $spec_6 :: prog \Rightarrow bool$   
**where**  $spec_6 p \equiv p = p_0$

**lemma** *step-6-correct*:  
 $spec_6 p \implies spec_5 p$   
*<proof>*

The program satisfies  $spec_0$  by construction. The program witnesses the consistency of the requirements, i.e. the fact that  $spec_0$  is not always false.

**lemma** *p<sub>0</sub>-sat-spec<sub>0</sub>*:  
 $spec_0 p_0$   
*<proof>*

From  $p_0$ , the program text

```
prog {
  vars {
    lowIn
    lowOut
    highIn
    highOut
  }
  body {
    if (lowIn == 0) {
      randomize lowOut;
    } else {
      lowOut = lowIn + 1;
    }
    highOut = lowIn + highIn;
  }
}
```

is easily obtained.

# Chapter 4

## General Remarks

The following remarks apply to pop-refinement in general, beyond the examples in Chapter 2 and Chapter 3.

### 4.1 Program-Level Requirements

By predicating directly over programs, a pop-refinement specification (like  $spec_0$  in Section 2.2 and Section 3.3) can express program-level requirements that are defined in terms of the vocabulary of the target language, e.g. constraints on memory footprint (important for embedded software), restrictions on calls to system libraries to avoid or limit information leaks (important for security), conformance to coding standards (important for certain certifications), and use or provision of interfaces (important for integration with existing code). Simple examples are  $wfp\ p$  in Section 2.2 and Section 3.3,  $para\ p = [\"x\", \"y\"]$  in Section 2.2, and  $iovars\ p$  in Section 3.3.

### 4.2 Non-Functional Requirements

Besides functional requirements, a pop-refinement specification can express non-functional requirements, e.g. constraints on computational complexity, timing, power consumption, etc.<sup>1</sup> A simple example is  $costp\ p \leq 3$  in Section 2.2.

---

<sup>1</sup>In order to express these requirements, the formalized semantics of the target language must suitably include non-functional aspects, as in the simple model in Section 2.1.4.



### 4.3 Links with High-Level Requirements

A pop-refinement specification can explicate links between high-level requirements and target programs.

For example,  $\forall x y. \text{exec } p [x, y] = \text{Some } (f x y)$  in  $\text{spec}_0$  in Section 2.2 links the high-level functional requirement expressed by  $f$  to the target program  $p$ .<sup>2</sup>

As another example, a function  $\text{sort} :: \text{nat list} \Rightarrow \text{nat list}$ , defined to map each list of natural numbers to its sorted permutation, expresses a high-level functional requirement that can be realized in different ways. An option is a procedure that destructively sorts an array in place. Another option is a procedure that returns a newly created sorted linked list from a linked list passed as argument and left unmodified. A pop-refinement specification can pin down the choice, which matters to external code that uses the procedure.

As a third example, a high-level model of a video game or physical simulator could use real numbers and differential equations. A pop-refinement specification could state required bounds on how the idealized model is approximated by an implementation that uses floating point numbers and difference equations.

Different pop-refinement specifications could use the same high-level requirements to constrain programs in different target languages or in different ways, as in the *sort* example above. As another example, the high-level behavior of an operating system could be described by a state transition system that abstractly models internal states and system calls; the same state transition system could be used in a pop-refinement specification of a Haskell simulator that runs on a desktop, as well as in a pop-refinement specification of a C/Assembly implementation that runs on a specific hardware platform.

### 4.4 Non-Determinism and Under-Specification

The interaction of refinement with non-determinism and under-specification is delicate in general. The one-to-many associations of a relational specification (e.g. a state transition system where the next-state relation may associate multiple new states to each old state) could be interpreted as non-determinism (i.e. different outcomes at different times, from the same state) or under-specification (i.e. any outcome is allowed, deterministically or non-deterministically). Hyperproperties like GNI are consistent with the interpretation as non-determinism, because security depends on the ability to yield different outcomes, e.g. generating a nonce in a cryptographic protocol. The popular notion of refinement

---

<sup>2</sup>Similarly, the functional requirements in Section 3.3 could be expressed abstractly in terms of mappings between low and high inputs and outputs (without reference to program variables and executions) and linked to program variables and executions. But Section 3.3 expresses such functional requirements directly in terms of programs to keep the example (whose focus is on hyperproperties) simpler.

as inclusion of sets of traces (e.g. [1]) is consistent with the interpretation as under-specification, because a refined specification is allowed to reduce the possible outcomes. Thus, hyperproperties are not always preserved by refinement as trace set inclusion [3].

As exemplified in Section 3.3, a pop-refinement specification can explicitly distinguish non-determinism and under-specification. Each pop-refinement step preserves all the hyperproperties expressed or implied by the requirement specification.<sup>3</sup>

## 4.5 Specialized Formalisms

Specialized formalisms (e.g. state machines, temporal logic), shallowly or deeply embedded into the logic of the theorem prover (e.g. [7, 6]), can be used to express some of the requirements of a pop-refinement specification. The logic of the theorem prover provides semantic integration of different specialized formalisms.

## 4.6 Strict and Non-Strict Refinement Steps

In a pop-refinement step from  $spec_i$  to  $spec_{i+1}$ , the two predicates may be equivalent, i.e.  $spec_{i+1} = spec_i$ . But the formulation of  $spec_{i+1}$  should be “closer” to the implementation than the formulation of  $spec_i$ . An example is in Section 2.3.1.

When the implication  $spec_{i+1} p \implies spec_i p$  is strict, potential implementations are eliminated. Since the final predicate of a pop-refinement derivation must characterize a unique program, some refinement steps must be strict—unless the initial predicate  $spec_0$  is satisfiable by a unique program, which is unlikely.

A strict refinement step may lead to a blind alley where  $spec_{i+1} = \lambda p. False$ , which cannot lead to a final predicate that characterizes a unique program. An example is discussed in Section 2.3.4.

## 4.7 Final Predicate

The predicate that concludes a pop-refinement derivation must have the form  $spec_n p \equiv p = p_0$ , where  $p_0$  is the representation of a program’s abstract syntax

---

<sup>3</sup>Besides security hyperproperties expressed in terms of non-determinism, pop-refinement can handle more explicit security randomness properties. The formalized semantics of a target language could manipulate probability distributions over values (instead of just values), with random number generation libraries that return known distributions (e.g. uniform), and with language operators that transform distributions. A pop-refinement specification could include randomness requirements on program outcomes expressed in terms of distributions, and each pop-refinement step would preserve such requirements.

in the theorem prover, as in Section 2.3.7 and Section 3.4.6. This form guarantees that the predicate characterizes exactly one program and that the program is explicitly determined.  $p_0$  witnesses the consistency of the requirements, i.e. the fact that  $spec_0$  is not always false; inconsistent requirements cannot lead to a predicate of this form.

A predicate of the form  $spec_i \equiv p = p_0 \wedge \Phi p$  may not characterize a unique program: if  $\Phi p_0$  is false,  $spec_i$  is always false. To conclude the derivation,  $\Phi p_0$  must be proved. But it may be easier to prove the constraints expressed by  $\Phi$  as  $p_0$  is constructed in the derivation. For example, deriving a program from  $spec_0$  in Section 2.2 based on the functional constraint and ignoring the cost constraint would lead to a predicate  $spec_i \equiv p = p_0 \wedge costp p \leq 3$ , where  $costp p_0 \leq 3$  must be proved to conclude the derivation; instead, the derivation in Section 2.3 proves the cost constraint as  $p_0$  is constructed. Taking all constraints into account at each stage of the derivation can help choose the next refinement step and reduce the chance of blind alleys (cf. Section 2.3.4).

The final predicate  $spec_n$  expresses a purely syntactic requirement, while the initial predicate  $spec_0$  usually includes semantic requirements. A pop-refinement derivation progressively turns semantic requirements into syntactic requirements. This may involve rephrasing functional requirements to use only operations supported by the target language (e.g. lemma *f-rephrased* in Section 2.3.4), obtaining shallowly embedded program fragments, and turning them into their deeply embedded counterparts (e.g. Section 2.3.5 and Section 2.3.6).<sup>4</sup>

## 4.8 Proof Coverage

In a chain of predicate inclusions as in Section 2.3 and Section 3.4, the proofs checked by the theorem prover encompass the range from the specified requirements to the implementation code. No separate code generator is needed to turn low-level specifications into code: pop-refinement folds code generation into the refinement sequence, providing fine-grained control on the implementation code.

A purely syntactic pretty-printer is needed to turn program abstract syntax, as in Section 2.3.7 and Section 3.4.6, to concrete syntax. This pretty-printer can be eliminated by formalizing in the theorem prover the concrete syntax of the target language and its relation to the abstract syntax, and by defining the  $spec_i$  predicates over program concrete syntax—thus, folding pretty-printing into the refinement sequence.

---

<sup>4</sup>In Section 3.4, program fragments are introduced directly, without going through shallow embeddings.

## 4.9 Generality and Flexibility

Inclusion of predicates over programs is a general and flexible notion of refinement. More specialized notions of refinement (e.g. [8, 14]) can be used for any auxiliary types, functions, etc. out of which the  $spec_i$  predicates may be constructed, as long as the top-level implication  $spec_{i+1} p \implies spec_i p$  holds at every step.

## Chapter 5

# Related Work

In existing approaches to stepwise refinement (e.g. [2, 17, 9, 15]), specifications express requirements less directly than pop-refinement: a specification implicitly characterizes its possible implementations as the set of programs that can be derived from the specification via refinement (and code generation). This is a more restrictive way to characterize a set of programs than defining a predicate over deeply embedded programs in a theorem prover’s general-purpose logic (as in pop-refinement).

This restrictiveness precludes some of the abilities discussed in Chapter 4, e.g. the ability to express, and guarantee through refinement, certain program-level requirements like constraints on memory footprint. A derivation may be steered to produce a program that satisfies desired requirements not expressed by the specification, but the derivation or program must be examined in order to assess that, instead of just examining the specification and letting the theorem prover automatically check the sequence of refinement steps (as with pop-refinement). Existing refinement approaches could be extended to handle additional kinds of requirements (e.g. non-functional), but for pop-refinement no theorem prover extensions are necessary.

In existing refinement approaches, each refinement step yields a new specification that characterizes a (strict or non-strict) subset of the implementations characterized by the old specification, analogously to pop-refinement. However, the restrictiveness explained above, together with any inherent constraints imposed by the refinement relation over specifications, limits the choice of the subset, providing less fine-grained control than pop-refinement.

In existing refinement approaches, the “indirection” between a specification and its set of implementations may create a disconnect between properties of a specification and properties of its implementations. For example, along the lines discussed in Section 4.4, a relational specification may satisfy a hyperproperty but some of its implementations may not, because the refinement relation may reduce the possible behaviors. Since a pop-refinement specification directly

makes statements about the possible implementations of the requirements, this kind of disconnect is avoided.

# Chapter 6

## Future Work

### 6.1 Populating the Framework

Pop-refinement provides a framework, which must be populated with re-usable concepts, methodologies, and theorem prover libraries for full fruition. The simple examples in Chapter 2 and Chapter 3, and the discussion in Chapter 4, suggests a few initial ideas. Working out examples of increasing complexity should suggest more ideas.

### 6.2 Automated Transformations

A pop-refinement step from  $spec_i$  can be performed manually, by writing down  $spec_{i+1}$  and proving  $spec_{i+1} p \implies spec_i p$ . It is sometimes possible to generate  $spec_{i+1}$  from  $spec_i$ , along with a proof of  $spec_{i+1} p \implies spec_i p$ , using automated transformation techniques like term rewriting, application of algorithmic templates, and term construction by witness finding, e.g. [16, 10]. Automated transformations may require parameters to be provided and applicability conditions to be proved, but should generally save effort and make derivations more robust against changes in requirement specifications. Extending existing theorem provers with automated transformation capabilities would be advantageous for pop-refinement.

### 6.3 Other Kinds of Design Objects

It has been suggested [13] that pop-refinement could be used to develop other kinds of design objects than programs, e.g. protocols, digital circuits, and hybrid systems. Perhaps pop-refinement could be used to develop engines, cars,

buildings, etc. So long as these design objects can be described by languages amenable to formalization, pop-refinement should be applicable.



# Bibliography

- [1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Journal of Theoretical Computer Science*, 82(2):253–284, 1991.
- [2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [3] Michael Clarkson and Fred Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
- [4] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8(3):174–186, 1968.
- [5] Joseph Goguen and José Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
- [6] Gudmund Grov and Stephan Merz. A definitional encoding of TLA\* in Isabelle/HOL. *Archive of Formal Proofs*, 2011. <http://isa-afp.org/entries/TLA.shtml>, Formal proof development.
- [7] Steffen Helke and Florian Kammüller. Formalizing Statecharts using hierarchical automata. *Archive of Formal Proofs*, 2010. <http://isa-afp.org/entries/Statecharts.shtml>, Formal proof development.
- [8] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [9] Cliff Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.
- [10] Kestrel Institute. Specware. <http://www.specware.org>.
- [11] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
- [12] John McLean. A general theory of composition for a class of “possibilistic” properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.
- [13] Lambert Meertens. Private communication, 2012.

- [14] Robin Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Stanford University, 1971.
- [15] Carroll Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1998.
- [16] Douglas R. Smith. Mechanizing the development of software. In Manfred Broy, editor, *Calculational System Design, Proc. Marktoberdorf Summer School*. IOS Press, 1999.
- [17] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
- [18] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.