Pop-Refinement

Alessandro Coglio

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

July 1, 2014

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

1 Definition **3**

2 First Example **4**
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

3 Second Example **15**
3.1 Hyperproperties . . . 15
3.2 Target Programming Language . . . 16
3.2.1 Syntax . . . 17
3.2.2 Static Semantics . . . 17
3.2.3 Dynamic Semantics . . . 18
3.3 Requirement Specification . . . 23
3.3.1 Input/Output Variables . . . 23
3.3.2 Low Processing . . . 23
3.3.3 High Processing . . . 24
3.3.4 All Requirements . . . 24
3.3.5 Generalized Non-Interference . . . 24
3.4 Stepwise Refinement . . . 27
3.4.1 Step 1 . . . 27
3.4.2 Step 2 . . . 29
3.4.3 Step 3 . . . 33

3.4.4 Step 4 35
3.4.5 Step 5 35
3.4.6 Step 6 36

4 General Remarks **38**
4.1 Program-Level Requirements 38
4.2 Non-Functional Requirements 38
4.3 Links with High-Level Requirements 39
4.4 Non-Determinism and Under-Specification 39
4.5 Specialized Formalisms 40
4.6 Strict and Non-Strict Refinement Steps 40
4.7 Final Predicate 40
4.8 Proof Coverage 41
4.9 Generality and Flexibility 42

5 Related Work **43**

6 Future Work **45**
6.1 Populating the Framework 45
6.2 Automated Transformations 45
6.3 Other Kinds of Design Objects 45

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 $env = name \rightharpoonup nat$

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 $U_1 \otimes U_2 \equiv$
  $\mathit{case}\ (U_1,\ U_2)\ \mathit{of}\ (\mathit{Some}\ u_1,\ \mathit{Some}\ u_2) \Rightarrow \mathit{Some}\ (u_1 * u_2) \mid \text{-} \Rightarrow \mathit{None}$

definition *add-opt* :: *nat option* $\Rightarrow$ *nat option* $\Rightarrow$ *nat option* (**infixl** $\oplus$ *65*)
— Lifting of addition to *nat option*.
where $U_1 \oplus U_2 \equiv$
  $\mathit{case}\ (U_1,\ U_2)\ \mathit{of}\ (\mathit{Some}\ u_1,\ \mathit{Some}\ u_2) \Rightarrow \mathit{Some}\ (u_1 + u_2) \mid \text{-} \Rightarrow \mathit{None}$

fun *eval* :: *env* $\Rightarrow$ *expr* $\Rightarrow$ *nat option*
where
  $\mathit{eval}\ \mathcal{E}\ (\mathit{Const}\ c) = \mathit{Some}\ c \mid$
  $\mathit{eval}\ \mathcal{E}\ (\mathit{Var}\ v) = \mathcal{E}\ v \mid$
  $\mathit{eval}\ \mathcal{E}\ (\mathit{Double}\ e) = \mathit{Some}\ \mathit{2} \otimes \mathit{eval}\ \mathcal{E}\ e \mid$
  $\mathit{eval}\ \mathcal{E}\ (\mathit{Add}\ e_1\ e_2) = \mathit{eval}\ \mathcal{E}\ e_1 \oplus \mathit{eval}\ \mathcal{E}\ e_2$

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

lemma *eval-wfe*:
  $\mathit{wfe}\ \Gamma\ e \Longrightarrow \mathit{match}\ \mathcal{E}\ \Gamma \Longrightarrow \mathit{eval}\ \mathcal{E}\ e \neq \mathit{None}$
by (*induct e*, *auto simp*: *match-def mul-opt-def add-opt-def*)

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

definition *envs* :: *prog* $\Rightarrow$ *env set*
where $\mathit{envs}\ p \equiv \{\mathcal{E}.\ \mathit{match}\ \mathcal{E}\ (\mathit{ctxt}\ p)\}$

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

lemma *eval-wfp*:
  $\mathit{wfp}\ p \Longrightarrow \mathcal{E} \in \mathit{envs}\ p \Longrightarrow \mathit{eval}\ \mathcal{E}\ (\mathit{body}\ p) \neq \mathit{None}$
by (*metis envs-def eval-wfe mem-Collect-eq wfp-def*)

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* $\equiv$

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

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

lemma *supply-wfp*:
$\mathit{wfp\ p} \Longrightarrow$
$\mathit{length\ us} = \mathit{length\ (para\ p)} \Longrightarrow$
$\exists \mathcal{E} \in \mathit{envs\ p}.\ \mathit{supply\ p\ us} = \mathit{Some}\ \mathcal{E}$
by (*auto*
simp: *wfp-def supply-def envs-def ctxt-def match-def split*: *option.split*)

lemma *exec-wfp*:
$\mathit{wfp\ p} \Longrightarrow \mathit{length\ us} = \mathit{length\ (para\ p)} \Longrightarrow \mathit{exec\ p\ us} \neq \mathit{None}$
by (*metis eval-wfp exec-def option.simps*(*5*) *supply-wfp*)

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* ⇒ *nat*
where
$\mathit{coste}\ (\mathit{Const\ c}) = 0\ |$
$\mathit{coste}\ (\mathit{Var\ v}) = 0\ |$
$\mathit{coste}\ (\mathit{Double\ e}) = 1 + \mathit{coste\ e}\ |$
$\mathit{coste}\ (\mathit{Add}\ e_1\ e_2) = 1 + \mathit{coste}\ e_1 + \mathit{coste}\ e_2$

definition *costp* :: *prog* ⇒ *nat*
where $\mathit{costp\ p} \equiv \mathit{coste}\ (\mathit{body\ p})$

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

definition $\mathit{spec}_0 :: \mathit{prog} \Rightarrow \mathit{bool}$
where $\mathit{spec}_0\ p \equiv$
$\mathit{wfp}\ p\ \wedge$
$\mathit{para}\ p = [''x'',\ ''y''] \wedge$
$(\forall\, x\ y.\ \mathit{exec}\ p\ [x,\ y] = \mathit{Some}\ (f\ x\ y)) \wedge$
$\mathit{costp}\ p \leq \mathit{3}$

f is used by spec_0 to express a functional requirement on the execution of the program. spec_0 includes the non-functional requirement $\mathit{costp}\ p \leq \mathit{3}$ and the syntactic interface requirement $\mathit{para}\ p = [''x'',\ ''y'']$, 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 $\Gamma_{xy} :: \mathit{ctxt}$
where $\Gamma_{xy} \equiv \{''x'',\ ''y''\}$

abbreviation $\mathcal{E}_{xy} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{env}$
where $\mathcal{E}_{xy}\ x\ y \equiv [''x'' \mapsto x,\ ''y'' \mapsto y]$

lemma *reduce-prog-to-body*:
$\mathit{para}\ p = [''x'',\ ''y''] \Longrightarrow$
$\mathit{wfp}\ p = \mathit{wfe}\ \Gamma_{xy}\ (\mathit{body}\ p) \wedge$
$\mathit{exec}\ p\ [x,\ y] = \mathit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ (\mathit{body}\ p)$
by (*auto simp*: *wfp-def ctxt-def exec-def supply-def fun-upd-twist*)

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 $spec_1\ p \equiv$
$wfe\ \Gamma_{xy}\ (body\ p) \wedge$
$para\ p = [''x'', ''y''] \wedge$
$(\forall x\ y.\ eval\ (\mathcal{E}_{xy}\ x\ y)\ (body\ p) = Some\ (f\ x\ y)) \wedge$
$coste\ (body\ p) \leq 3$

lemma *step-1-correct*:
$spec_1\ p \Longrightarrow spec_0\ p$
by (*auto simp*: $spec_1$*-def* $spec_0$*-def reduce-prog-to-body costp-def*)

$spec_1$ 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 $spec_2\ p \equiv$
$para\ p = [''x'', ''y''] \wedge$
$(\exists\, e_1\ e_2.$
$body\ p = Add\ e_1\ e_2 \wedge$
$wfe\ \Gamma_{xy}\ (body\ p) \wedge$
$(\forall x\ y.\ eval\ (\mathcal{E}_{xy}\ x\ y)\ (body\ p) = Some\ (f\ x\ y)) \wedge$
$coste\ (body\ p) \leq 3)$

lemma *step-2-correct*:
$spec_2\ p \Longrightarrow spec_1\ p$
by (*auto simp*: $spec_2$*-def* $spec_1$*-def*)

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*:
  $\textit{body}\ p = \textit{Add}\ e_1\ e_2 \Longrightarrow$
  $\textit{wfe}\ \Gamma_{xy}\ (\textit{body}\ p) = (\textit{wfe}\ \Gamma_{xy}\ e_1 \wedge \textit{wfe}\ \Gamma_{xy}\ e_2) \wedge$
  $\textit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ (\textit{body}\ p) = \textit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ e_1 \oplus \textit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ e_2 \wedge$
  $\textit{coste}\ (\textit{body}\ p) = \mathit{1} + \textit{coste}\ e_1 + \textit{coste}\ e_2$
by *auto*

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

definition $\textit{spec}_3$ :: *prog* $\Rightarrow$ *bool*
where $\textit{spec}_3\ p \equiv$
  $\textit{para}\ p = [''x'', ''y''] \wedge$
  $(\exists\, e_1\ e_2.$
    $\textit{body}\ p = \textit{Add}\ e_1\ e_2 \wedge$
    $\textit{wfe}\ \Gamma_{xy}\ e_1 \wedge$
    $\textit{wfe}\ \Gamma_{xy}\ e_2 \wedge$
    $(\forall\, x\ y.\ \textit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ e_1 \oplus \textit{eval}\ (\mathcal{E}_{xy}\ x\ y)\ e_2 = \textit{Some}\ (f\ x\ y)) \wedge$
    $\textit{coste}\ e_1 + \textit{coste}\ e_2 \leq \mathit{2})$

lemma *step-3-correct*:
  $\textit{spec}_3\ p \Longrightarrow \textit{spec}_2\ p$
by (*auto simp*: $\textit{spec}_3$*-def* $\textit{spec}_2$*-def*)
— No need to use *reduce-body-to-addends* explicitly,
— as the default rules that *auto* uses to prove it apply here too.

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 $\textit{spec}_3$ suggests to split $f\ x\ y$ into two addends to be computed by e_1 and e_2.

The addends $\mathit{3} * x$ and $\mathit{2} * y$ suggested by the definition of f would lead to a blind alley, where the cost constraints could not be satisfied—the resulting $\textit{spec}_4$ 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 $\textit{spec}_3$ and split $f\ x\ y$ differently.

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

lemma *f-rephrased*:

$f\ x\ y = x + (2 * x + 2 * y)$
by (*auto simp*: *f-def*)

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*:
$spec_4\ p \Longrightarrow spec_3\ p$
by (*auto simp*: $spec_4$*-def* $spec_3$*-def add-opt-def f-rephrased*)

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*:
$e_1 = Var\ ''x'' \Longrightarrow$
$eval\ (\mathcal{E}_{xy}\ x\ y)\ e_1 = Some\ x \wedge$
$wfe\ \Gamma_{xy}\ e_1 \wedge$
$coste\ e_1 = 0$
by *auto*

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 $spec_5\ p \equiv$
$para\ p = [''x'', ''y''] \wedge$
$(\exists\ e_2.$
$body\ p = Add\ (Var\ ''x'')\ e_2 \wedge$
$wfe\ \Gamma_{xy}\ e_2 \wedge$
$(\forall\ x\ y.\ eval\ (\mathcal{E}_{xy}\ x\ y)\ e_2 = Some\ (2 * x + 2 * y)) \wedge$
$coste\ e_2 \leq 2)$

lemma *step-5-correct*:
$spec_5\ p \Longrightarrow spec_4\ p$
by (*auto simp*: $spec_5$*-def* $spec_4$*-def*)
— No need to use *first-addend* explicitly,
— as the default rules that *auto* uses to prove it apply here too.

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 $spec_5$ is a shallow embedding of the program expression `2 * x + 2 * y`, whose deep embedding is the term $Add\ (Double\ (Var\ ''x''))\ (Double\ (Var\ ''y''))$. Using the latter as e_2, the second-to-last conjunct in $spec_5$ 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*:
$(2{::}nat) * x + 2 * y = 2 * (x + y)$
by *auto*

The deeply embedded expression $Double\ (Add\ (Var\ ''x'')\ (Var\ ''y''))$, which corresponds to the shallowly embedded expression $2 * (x + y)$, satisfies the second-to-last conjunct of $spec_5$, is well-formed, and has cost 2.

lemma *second-addend*:
$e_2 = Double\ (Add\ (Var\ ''x'')\ (Var\ ''y'')) \Longrightarrow$
$eval\ (\mathcal{E}_{xy}\ x\ y)\ e_2 = Some\ (2 * x + 2 * y) \wedge$
$wfe\ \Gamma_{xy}\ e_2 \wedge$
$coste\ e_2 = 2$
by (*auto simp*: *add-opt-def mul-opt-def*)
— No need to use *factorization* explicitly,
— as the default rules that *auto* uses to prove it apply here too.

Adding $e_2 = Double\ (Add\ (Var\ ''x'')\ (Var\ ''y''))$ as a conjunct to $spec_5$ and simplifying, $spec_5$ is refined as follows.

definition $spec_6 :: prog \Rightarrow bool$

where $\mathit{spec}_6\ p \equiv$
  $\mathit{para}\ p = [''x'', ''y''] \wedge$
  $\mathit{body}\ p = \mathit{Add}\ (\mathit{Var}\ ''x'')\ (\mathit{Double}\ (\mathit{Add}\ (\mathit{Var}\ ''x'')\ (\mathit{Var}\ ''y'')))$

lemma *step-6-correct*:
  $\mathit{spec}_6\ p \Longrightarrow \mathit{spec}_5\ p$
by (*auto simp add*: spec_6*-def* spec_5*-def second-addend simp del*: *eval.simps*)

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 $p_0 \equiv$
  $(\!|\mathit{para} = [''x'', ''y''],$
   $\mathit{body} = \mathit{Add}\ (\mathit{Var}\ ''x'')\ (\mathit{Double}\ (\mathit{Add}\ (\mathit{Var}\ ''x'')\ (\mathit{Var}\ ''y'')))|\!)$

definition spec_7 :: $\mathit{prog} \Rightarrow \mathit{bool}$
where $\mathit{spec}_7\ p \equiv p = p_0$

lemma *step-7-correct*:
  $\mathit{spec}_7\ p \Longrightarrow \mathit{spec}_6\ p$
by (*auto simp*: spec_7*-def* spec_6*-def*)

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:
  $\mathit{spec}_0\ p_0$
by (*metis*
 step-1-correct
 step-2-correct
 step-3-correct
 step-4-correct
 step-5-correct
 step-6-correct
 step-7-correct
 spec_7*-def*)

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 τ_1 and τ_2, there is always a trace τ_3 with the same high inputs as τ_1 and the same low inputs and low outputs as τ_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 τ_2 is indistinguishable from observing some other trace τ_3 that has the same high inputs as an arbitrary trace τ_1.

locale *generalized-non-interference* =
  fixes *low-in* :: $'\tau \Rightarrow 'i$ — low inputs
  fixes *low-out* :: $'\tau \Rightarrow 'o$ — low outputs
  fixes *high-in* :: $'\tau \Rightarrow 'i$ — high inputs
  fixes *high-out* :: $'\tau \Rightarrow 'o$ — high outputs

definition (**in** *generalized-non-interference*) *GNI* :: $'\tau$ *set* $\Rightarrow$ *bool*
where *GNI* $\mathcal{T} \equiv$
  $\forall \tau_1 \in \mathcal{T}.\ \forall \tau_2 \in \mathcal{T}.\ \exists \tau_3 \in \mathcal{T}.$
    $\textit{high-in}\ \tau_3 = \textit{high-in}\ \tau_1 \wedge \textit{low-in}\ \tau_3 = \textit{low-in}\ \tau_2 \wedge \textit{low-out}\ \tau_3 = \textit{low-out}\ \tau_2$

3.2 Target Programming Language

In the target language used in this example,[1] 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 $\{2, 3, \ldots\}$.

[1]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* | *Var name* | *Add expr expr*

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

datatype *stmt* =
  Assign name expr |
  Random name |
  IfEq expr expr stmt stmt |
  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* ⇒ *expr* ⇒ *bool*
where
  $wfe\ \Gamma\ (Const\ c) \longleftrightarrow True$ |
  $wfe\ \Gamma\ (Var\ v) \longleftrightarrow v \in \Gamma$ |
  $wfe\ \Gamma\ (Add\ e_1\ e_2) \longleftrightarrow wfe\ \Gamma\ e_1 \wedge wfe\ \Gamma\ e_2$

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* ⇒ *stmt* ⇒ *bool*
where
  $wfs\ \Gamma\ (Assign\ v\ e) \longleftrightarrow v \in \Gamma \wedge wfe\ \Gamma\ e$ |
  $wfs\ \Gamma\ (Random\ v) \longleftrightarrow v \in \Gamma$ |
  $wfs\ \Gamma\ (IfEq\ e_1\ e_2\ s_1\ s_2) \longleftrightarrow wfe\ \Gamma\ e_1 \wedge wfe\ \Gamma\ e_2 \wedge wfs\ \Gamma\ s_1 \wedge wfs\ \Gamma\ s_2$ |

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

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 $state = name \rightharpoonup nat$

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 $add\text{-}opt :: nat\ option \Rightarrow nat\ option \Rightarrow nat\ option$ (**infixl** $\oplus$ *65*)
— Lifting of addition to *nat option*.
where $U_1 \oplus U_2 \equiv$
$case\ (U_1,\ U_2)\ of\ (Some\ u_1,\ Some\ u_2) \Rightarrow Some\ (u_1 + u_2) \mid \text{-} \Rightarrow None$

fun $eval :: state \Rightarrow expr \Rightarrow nat\ option$
where
$eval\ \sigma\ (Const\ c) = Some\ c\ |$
$eval\ \sigma\ (Var\ v) = \sigma\ v\ |$
$eval\ \sigma\ (Add\ e_1\ e_2) = eval\ \sigma\ e_1 \oplus eval\ \sigma\ e_2$

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

lemma *eval-wfe*:
$wfe\ \Gamma\ e \Longrightarrow match\ \sigma\ \Gamma \Longrightarrow eval\ \sigma\ e \neq None$
by (*induct e*, *auto simp*: *match-def add-opt-def*)

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

($\text{-} \rhd \text{-} \leadsto \text{-}$ [*50*, *50*, *50*] *50*)
where
ExecAssignNoVar:
$v \notin dom\ \sigma \Longrightarrow Assign\ v\ e \rhd \sigma \leadsto None$ |
ExecAssignEvalError:
$eval\ \sigma\ e = None \Longrightarrow Assign\ v\ e \rhd \sigma \leadsto None$ |
ExecAssignOK:
$v \in dom\ \sigma \Longrightarrow$
$eval\ \sigma\ e = Some\ u \Longrightarrow$
$Assign\ v\ e \rhd \sigma \leadsto Some\ (\sigma(v \mapsto u))$ |
ExecRandomNoVar:
$v \notin dom\ \sigma \Longrightarrow Random\ v \rhd \sigma \leadsto None$ |
ExecRandomOK:
$v \in dom\ \sigma \Longrightarrow Random\ v \rhd \sigma \leadsto Some\ (\sigma(v \mapsto u))$ |
ExecCondEvalError1:
$eval\ \sigma\ e_1 = None \Longrightarrow IfEq\ e_1\ e_2\ s_1\ s_2 \rhd \sigma \leadsto None$ |
ExecCondEvalError2:
$eval\ \sigma\ e_2 = None \Longrightarrow IfEq\ e_1\ e_2\ s_1\ s_2 \rhd \sigma \leadsto None$ |
ExecCondTrue:
$eval\ \sigma\ e_1 = Some\ u_1 \Longrightarrow$
$eval\ \sigma\ e_2 = Some\ u_2 \Longrightarrow$
$u_1 = u_2 \Longrightarrow$
$s_1 \rhd \sigma \leadsto \varrho \Longrightarrow$
$IfEq\ e_1\ e_2\ s_1\ s_2 \rhd \sigma \leadsto \varrho$ |
ExecCondFalse:
$eval\ \sigma\ e_1 = Some\ u_1 \Longrightarrow$
$eval\ \sigma\ e_2 = Some\ u_2 \Longrightarrow$
$u_1 \neq u_2 \Longrightarrow$
$s_2 \rhd \sigma \leadsto \varrho \Longrightarrow$
$IfEq\ e_1\ e_2\ s_1\ s_2 \rhd \sigma \leadsto \varrho$ |
ExecSeqError:
$s_1 \rhd \sigma \leadsto None \Longrightarrow Seq\ s_1\ s_2 \rhd \sigma \leadsto None$ |
ExecSeqOK:
$s_1 \rhd \sigma \leadsto Some\ \sigma' \Longrightarrow s_2 \rhd \sigma' \leadsto \varrho \Longrightarrow Seq\ s_1\ s_2 \rhd \sigma \leadsto \varrho$

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

lemma *exec-always*:
$\exists\, \varrho.\ s \rhd \sigma \leadsto \varrho$
proof (*induct s arbitrary*: σ)
case *Assign*
show *?case*
by (*metis*
ExecAssignEvalError ExecAssignNoVar ExecAssignOK option.exhaust)
next
case *Random*
show *?case*

by (*metis ExecRandomNoVar ExecRandomOK*)
next
case *IfEq*
thus *?case*
by (*metis*
ExecCondEvalError1 ExecCondEvalError2 ExecCondFalse ExecCondTrue
option.exhaust)
next
case *Seq*
thus *?case*
by (*metis ExecSeqError ExecSeqOK option.exhaust*)
qed

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*:
$wfs\ \Gamma\ s \Longrightarrow match\ \sigma\ \Gamma \Longrightarrow s \rhd \sigma \leadsto Some\ \sigma' \Longrightarrow match\ \sigma'\ \Gamma$
proof (*induct s arbitrary*: $\sigma\ \sigma'$)
case (*Assign v e*)
then obtain u
where $eval\ \sigma\ e = Some\ u$
and $\sigma' = \sigma(v \mapsto u)$
by (*auto elim*: *exec.cases*)
with *Assign*
show *?case*
by (*metis*
domIff dom-fun-upd fun-upd-triv match-def option.distinct(*1*) *wfs.simps*(*1*))
next
case (*Random v*)
then obtain u
where $\sigma' = \sigma(v \mapsto u)$
by (*auto elim*: *exec.cases*)
with *Random*
show *?case*
by (*metis*
domIff dom-fun-upd fun-upd-triv match-def option.distinct(*1*) *wfs.simps*(*2*))
next
case (*IfEq* $e_1\ e_2\ s_1\ s_2$)
hence $s_1 \rhd \sigma \leadsto Some\ \sigma' \lor s_2 \rhd \sigma \leadsto Some\ \sigma'$
by (*blast elim*: *exec.cases*)
with *IfEq*
show *?case*
by (*metis wfs.simps*(*3*))
next
case (*Seq* $s_1\ s_2$)
then obtain σ_i

where $s_1 \rhd \sigma \leadsto \mathit{Some}\ \sigma_i$
and $s_2 \rhd \sigma_i \leadsto \mathit{Some}\ \sigma'$
by (*blast elim*: *exec.cases*)
with *Seq*
show *?case*
by (*metis wfs.simps(4)*)
qed

lemma *exec-wfs-no-error*:
$\mathit{wfs}\ \Gamma\ s \Longrightarrow \mathit{match}\ \sigma\ \Gamma \Longrightarrow \neg\ (s \rhd \sigma \leadsto \mathit{None})$
proof (*induct s arbitrary*: σ)
case (*Assign v e*)
hence *Var*: $v \in \mathit{dom}\ \sigma$
by (*auto simp*: *match-def*)
from *Assign*
have $\mathit{eval}\ \sigma\ e \neq \mathit{None}$
by (*metis eval-wfe wfs.simps(1)*)
with *Var*
show *?case*
by (*auto elim*: *exec.cases*)
next
case (*Random v*)
thus *?case*
by (*auto simp*: *match-def elim*: *exec.cases*)
next
case (*IfEq* e_1 e_2 s_1 s_2)
then obtain u_1 u_2
where $\mathit{eval}\ \sigma\ e_1 = \mathit{Some}\ u_1$
and $\mathit{eval}\ \sigma\ e_2 = \mathit{Some}\ u_2$
by (*metis eval-wfe not-Some-eq wfs.simps(3)*)
with *IfEq*
show *?case*
by (*auto elim*: *exec.cases*)
next
case (*Seq* s_1 s_2)
show *?case*
proof
assume $\mathit{Seq}\ s_1\ s_2 \rhd \sigma \leadsto \mathit{None}$
hence $s_1 \rhd \sigma \leadsto \mathit{None} \lor (\exists \sigma'.\ s_1 \rhd \sigma \leadsto \mathit{Some}\ \sigma' \land s_2 \rhd \sigma' \leadsto \mathit{None})$
by (*auto elim*: *exec.cases*)
with *Seq exec-wfs-match*
show *False*
by (*metis wfs.simps(4)*)
qed
qed

lemma *exec-wfs-always-match*:
  $wfs\ \Gamma\ s \Longrightarrow match\ \sigma\ \Gamma \Longrightarrow \exists \sigma'.\ s \rhd \sigma \rightsquigarrow Some\ \sigma' \wedge match\ \sigma'\ \Gamma$
by (*metis exec-always exec-wfs-match exec-wfs-no-error option.exhaust*)

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*:
  $wfp\ p \Longrightarrow \sigma \in states\ p \Longrightarrow \neg\ (body\ p \rhd \sigma \rightsquigarrow None)$
by (*metis exec-wfs-no-error mem-Collect-eq states-def wfp-def*)

lemma *exec-wfp-in-states*:
  $wfp\ p \Longrightarrow \sigma \in states\ p \Longrightarrow body\ p \rhd \sigma \rightsquigarrow Some\ \sigma' \Longrightarrow \sigma' \in states\ p$
by (*metis exec-wfs-match mem-Collect-eq states-def wfp-def*)

lemma *exec-wfp-always-in-states*:
  $wfp\ p \Longrightarrow \sigma \in states\ p \Longrightarrow \exists \sigma'.\ body\ p \rhd \sigma \rightsquigarrow Some\ \sigma' \wedge \sigma' \in states\ p$
by (*metis exec-always exec-wfp-in-states exec-wfp-no-error option.exhaust*)

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*!]:
  $\sigma \in states\ p \Longrightarrow$
  $body\ p \rhd \sigma \rightsquigarrow Some\ \sigma' \Longrightarrow$
  $(\!|initial = \sigma,\ final = \sigma'|\!) \in traces\ p$

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.[2]

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* ⇒ *bool*
where $\textit{io-vars}\ p \equiv \textit{ctxt}\ p \supseteq \{''\textit{lowIn}'', ''\textit{lowOut}'', ''\textit{highIn}'', ''\textit{highOut}''\}$

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* ⇒ *bool*
where *low-proc-non0 p* ≡
$\forall \sigma \in \textit{states}\ p.\ \forall \sigma'.$
$\quad \textit{the}\ (\sigma\ ''\textit{lowIn}'') \neq \textit{0} \ \wedge$
$\quad \textit{body}\ p \rhd \sigma \rightsquigarrow \textit{Some}\ \sigma' \longrightarrow$
$\quad \textit{the}\ (\sigma'\ ''\textit{lowOut}'') = \textit{the}\ (\sigma\ ''\textit{lowIn}'') + \textit{1}$

[2] As in Section 3.1, 'low' and 'high' have the usual security meaning, e.g. 'low' means 'unclassified' and 'high' means 'classified'.

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* ⇒ *bool*
where *low-proc-0 p* ≡
$\forall\, \sigma \in$ *states p*. $\forall\, u$.
the (σ ″*lowIn*″) = *0* ⟶
($\exists\, \sigma'$. *body p* ⊳ σ ⇝ *Some* σ' ∧ *the* (σ' ″*lowOut*″) = *u*)

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* ⇒ *bool*
where *high-proc p* ≡
$\forall\, \sigma \in$ *states p*. $\forall\, \sigma'$.
body p ⊳ σ ⇝ *Some* σ' ⟶
the (σ' ″*highOut*″) ≥ *the* (σ ″*lowIn*″) + *the* (σ ″*highIn*″)

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 $spec_0$ :: *prog* ⇒ *bool*
where $spec_0$ *p* ≡
wfp p ∧ *io-vars p* ∧ *low-proc-non0 p* ∧ *low-proc-0 p* ∧ *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* ⇒ *nat*
where *low-in* τ ≡ *the* (*initial* τ *''lowIn''*)

definition *low-out* :: *trace* ⇒ *nat*
where *low-out* τ ≡ *the* (*final* τ *''lowOut''*)

definition *high-in* :: *trace* ⇒ *nat*
where *high-in* τ ≡ *the* (*initial* τ *''highIn''*)

definition *high-out* :: *trace* ⇒ *nat*
where *high-out* τ ≡ *the* (*final* τ *''highOut''*)

interpretation
Target: *generalized-non-interference low-in low-out high-in high-out* **.**

abbreviation *GNI* :: *trace set* ⇒ *bool*
where *GNI* ≡ *Target.GNI*

The requirements in $spec_0$ imply that the set of traces of the target program satisfies GNI.

lemma $spec_0$-*GNI*:
$spec_0$ *p* ⟹ *GNI* (*traces p*)
proof (*auto simp*: *Target.GNI-def*)
assume *Spec*: $spec_0$ *p*
— Consider a trace τ_1 and its high input:
fix τ_1::*trace*
def *highIn* ≡ *high-in* τ_1
— Consider a trace τ_2, its low input and output, and its states:
fix τ_2::*trace*
def *lowIn* ≡ *low-in* τ_2
and *lowOut* ≡ *low-out* τ_2
and σ_2 ≡ *initial* τ_2
and σ_2' ≡ *final* τ_2
assume $\tau_2 \in$ *traces p*
hence *Exec2*: *body p* ▷ $\sigma_2 \rightsquigarrow$ *Some* σ_2'
and *State2*: $\sigma_2 \in$ *states p*
by (*auto simp*: σ_2-*def* σ_2'-*def elim*: *traces.cases*)
— Construct the initial state of the witness trace τ_3:
def $\sigma_3 \equiv \sigma_2$ (*''highIn''* ↦ *highIn*)
hence *LowIn3*: *the* (σ_3 *''lowIn''*) = *lowIn*
and *HighIn3*: *the* (σ_3 *''highIn''*) = *highIn*
by (*auto simp*: *lowIn-def low-in-def* σ_2-*def*)

from *Spec State2*
have *State3*: $\sigma_3 \in$ *states p*
by (*auto simp*: σ_3*-def states-def match-def spec*$_0$*-def io-vars-def*)
— Construct the final state of τ_3, and τ_3, by cases on *lowIn*:
show
  $\exists \tau_3 \in$ *traces p*.
    high-in τ_3 = *high-in* τ_1 $\wedge$
    low-in τ_3 = *low-in* τ_2 $\wedge$
    low-out τ_3 = *low-out* τ_2
proof (*cases lowIn*)
  case *0*
  — Use as final state the one required by *low-proc-0*:
  with *Spec State3 LowIn3*
  obtain $\sigma_3{}'$
  where *Exec3*: *body p* $\triangleright$ σ_3 $\rightsquigarrow$ *Some* $\sigma_3{}'$
  and *LowOut3*: *the* ($\sigma_3{}'$ ''*lowOut*'') = *lowOut*
  by (*auto simp*: *spec*$_0$*-def low-proc-0-def*)
  — Construct τ_3 from its initial and final states:
  def $\tau_3 \equiv$ (|*initial* = σ_3, *final* = $\sigma_3{}'$|)
  with *Exec3 State3*
  have *Trace3*: $\tau_3 \in$ *traces p*
  by *auto*
  have *high-in* τ_3 = *high-in* τ_1
  and *low-in* τ_3 = *low-in* τ_2
  and *low-out* τ_3 = *low-out* τ_2
  by (*auto simp*:
    high-in-def low-in-def low-out-def
    τ_3*-def* σ_2*-def* $\sigma_2{}'$*-def*
    highIn-def lowIn-def lowOut-def
    LowIn3 HighIn3 LowOut3)
  with *Trace3*
  show *?thesis*
  by *auto*
next
  case *Suc*
  hence *Not0*: *lowIn* $\neq$ *0*
  by *auto*
  — Derive τ_2's low output from *low-proc-non0*:
  with *Exec2 State2 Spec*
  have *LowOut2*: *lowOut* = *lowIn* + *1*
  by (*auto simp*:
    spec$_0$*-def low-proc-non0-def* σ_2*-def* $\sigma_2{}'$*-def*
    low-in-def low-out-def lowIn-def lowOut-def)
  — Use any final state for τ_3:
  from *Spec*
  have *wfp p*

```
    by (auto simp: spec₀-def)
    with State3
    obtain σ₃'
    where Exec3: body p ▷ σ₃ ⇝ Some σ₃'
    by (metis exec-always exec-wfp-no-error not-Some-eq)
    — Derive τ₃'s low output from low-proc-non0:
    with State3 Spec Not0
    have LowOut3: the (σ₃' ''lowOut'') = lowIn + 1
    by (auto simp: spec₀-def low-proc-non0-def LowIn3)
    — Construct τ₃ from its initial and final states:
    def τ₃ ≡ (|initial = σ₃, final = σ₃'|)
    with Exec3 State3
    have Trace3: τ₃ ∈ traces p
    by auto
    have high-in τ₃ = high-in τ₁
    and low-in τ₃ = low-in τ₂
    and low-out τ₃ = low-out τ₂
    by (auto simp:
      high-in-def low-in-def low-out-def
      τ₃-def σ₃-def σ₂-def
      LowOut2 LowOut3
      highIn-def lowOut-def[unfolded low-out-def, symmetric])
    with Trace3
    show ?thesis
    by auto
  qed
qed
```

Since GNI is implied by $spec_0$ and since every pop-refinement of $spec_0$ implies $spec_0$, GNI is preserved through every pop-refinement of $spec_0$. 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 $spec_0$, 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 $vars_0$ :: *name list*
where $vars_0 \equiv [''lowIn'', ''lowOut'', ''highIn'', ''highOut'']$
— The order of the variables in the list is arbitrary.

lemma $vars_0$-*correct*:
$vars\ p = vars_0 \Longrightarrow io\text{-}vars\ p$
by (*auto simp*: *io-vars-def ctxt-def*)

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

abbreviation Γ_0 :: *ctxt*
where $\Gamma_0 \equiv \{''lowIn'', ''lowOut'', ''highIn'', ''highOut''\}$

lemma *reduce-wf-prog-to-body*:
$vars\ p = vars_0 \Longrightarrow wfp\ p \longleftrightarrow wfs\ \Gamma_0\ (body\ p)$
by (*auto simp*: *wfp-def ctxt-def*)

The refinement of *io-vars* induces a simplification of the processing constraints: since the context of the program is now defined to be Γ_0, the $\sigma \in states\ p$ conditions are replaced with $match\ \sigma\ \Gamma_0$ conditions.

definition $low\text{-}proc\text{-}non0_1$ :: $prog \Rightarrow bool$
where $low\text{-}proc\text{-}non0_1\ p \equiv$
$\forall \sigma\ \sigma'.$
$match\ \sigma\ \Gamma_0\ \wedge$
$the\ (\sigma\ ''lowIn'') \neq 0\ \wedge$
$body\ p \rhd \sigma \rightsquigarrow Some\ \sigma' \longrightarrow$
$the\ (\sigma'\ ''lowOut'') = the\ (\sigma\ ''lowIn'') + 1$

lemma $low\text{-}proc\text{-}non0_1$-*correct*:
$vars\ p = vars_0 \Longrightarrow low\text{-}proc\text{-}non0_1\ p \longleftrightarrow low\text{-}proc\text{-}non0\ p$
by (*auto simp*: $low\text{-}proc\text{-}non0_1$-*def low-proc-non0-def states-def ctxt-def*)

definition $low\text{-}proc\text{-}0_1$ :: $prog \Rightarrow bool$
where $low\text{-}proc\text{-}0_1\ p \equiv$
$\forall \sigma\ u.$
$match\ \sigma\ \Gamma_0\ \wedge$
$the\ (\sigma\ ''lowIn'') = 0 \longrightarrow$
$(\exists \sigma'.\ body\ p \rhd \sigma \rightsquigarrow Some\ \sigma' \wedge the\ (\sigma'\ ''lowOut'') = u)$

lemma $low\text{-}proc\text{-}0_1$-*correct*:
$vars\ p = vars_0 \Longrightarrow low\text{-}proc\text{-}0_1\ p \longleftrightarrow low\text{-}proc\text{-}0\ p$
by (*auto simp*: $low\text{-}proc\text{-}0_1$-*def low-proc-0-def states-def ctxt-def*)

definition $high\text{-}proc_1$ :: $prog \Rightarrow bool$
where $high\text{-}proc_1\ p \equiv$
$\forall \sigma\ \sigma'.$

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

lemma *high-proc*$_1$*-correct*:
$vars\ p = vars_0 \Longrightarrow high\text{-}proc_1\ p \longleftrightarrow high\text{-}proc\ p$
by (*auto simp*: *high-proc*$_1$*-def high-proc-def states-def ctxt-def*)

The refinement of $spec_0$ consists of the refinement of *io-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-1-correct*:
$spec_1\ p \Longrightarrow spec_0\ p$
by (*auto simp*:
spec$_1$*-def spec*$_0$*-def*
vars$_0$*-correct*
reduce-wf-prog-to-body
low-proc-non0$_1$*-correct*
low-proc-0$_1$*-correct*
high-proc$_1$*-correct*)

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-wf-body-to-stmts*:
$body\text{-}split\ p\ s_L\ s_H \Longrightarrow wfs\ \Gamma_0\ (body\ p) \longleftrightarrow wfs\ \Gamma_0\ s_L \wedge wfs\ \Gamma_0\ s_H$
by (*auto simp*: *body-split-def*)

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-proc-non0*$_2$ :: *stmt* ⇒ *bool*
where *low-proc-non0*$_2$ s_L ≡
$\forall \sigma\ \sigma'.$
$\quad$ *match* $\sigma\ \Gamma_0 \wedge$
$\quad$ *the* $(\sigma\ ''lowIn'') \neq 0 \wedge$
$\quad$ $s_L \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \longrightarrow$
$\quad$ *the* $(\sigma'\ ''lowOut'') =$ *the* $(\sigma\ ''lowIn'') + 1$

definition *low-proc-0*$_2$ :: *stmt* ⇒ *bool*
where *low-proc-0*$_2$ s_L ≡
$\forall \sigma\ u.$
$\quad$ *match* $\sigma\ \Gamma_0 \wedge$
$\quad$ *the* $(\sigma\ ''lowIn'') = 0 \longrightarrow$
$\quad$ $(\exists \sigma'.\ s_L \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \wedge$ *the* $(\sigma'\ ''lowOut'') = u)$

definition *low-proc-no-input-change* :: *stmt* ⇒ *bool*
where *low-proc-no-input-change* s_L ≡
$\forall \sigma\ \sigma'.$
$\quad$ *match* $\sigma\ \Gamma_0 \wedge$
$\quad$ $s_L \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \longrightarrow$
$\quad$ *the* $(\sigma'\ ''lowIn'') =$ *the* $(\sigma\ ''lowIn'') \wedge$
$\quad$ *the* $(\sigma'\ ''highIn'') =$ *the* $(\sigma\ ''highIn'')$

definition *high-proc*$_2$ :: *stmt* ⇒ *bool*
where *high-proc*$_2$ s_H ≡
$\forall \sigma\ \sigma'.$
$\quad$ *match* $\sigma\ \Gamma_0 \wedge$
$\quad$ $s_H \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \longrightarrow$
$\quad$ *the* $(\sigma'\ ''highOut'') \geq$ *the* $(\sigma\ ''lowIn'') +$ *the* $(\sigma\ ''highIn'')$

definition *high-proc-no-low-output-change* :: *stmt* ⇒ *bool*
where *high-proc-no-low-output-change* s_H ≡
$\forall \sigma\ \sigma'.$
$\quad$ *match* $\sigma\ \Gamma_0 \wedge$
$\quad$ $s_H \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \longrightarrow$
$\quad$ *the* $(\sigma'\ ''lowOut'') =$ *the* $(\sigma\ ''lowOut'')$

lemma *proc*$_2$*-correct*:
assumes *Body*: *body-split* $p\ s_L\ s_H$
assumes *WfLow*: *wfs* $\Gamma_0\ s_L$
assumes *WfHigh*: *wfs* $\Gamma_0\ s_H$
assumes *LowNon0*: *low-proc-non0*$_2$ s_L

assumes *Low0*: *low-proc-0*$_2$ s_L
assumes *LowSame*: *low-proc-no-input-change* s_L
assumes *High*: *high-proc*$_2$ s_H
assumes *HighSame*: *high-proc-no-low-output-change* s_H
shows *low-proc-non0*$_1$ *p* $\wedge$ *low-proc-0*$_1$ *p* $\wedge$ *high-proc*$_1$ *p*
proof *auto*
— Processing of non-0 low input:
case *goal1*
show *?case*
proof (*auto simp*: *low-proc-non0*$_1$*-def*)
fix σ σ'
assume *body p* $\triangleright$ σ $\rightsquigarrow$ *Some* σ'
with *Body*
obtain σ_i
where *ExecLow*: s_L $\triangleright$ σ $\rightsquigarrow$ *Some* σ_i
and *ExecHigh*: s_H $\triangleright$ σ_i $\rightsquigarrow$ *Some* σ'
by (*auto simp*: *body-split-def elim*: *exec.cases*)
assume *Non0*: *the* (σ *''lowIn''*) > *0*
assume *InitMatch*: *match* σ Γ_0
with *ExecLow WfLow*
have *match* σ_i Γ_0
by (*auto simp*: *exec-wfs-match*)
with *Non0 InitMatch ExecLow ExecHigh HighSame LowNon0*
show *the* (σ' *''lowOut''*) = *Suc* (*the* (σ *''lowIn''*))
unfolding *high-proc-no-low-output-change-def low-proc-non0*$_2$*-def*
by (*metis Suc-eq-plus1 gr-implies-not0*)
qed
next
— Processing of 0 low input:
case *goal2*
show *?case*
proof (*auto simp*: *low-proc-0*$_1$*-def*)
fix σ *u*
assume *InitMatch*: *match* σ Γ_0
and *the* (σ *''lowIn''*) = *0*
with *Low0*
obtain σ_i
where *ExecLow*: s_L $\triangleright$ σ $\rightsquigarrow$ *Some* σ_i
and *LowOut*: *the* (σ_i *''lowOut''*) = *u*
by (*auto simp*: *low-proc-0*$_2$*-def*)
from *InitMatch ExecLow WfLow*
have *MidMatch*: *match* σ_i Γ_0
by (*auto simp*: *exec-wfs-match*)
with *WfHigh*
obtain σ'
where *ExecHigh*: s_H $\triangleright$ σ_i $\rightsquigarrow$ *Some* σ'

    by (*metis exec-wfs-always-match*)
    with *HighSame MidMatch*
    have $the\ (\sigma'\ ''lowOut'') = the\ (\sigma_i\ ''lowOut'')$
    by (*auto simp*: *high-proc-no-low-output-change-def*)
    with *ExecLow ExecHigh Body LowOut*
    show $\exists \sigma'.\ body\ p \rhd \sigma \leadsto Some\ \sigma' \wedge the\ (\sigma'\ ''lowOut'') = u$
    by (*metis* (*hide-lams*, *no-types*) *ExecSeqOK body-split-def*)
  qed
next
  — Processing of high input:
  case *goal3*
  show *?case*
  proof (*auto simp*: *high-proc*$_1$*-def*)
    fix $\sigma\ \sigma'$
    assume $body\ p \rhd \sigma \leadsto Some\ \sigma'$
    with *Body*
    obtain σ_i
    where *ExecLow*: $s_L \rhd \sigma \leadsto Some\ \sigma_i$
    and *ExecHigh*: $s_H \rhd \sigma_i \leadsto Some\ \sigma'$
    by (*auto simp*: *body-split-def elim*: *exec.cases*)
    assume *InitMatch*: $match\ \sigma\ \Gamma_0$
    with *ExecLow WfLow*
    have $match\ \sigma_i\ \Gamma_0$
    by (*auto simp*: *exec-wfs-match*)
    with *InitMatch ExecLow ExecHigh LowSame High*
    show $the\ (\sigma'\ ''highOut'') \geq the\ (\sigma\ ''lowIn'') + the\ (\sigma\ ''highIn'')$
    unfolding *low-proc-no-input-change-def high-proc*$_2$*-def*
    by *metis*
  qed
qed

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

definition $spec_2 :: prog \Rightarrow bool$
where $spec_2\ p \equiv$
  $vars\ p = vars_0 \wedge$
  $(\exists s_L\ s_H.$
    $body\text{-}split\ p\ s_L\ s_H \wedge$
    $wfs\ \Gamma_0\ s_L \wedge$
    $wfs\ \Gamma_0\ s_H \wedge$
    $low\text{-}proc\text{-}non0_2\ s_L \wedge$
    $low\text{-}proc\text{-}0_2\ s_L \wedge$
    $low\text{-}proc\text{-}no\text{-}input\text{-}change\ s_L \wedge$
    $high\text{-}proc_2\ s_H \wedge$
    $high\text{-}proc\text{-}no\text{-}low\text{-}output\text{-}change\ s_H)$

lemma *step-2-correct*:
  $spec_2\ p \Longrightarrow spec_1\ p$
by (*auto simp*: *spec*$_2$*-def spec*$_1$*-def reduce-wf-body-to-stmts proc*$_2$*-correct*)

3.4.3 Step 3

The processing constraints *low-proc-non0*$_2$ and *low-proc-0*$_2$ 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
    (*Var ''lowIn''*)
    (*Const 0*)
    (*Random ''lowOut''*)
    (*Assign ''lowOut''* (*Add* (*Var ''lowIn''*) (*Const 1*)))

lemma *wfs-*s_{L0}:
  wfs Γ_0 s_{L0}
by *auto*

lemma *low-proc-non0-*s_{L0}:
  low-proc-non0$_2$ s_{L0}
proof (*auto simp only*: *low-proc-non0*$_2$*-def*)
  fix $\sigma\ \sigma'$
  assume *Match*: *match* $\sigma\ \Gamma_0$
  assume $s_{L0} \rhd \sigma \rightsquigarrow$ *Some* σ'
  and *the* (σ *''lowIn''*) $>$ *0*
  hence (*Assign ''lowOut''* (*Add* (*Var ''lowIn''*) (*Const 1*))) $\rhd \sigma \rightsquigarrow$ *Some* σ'
  by (*auto elim*: *exec.cases*)
  hence $\sigma' = \sigma$ (*''lowOut''* $\mapsto$ *the* (*eval* σ (*Add* (*Var ''lowIn''*) (*Const 1*))))
  by (*auto elim*: *exec.cases*)
  with *Match*
  show *the* (σ' *''lowOut''*) $=$ *the* (σ *''lowIn''*) $+$ *1*
  by (*auto simp*: *match-def add-opt-def split*: *option.split*)
qed

lemma *low-proc-0-*s_{L0}:
  low-proc-0$_2$ s_{L0}
proof (*auto simp only*: *low-proc-0*$_2$*-def*)
  fix σ *u*
  assume *Match*: *match* $\sigma\ \Gamma_0$
  and *the* (σ *''lowIn''*) $=$ *0*
  hence *LowIn0*: σ *''lowIn''* $=$ *Some 0*
  by (*cases* σ *''lowIn''*, *auto simp*: *match-def*)

from *Match*
have $''lowOut'' \in dom\ \sigma$
by (*auto simp*: *match-def*)
then obtain σ'
where *ExecRand*: $Random\ ''lowOut'' \rhd \sigma \leadsto Some\ \sigma'$
and $\sigma' = \sigma\ (''lowOut'' \mapsto u)$
by (*auto intro*: *ExecRandomOK*)
hence $the\ (\sigma'\ ''lowOut'') = u$
by *auto*
with *ExecRand LowIn0*
show $\exists \sigma'.\ s_{L0} \rhd \sigma \leadsto Some\ \sigma' \wedge the\ (\sigma'\ ''lowOut'') = u$
by (*metis ExecCondTrue eval.simps*(*1*) *eval.simps*(*2*))
qed

lemma *low-proc-no-input-change-*s_{L0}:
low-proc-no-input-change s_{L0}
proof (*unfold low-proc-no-input-change-def*, *clarify*)
fix $\sigma\ \sigma'$
assume $s_{L0} \rhd \sigma \leadsto Some\ \sigma'$
hence
$Random\ ''lowOut'' \rhd \sigma \leadsto Some\ \sigma' \vee$
$Assign\ ''lowOut''\ (Add\ (Var\ ''lowIn'')\ (Const\ 1)) \rhd \sigma \leadsto Some\ \sigma'$
by (*auto elim*: *exec.cases*)
thus
$the\ (\sigma'\ ''lowIn'') = the\ (\sigma\ ''lowIn'') \wedge$
$the\ (\sigma'\ ''highIn'') = the\ (\sigma\ ''highIn'')$
by (*auto elim*: *exec.cases*)
qed

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

definition $spec_3 :: prog \Rightarrow bool$
where $spec_3\ p \equiv$
$vars\ p = vars_0 \wedge$
$(\exists s_H.$
$body\text{-}split\ p\ s_{L0}\ s_H \wedge$
$wfs\ \Gamma_0\ s_H \wedge$
$high\text{-}proc_2\ s_H \wedge$
$high\text{-}proc\text{-}no\text{-}low\text{-}output\text{-}change\ s_H)$

lemma *step-3-correct*:
$spec_3\ p \Longrightarrow spec_2\ p$
unfolding $spec_3$*-def* $spec_2$*-def*
by (*metis*
wfs-s_{L0} *low-proc-non0-*s_{L0} *low-proc-0-*s_{L0} *low-proc-no-input-change-*s_{L0})

The non-determinism required by *low-proc-0* cannot be pop-refined away. In particular, s_L 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*$_2$ on s_H can be satisfied in different ways. A simple way is to pick the sum of the low and high inputs: *high-proc*$_2$ is refined by replacing the inequality with an equality.

definition *high-proc*$_4$ :: *stmt* $\Rightarrow$ *bool*
where *high-proc*$_4$ s_H $\equiv$
$\forall\, \sigma\ \sigma'.$
  match σ Γ_0 $\wedge$
  $s_H \rhd \sigma \rightsquigarrow$ *Some* $\sigma' \longrightarrow$
  the $(\sigma'$ *''highOut''*$)$ = *the* $(\sigma$ *''lowIn''*$)$ + *the* $(\sigma$ *''highIn''*$)$

lemma *high-proc*$_4$*-correct*:
  high-proc$_4$ $s_H \Longrightarrow$ *high-proc*$_2$ s_H
by (*auto simp*: *high-proc*$_4$*-def high-proc*$_2$*-def*)

The refined specification is obtained by substituting the refined processing constraint on s_H.

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 Γ_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* $\Longrightarrow$ *spec*$_3$ *p*
by (*auto simp*: *spec*$_4$*-def spec*$_3$*-def high-proc*$_4$*-correct*)

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_{H0}:
  wfs Γ_0 s_{H0}

by *auto*

lemma *high-proc*$_4$-s_{H0}:
high-proc$_4$ s_{H0}
proof (*auto simp*: *high-proc*$_4$-*def*)
fix σ σ'
assume *Match*: *match* σ Γ_0
assume $s_{H0} \rhd \sigma \rightsquigarrow$ *Some* σ'
hence
$\sigma' = \sigma$ (''*highOut*'' $\mapsto$ *the* (*eval* σ (*Add* (*Var* ''*lowIn*'') (*Var* ''*highIn*''))))
by (*auto elim*: *exec.cases*)
with *Match*
show *the* (σ' ''*highOut*'') = *the* (σ ''*lowIn*'') + *the* (σ ''*highIn*'')
by (*auto simp*: *match-def add-opt-def split*: *option.split*)
qed

lemma *high-proc-no-low-output-change*-s_{H0}:
high-proc-no-low-output-change s_{H0}
by (*auto simp*: *high-proc-no-low-output-change-def elim*: *exec.cases*)

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* $\Longrightarrow$ *spec*$_4$ *p*
unfolding *spec*$_5$-*def* *spec*$_4$-*def*
by (*metis wfs*-s_{H0} *high-proc*$_4$-s_{H0} *high-proc-no-low-output-change*-s_{H0})

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 $p_0 \equiv$ (|*vars* = *vars*$_0$, *body* = *Seq* s_{L0} s_{H0}|)

definition *spec*$_6$ :: *prog* $\Rightarrow$ *bool*
where *spec*$_6$ *p* $\equiv$ *p* = p_0

lemma *step-6-correct*:
spec$_6$ *p* $\Longrightarrow$ *spec*$_5$ *p*
by (*auto simp*: *spec*$_6$-*def* *spec*$_5$-*def* *body-split-def*)

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
by (*metis*
step-1-correct
step-2-correct
step-3-correct
step-4-correct
step-5-correct
step-6-correct
spec$_6$-*def*)

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 $\mathit{wfp}\ p$ in Section 2.2 and Section 3.3, $\mathit{para}\ p = [''x'', ''y'']$ in Section 2.2, and $\mathit{iovars}\ p$ 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.[1] A simple example is $\mathit{costp}\ p \leq \mathit{3}$ in Section 2.2.

[1]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, $\forall\, x\ y.\ exec\ p\ [x,\ y] = Some\ (f\ x\ y)$ in $spec_0$ in Section 2.2 links the high-level functional requirement expressed by f to the target program p.[2]

As another example, a function $sort :: nat\ list \Rightarrow nat\ list$, 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

[2]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.[3]

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. $\mathit{spec}_{i+1} = \mathit{spec}_i$. 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 $\mathit{spec}_{i+1}\ p \Longrightarrow \mathit{spec}_i\ p$ 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 $\mathit{spec}_{i+1} = \lambda p.\ \mathit{False}$, 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 $\mathit{spec}_n\ p \equiv p = p_0$, where p_0 is the representation of a program's abstract syntax

[3]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 Φ 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).[4]

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.

[4]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 $spec_{i+1}\ p \Longrightarrow spec_i\ p$ 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 $spec_{i+1}\ p \Longrightarrow spec_i\ p$. It is sometimes possible to generate $spec_{i+1}$ from $spec_i$, along with a proof of $spec_{i+1}\ p \Longrightarrow spec_i\ p$, 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://afp.sf.net/entries/TLA.shtml, Formal proof development.

[7] Steffen Helke and Florian Kammüller. Formalizing Statecharts using hierarchical automata. *Archive of Formal Proofs*, 2010. http://afp.sf.net/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.