# On a New Notion of Partial Refinement

Emil Sekerinski
McMaster University
Hamilton, Canada
emil@mcmaster.ca

Tian Zhang
McMaster University
Hamilton, Canada
zhangt26@mcmaster.ca

Formal specification techniques allow expressing idealized specifications, which abstract from restrictions that may arise in implementations. However, partial implementations are universal in software development due to practical limitations. Our goal is to contribute to a method of program refinement that allows for partial implementations. For programs with a normal and an exceptional exit, we propose a new notion of partial refinement which allows an implementation to terminate exceptionally if the desired results cannot be achieved, provided the initial state is maintained. Partial refinement leads to a systematic method of developing programs with exception handling.

## 1 Introduction

In software development, specifications are meant to be concise by stating abstractly only the intention of a program rather than elaborating on a possible implementation. However, practical restrictions can prevent idealized specifications from being fully implemented. In general, there are three sources of *partiality* in implementations: there may be inherent limitations of the implementation, some features may intentionally not (yet) be implemented, or there may be a genuine fault.

As an example of inherent limitations of an implementation, consider a class for the analysis of a collection of integers. The operations are initialization, inserting an integer, and summing all its elements. Assume that int is a type for machine-representable integers, bounded by *MIN* and *MAX*, and machine arithmetic is bounded, i.e. an overflow caused by arithmetic operations on int is detected and raises an exception, as available in x86 assembly language [11] and .NET [20]. We define:

```
class IntCollection
  bag(int) b
  invariant inv : ∀ x ∈ b · MIN ≤ x ≤ MAX
  method init()
    b := []
  method insert(n : int)
    b := b + [n]
  method sum() : int
    result := ∑ b
```

This specification allows an unbounded number of machine-representable integers to be stored in the abstract bag $b$, which is an unordered collection that, unlike a set, allows duplication of elements, The empty bag is written as $[]$, the bag consisting only of a single $n$ as $[n]$, union of bags $b, b'$ as $b + b'$, and the sum of all elements of bag $b$ as $\sum b$. (A model of bags is a function from elements to their number of occurrences in the bag.) However, in an implementation, method *init* (object initialization) and *insert* may fail due to memory exhaustion at heap allocation (and raise an exception), and method *sum* may fail due to overflow of the result (and raise an exception). Even if the result of *sum* is machine-representable, the iterative computation of the sum in a particular order may still overflow. Hence no

John Derrick, Eerke Boiten, Steve Reeves (Eds.):
Refinement Workshop 2013
EPTCS 115, 2013, pp. 1–14, doi:10.4204/EPTCS.115.1

realistic implementation of this class can be faithful. Still, it is a useful specification because of its clarity and brevity. Obviously, using mathematical integers instead of machine-representable integers in the specification would make implementations even "more partial".

The second source of partiality is intentionally missing features. The evolutionary development of software consists of anticipating and performing *extensions* and *contractions* [22]. Anticipated extensions can be expressed by features that are present but not yet implemented. Contractions lead to obsolete features that will eventually be removed. In both cases, implementations of features may be missing. A common practice is to raise an exception if a feature is not yet implemented. Here is a recommendation from the Microsoft Developer Documentation for .NET [19]:

```
static void FutureFeature()
{
    // Not developed yet.
    throw new NotImplementedException();
}
```

The third source of partiality is genuine faults. These may arise from the use of software layers that are themselves faulty (operating system, compilers, libraries), from faults in the hardware (transient or permanent), or from errors in the design (errors in the correctness argument, incorrect hypothesis about the abstract machine).

Our goal is to contribute to a method of program refinement that allows for partial implementations that guarantee "safe" failure if the desired outcome cannot be computed. To this end, we consider program statements with a one entry and two exits, a normal and an exceptional exit. Specifications are considered to be abstract programs and implementations are considered to be concrete programs. For program statements $S$ (the specification) and $T$ (the implementation), the total refinement of $S$ by $T$ means that either $T$ terminates normally and establishes a postcondition that $S$ may also establish on normal termination, or $T$ terminates exceptionally and establishes a postcondition that $S$ may also establish on exceptional termination. As a relaxation, *partial refinement* allows $T$ additionally to terminate exceptionally provided the initial state is preserved. The intention is that the implementation $T$ tries to meet specification $S$, but if it cannot do so, $T$ can fail safely by not changing the state and terminating exceptionally. When applying partial refinement to data refinement, an implementation that cannot meet the specification and fails may still change the state as long as the change is not visible in the specification.

The exception handling of the Eiffel programming language provided the inspiration for partial refinement [18]: in Eiffel, each method has one entry and two exits, a normal and an exceptional exit, but is specified by a single precondition and single postcondition only. The normal exit is taken if the desired postcondition is established and the exceptional exit is taken if the desired postcondition cannot be established, thus allowing partial implementations. Other approaches to exceptions rely on specifying one postcondition for normal exit and one for each exceptional exit (in case there is more than one exceptional exit). Compared to that, the Eiffel approach is simpler in requiring a single postcondition and more general in that an valid outcome is always possible, even in presence of unanticipated failure. Compared to Eiffel, our notion of partial refinement imposes the additional constraint that in case of exceptional exit, the (abstract) state must be preserved. In Eiffel, the postcondition must be evaluated at run-time to determine if the normal or exceptional exit it taken (and hence must be efficiently computable). Partial refinement is more general in the sense that it does not require that a postcondition to be evaluated at run-time, as long as faults are detected at run-time in some way. In our earlier work we introduced a new notion of *partial correctness* [23], which also models the programs that can fail safely. (This notion of partial correctness is different from relaxing total correctness by not guaranteeing termination [13].)

Partial refinement is related to this notion of partial correctness in the same sense as total refinement is related to total correctness.

Retrenchment also addresses the issue of partial implementations, but with statements with one entry and one exit [4]: the refinement of each operation of a data type requires a *within* and a *concedes* relation that restrict the initial states and widen the possible final states of an implementation. Compared to retrenchment, partial refinement does not require additional relations to be specified. Partial refinement is more restrictive in the sense that initial states cannot be restricted and the final state cannot be widened on normal termination. In partial refinement, the caller is notified through an exception of the failure; retrenchment is a design technique that is independent of exception handling.

The term partial refinement has been introduced with a different meaning in [3] for statements with one exit. There, partial refinement allows the domain of termination to be reduced. In the present work, partial refinement requires termination, either on the normal or exceptional exit. Partial refinement in [14] refers to transition system refinement with a partial refinement relation; the approach is to construct fault-tolerant systems in two phases, first with an idealized specification and then adding fault-tolerant behaviour. Here we use partial refinement specifically for statements with two exits.

We define statements by predicate transformers, as they are expressive in that they can describe blocking, abortion, demonic nondeterminism, and angelic nondeterminism, see e.g. Back and von Wright [2] for a comprehensive treatment; that line of work considers statements with one entry and one exit. In the approach to exception handling by Cristian [6] statements have one entry and multiple exits (one of those being the normal one) and are defined by a set of predicate transformers, one for each exit. As pointed out by King and Morgan [15], this disallows nondeterminism, which precludes the use of the language for specification and design; their solution is to use a single predicate transformer with one postcondition for each exit instead and define total refinement as a generalization of refinement of statements with one exit. (For example, $\mathsf{skip} \sqcap \mathsf{raise}$ terminates, but is neither guaranteed to terminate normally nor exceptionally; hence its termination cannot be captured by a predicate transformer for each exit.) Leino and Snepscheut justify a very similar definition of statements by predicate transformers with two postconditions in terms of a trace semantics, but do not consider refinement [16]. Here, we identify a statement $S$ with its (higher-order) predicate transformer following [2], instead of introducing a function $wp(S)$ in Dijkstra's style [8]. Watson uses total refinement to further study the refinement in the style of King and Morgan [25]. With total refinement, an implementation has to meet its specifications, but may use raising and catching exceptions as an additional control structure. Our definition of partial refinement on statements with two exits is in terms of total refinement, but allows implementations to fail safely. Compared to earlier work on refinement with exceptions, we consider undefinedness in expressions and give corresponding (more involved) verification rules. King and Morgan trace the use of predicate transformers with multiple postconditions to Back and Karttunen [1].

**Overview.** The next two sections give the formal definitions of statements with two exits using predicate transformers. Section 4 and 5 introduce the notions of total correctness and partial correctness for statements with two exits. Section 6 gives verification rules for loops. In Section 7 we give the formal definition of partial refinement, and show its applications through examples. Section 8 concludes our contribution.

All the theorems and examples have been checked with the Isabelle proof assistant[1], so we allow ourselves to omit the proofs.

[1]The Isabelle proof files are available at `http://www.cas.mcmaster.ca/~zhangt26/REFINE/`.

## 2 Statements with Two Exits

We begin by formally defining statements with a one entry and two exits, a *normal* and an *exceptional* exit. A statement either *succeeds* (terminates normally), *fails* (terminates exceptionally), *aborts* (is out of control and may not terminate at all), or *blocks* (refuses to execute). Following [2], a statement $S$ is identified with its predicate transformer, but taking two postconditions as arguments: $S(q,r)$ is the weakest precondition such that either $S$ succeeds with postcondition $q$ or $S$ fails with postcondition $r$.

A *state predicate* of type $\mathscr{P}\Sigma$ is a function from state space $\Sigma$ to *Bool*, i.e. $\mathscr{P}\Sigma = \Sigma \to Bool$. A *relation* of type $\Delta \leftrightarrow \Sigma$ is a function from the initial state space $\Delta$ to a state predicate over the final state space $\Sigma$, i.e. $\Delta \leftrightarrow \Sigma = \Delta \to \mathscr{P}\Sigma$; we allow the initial and final state spaces to be different. It is isomorphic to $\mathscr{P}(\Delta \times \Sigma)$, but allows the test if $(a,b)$ is in relation $r$ to be simply written as $r\,a\,b$.

A *predicate transformer* is a function from a normal postcondition of type $\mathscr{P}\Psi$ and an exceptional postcondition of type $\mathscr{P}\Omega$, to a precondition of type $\mathscr{P}\Delta$, i.e. of type $\mathscr{P}\Psi \times \mathscr{P}\Omega \to \mathscr{P}\Delta$, for types $\Psi, \Omega, \Delta$. In the rest of the paper, we leave the types out if they can be inferred from the context.

On state predicates, conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, consequence $\Leftarrow$, and negation $\neg$ are defined by the pointwise extension of the corresponding operations on *Bool*, e.g. $(p \wedge q)\sigma \mathrel{\widehat{=}} p\,\sigma \wedge q\,\sigma$. The entailment ordering $\leq$ is defined by universal implication, $p \leq q \mathrel{\widehat{=}} \forall \sigma \cdot p\,\sigma \Rightarrow q\,\sigma$. The state predicates $\mathsf{true}$ and $\mathsf{false}$ represent the universally true respectively false predicates. Predicate transformer $S$ is *monotonic* if $q \leq q' \wedge r \leq r' \Rightarrow S(q,r) \leq S(q',r')$. A *statement* is a monotonic predicate transformer.

We define some basic predicate transformers: $\mathsf{abort}$ is completely unpredictable and may terminate normally or exceptionally in any state or may not terminate at all; $\mathsf{stop}$, also known as $\mathsf{magic}$, miraculously guarantees any postcondition by blocking execution; $\mathsf{skip}$ changes nothing and succeeds whereas $\mathsf{raise}$ changes nothing and fails. The *sequential composition* $S\,;T$ continues with $T$ only if $S$ succeeds whereas the *exceptional composition* $S\,;;T$ continues with $T$ only if $S$ fails. The notation suggests that ; is sequential composition on the first exit and ;; is sequential composition on the second exit. The *demonic choice* $S \sqcap T$ establishes a postcondition only if both $S$ and $T$ do. The *angelic choice* $S \sqcup T$ establishes a postcondition if either $S$ or $T$ does:

$$
\begin{array}{lcllcl}
\mathsf{abort}\,(q,r) & \mathrel{\widehat{=}} & \mathsf{false} & (S\,;T)\,(q,r) & \mathrel{\widehat{=}} & S\,(T\,(q,r),r) \\
\mathsf{stop}\,(q,r) & \mathrel{\widehat{=}} & \mathsf{true} & (S\,;;T)\,(q,r) & \mathrel{\widehat{=}} & S\,(q,T\,(q,r)) \\
\mathsf{skip}\,(q,r) & \mathrel{\widehat{=}} & q & (S \sqcap T)\,(q,r) & \mathrel{\widehat{=}} & S\,(q,r) \wedge T\,(q,r) \\
\mathsf{raise}\,(q,r) & \mathrel{\widehat{=}} & r & (S \sqcup T)\,(q,r) & \mathrel{\widehat{=}} & S\,(q,r) \vee T\,(q,r)
\end{array}
$$

Predicate transformers $\mathsf{abort}$, $\mathsf{stop}$, $\mathsf{skip}$, and $\mathsf{abort}$ are monotonic and hence statements. Operators ;, ;;, $\sqcap$, $\sqcup$ preserve monotonicity. Sequential composition is associative and has $\mathsf{skip}$ as unit, giving rise to a monoid structure. Dually, exceptional composition is associative and has $\mathsf{raise}$ as unit, giving rise to another monoid structure. Further properties of functions with two arguments, with application to semantics of exceptions, are studied in [16, 17].

The *total refinement ordering* $\sqsubseteq$ on predicate transformers with two arguments is defined by universal entailment [15]:

$$S \sqsubseteq T \quad \mathrel{\widehat{=}} \quad \forall q,r \cdot S\,(q,r) \leq T\,(q,r)$$

Both sequential and exceptional composition are monotonic in both arguments with respect to the total refinement ordering. Statements with the total refinement ordering form a complete lattice, with $\mathsf{abort}$ as the bottom element, $\mathsf{stop}$ as the top element, $\sqcap$ as the meet operator, and $\sqcup$ as the join operator. These properties are similar to those of predicate transformers with one argument [2], except that here we have two monoid structures.

We introduce statements for inspecting and modifying the state. Let $u, v$ be state predicates. For predicate transformers with one argument, the assumption $[u]$ does nothing if $u$ holds and blocks otherwise, and the assertions $\{u\}$ does nothing if $u$ hold and aborts otherwise. For predicate transformers with two arguments, the *assumption* $[u,v]$ succeeds if $u$ holds, fails if $v$ holds, chooses demonically between these two possibilities if both $u$ and $v$ hold, and blocks if neither $u$ nor $v$ hold. The *assertion* $\{u,v\}$ succeeds if $u$ holds, fails if $v$ holds, choosing angelically between these two possibilities if both $u$ and $v$ hold, and aborts if neither $u$ nor $v$ holds. Neither assumption nor assertion change the state if they succeed or fail.

$$[u,v]\,(q,r) \quad \widehat{=} \quad (u \Rightarrow q) \wedge (v \Rightarrow r) \qquad\qquad \{u,v\}\,(q,r) \quad \widehat{=} \quad (u \wedge q) \vee (v \wedge r)$$

Both assumption and assertion are monotonic and hence statements. We have that $[\mathsf{true},\mathsf{false}] = \mathsf{skip} = \{\mathsf{true},\mathsf{false}\}$ and that $[\mathsf{false},\mathsf{true}] = \mathsf{raise} = \{\mathsf{false},\mathsf{true}\}$. We also have that $[\mathsf{false},\mathsf{false}] = \mathsf{stop}$ and that $\{\mathsf{false},\mathsf{false}\} = \mathsf{abort}$. Finally we have that $[\mathsf{true},\mathsf{true}] = \mathsf{skip} \sqcap \mathsf{raise}$ and that $\{\mathsf{true},\mathsf{true}\} = \mathsf{skip} \sqcup \mathsf{raise}$.

The *demonic update* $[Q,R]$ and the *angelic update* $\{Q,R\}$ both update the state according to relation $Q$ and succeed, or update the state according to relation $R$ and fail, the difference being that both the choice offered by the relations and the choice between succeeding and failing are demonic with $[Q,R]$ and are angelic with $\{Q,R\}$. If $Q$ is of type $\Delta \leftrightarrow \Psi$ and $R$ is of type $\Delta \leftrightarrow \Omega$, then $[Q,R]$ and $\{Q,R\}$ are of type $\mathscr{P}\Psi \times \mathscr{P}\Omega \to \mathscr{P}\Delta$:

$$\begin{array}{lcl} [Q,R]\,(q,r)\delta & \widehat{=} & (\forall\, \psi \cdot Q\,\delta\,\psi \Rightarrow q\,\psi) \wedge (\forall\, \omega \cdot R\,\delta\,\omega \Rightarrow r\,\omega) \\ \{Q,R\}\,(q,r)\delta & \widehat{=} & (\exists\, \psi \cdot R\,\delta\,\psi \wedge q\,\psi) \vee (\exists\, \omega \cdot R\,\delta\,\omega \wedge r\,\omega) \end{array}$$

Both demonic update and angelic update are monotonic in both arguments and hence statements. Writing $\bot$ for the empty relation and $\mathsf{id}$ for the identity relation we have that $[\mathsf{id},\bot] = \mathsf{skip} = \{\mathsf{id},\bot\}$ and that $[\bot,\mathsf{id}] = \mathsf{raise} = \{\bot,\mathsf{id}\}$. We also have that $[\bot,\bot] = \mathsf{stop}$ and that $\{\bot,\bot\} = \mathsf{abort}$. Finally we have that $[\mathsf{id},\mathsf{id}] = \mathsf{skip} \sqcap \mathsf{raise}$ and that $\{\mathsf{id},\mathsf{id}\} = \mathsf{skip} \sqcup \mathsf{raise}$. Writing $\top$ for the universal relation, both updates $[\top,\top]$ and $\{\top,\top\}$ terminate, with $[\top,\top]$ making a demonic choice between succeeding and failing, and a demonic choice among the final states, and $\{\top,\top\}$ making these choices angelic.

## 3 Derived Statements

To establish the connection to predicate transformers with one argument we define:

$$\begin{array}{lcllcl} [u]\,(q,r) & \widehat{=} & u \Rightarrow q & [Q]\,(q,r)\,\delta & \widehat{=} & (\forall\, \psi \cdot Q\ \delta\ \psi \Rightarrow q\ \psi) \\ \{u\}\,(q,r) & \widehat{=} & u \wedge q & \{Q\}\,(q,r)\,\delta & \widehat{=} & (\exists\, \psi \cdot Q\ \delta\ \psi \wedge q\ \psi) \end{array}$$

These definitions are identical to predicate transformers with one argument, except for the additional parameter $r$ [2]. We have that $[u] = [u,\mathsf{false}]$ and $\{u\} = \{u,\mathsf{false}\}$ as well as $[Q] = [Q,\bot]$ and $\{Q\} = \{Q,\bot\}$.

The common $\mathsf{try}\,S\,\mathsf{catch}\,T$ statement with *body* $S$ and *handler* $T$ starts with $S$, if $S$ succeeds, the whole statement succeeds, if $S$ fails, execution continues with $T$. The try-catch statement is directly defined by exceptional composition. (We have introduced the operator ;; to stress the duality to ; in the algebraic structure; partial refinement will break this duality.) The statement $\mathsf{try}\,S\,\mathsf{catch}\,T\,\mathsf{finally}\,U$ with *finalization* $U$ is defined in terms of sequential and exceptional composition: $U$ is executed either after $S$ succeeds, after $S$ fails and $T$ succeeds, or after $S$ fails and $T$ fails, in which case the whole statement fails whether $U$ succeeds or fails:

$$\begin{array}{lcl} \mathsf{try}\,S\,\mathsf{catch}\,T & \widehat{=} & S\,;;\,T \\ \mathsf{try}\,S\,\mathsf{catch}\,T\,\mathsf{finally}\,U & \widehat{=} & (S\,;;\,(T\,;;\,(U\,;\mathsf{raise})))\,;\,U \end{array}$$

The *assignment* $x := E$ is defined in terms of an update statement that affects only component $x$ of the state space. For this we assume that the state is a tuple and variables select elements of the tuple. Here $E$ may be partially defined, as for example in $x := x \operatorname{div} y$. A division by zero should lead to failure without a state change, otherwise to success with $x$ updated. A *program expression* $E$ is a term for which *definedness* $\mathsf{def}\, E$ (the domain of $E$) and *value* $\mathsf{val}\, E$ are given; the result of $\mathsf{def}\, E$ and $\mathsf{val}\, E$ are expressions of the underlying logic of pre- and postconditions. For example, if the state space consists of variables $x, y$, we have $\mathsf{def}\text{“}x \operatorname{div} y\text{”} \equiv (\lambda\, x, y \cdot y \neq 0)$ and $\mathsf{val}\text{“}x \operatorname{div} y\text{”} = (\lambda\, x, y \cdot \mathsf{if}\ y \neq 0\ \mathsf{then}\ x\ \mathsf{div}\ y\ \mathsf{else}\ \mathit{NaN})$. We use upper case names $B, E, ES$ for program expressions (partial functions) and lower case names $e, es, p, q, r, \ldots$ for terms of higher-order logic (total functions). The *relational update* $x := e$ modifies the $x$ component of the state space to be $e\,(x, y)$ and leaves all other components of the state space unchanged; the initial and final state space are the same. The *nondeterministic relational update* $x :\in es$ modifies the $x$ component of the state space to be any element of the set $es\,(x, y)$. Provided that the state space consists of variables $x, y$ we define:

$$\begin{array}{lcl} x := e & \widehat{=} & \lambda\, x, y \cdot \lambda\, x', y' \cdot x' = e\,(x, y) \wedge y' = y \\ x :\in es & \widehat{=} & \lambda\, x, y \cdot \lambda\, x', y' \cdot x' \in es\,(x, y) \wedge y' = y \end{array}$$

The (deterministic) *assignment* $x := E$ fails if program expression $E$ is not defined, otherwise it succeeds and assigns the value of $E\,(x, y)$ to $x$. The *nondeterministic assignment* $x :\in ES$ fails if the program expression $ES$ is not defined, otherwise it succeeds and assigns any element of the set $ES\,(x, y)$ to $x$, the choice being demonic:

$$\begin{array}{lcl} x := E & \widehat{=} & [\mathsf{def}\, E, \neg\, \mathsf{def}\, E]\,;[x := \mathsf{val}\, E] \\ x :\in ES & \widehat{=} & [\mathsf{def}\, ES, \neg\, \mathsf{def}\, ES]\,;[x :\in \mathsf{val}\, ES] \end{array}$$

The *conditional* $\mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T$ fails if Boolean program expression $B$ is not defined, otherwise continues with either $S$ or $T$, depending on the value of $B$:

$$\mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T \quad \widehat{=} \quad [\mathsf{def}\, B, \neg\, \mathsf{def}\, B]\,;((\,[\mathsf{val}\, B]\,;S) \sqcap (\,[\neg\, \mathsf{val}\, B]\,;T))$$

The *loop* $\mathsf{while}\, B\, \mathsf{do}\, S$ is defined as the least fixed point of $F = (\lambda\, X \cdot \mathsf{if}\, B\, \mathsf{then}(S\,;X))$ with respect to the total refinement ordering. According to the Theorem of Knaster-Tarski, any monotonic function $f$ over a complete lattice has a unique least fixed point, written $\mu f$. Statements form a complete lattice, sequential composition and the conditional are monotonic, hence we can define for Boolean program expression $B$ and predicate transformer $S$:

$$\mathsf{while}\, B\, \mathsf{do}\, S \mathrel{\widehat{=}} \mu\, (\lambda\, X \cdot \mathsf{if}\, B\, \mathsf{then}\, S\,;X\, \mathsf{else}\, \mathsf{skip})$$

Loops preserve monotonicity: if $S$ is monotonic, then $\mathsf{while}\, B\, \mathsf{do}\, S$ is also monotonic.

# 4 Total Correctness

The total correctness assertion $[\![p]\!]\, S\, [\![q]\!]$ states that under precondition $p$, statement $S$ terminates with postcondition $q$. Correctness assertions give a better intuition than predicate transformers with entailment. Total correctness assertions are generalized to two postconditions, the normal and exceptional postcondition:

$$\begin{array}{lcl} [\![p]\!]\, S\, [\![q, r]\!] & \equiv & \text{Under precondition } p\text{, statement } S \text{ terminates and} \\ & & \text{– on normal termination } q \text{ holds finally,} \\ & & \text{– on exceptional termination } r \text{ holds finally.} \end{array}$$

We say that under $p$, statement $S$ succeeds with $q$ and fails with $r$. If $[\![p]\!]\, S\, [\![q, \mathsf{false}]\!]$ holds, then $S$ never fails, and we write this more concisely as $[\![p]\!]\, S\, [\![q]\!]$. Let $p, q, r$ be state predicates:

$$\begin{array}{lcl} [\![p]\!]\, S\, [\![q,r]\!] & \widehat{=} & p \le S(q,r) \\ [\![p]\!]\, S\, [\![q]\!] & \widehat{=} & p \le S(q, \mathsf{false}) \end{array}$$

The next theorem summarizes the basic rules for total correctness, considering possible undefinedness of program expressions, see also [6, 12, 15, 16]:

**Theorem 1** *Let $p, q, r$ be state predicates, $B$ be a Boolean expression, and $S, T$ be statements:*

$$\begin{array}{lcl}
[\![p]\!]\ \mathsf{abort}\ [\![q,r]\!] & \equiv & p = \mathsf{false} \\
[\![p]\!]\ \mathsf{stop}\ [\![q,r]\!] & \equiv & \mathsf{true} \\
[\![p]\!]\ \mathsf{skip}\ [\![q,r]\!] & \equiv & p \Rightarrow q \\
[\![p]\!]\ \mathsf{raise}\ [\![q,r]\!] & \equiv & p \Rightarrow r \\
[\![p]\!]\ x := E\ [\![q,r]\!] & \equiv & (\mathsf{def}\, E \wedge p \Rightarrow q[x \backslash \mathsf{val}\, E]) \wedge \\
 & & (\neg\, \mathsf{def}\, E \wedge p \Rightarrow r) \\
[\![p]\!]\ x :\in ES\ [\![q,r]\!] & \equiv & (\mathsf{def}\, ES \wedge p \Rightarrow \forall x' \in \mathsf{val}\, ES \cdot q[x \backslash x']) \wedge \\
 & & (\neg\, \mathsf{def}\, ES \wedge p \Rightarrow r) \\
[\![p]\!]\ S\,;T\ [\![q,r]\!] & \equiv & \exists h \cdot [\![p]\!]\, S\, [\![h,r]\!] \wedge \\
 & & \quad [\![h]\!]\, T\, [\![q,r]\!] \\
[\![p]\!]\ \mathsf{try}\, S\, \mathsf{catch}\, T\ [\![q,r]\!] & \equiv & \exists h \cdot [\![p]\!]\, S\, [\![q,h]\!] \wedge \\
 & & \quad [\![h]\!]\, T\, [\![q,r]\!] \\
[\![p]\!]\ S \sqcap T\ [\![q,r]\!] & \equiv & [\![p]\!]\, S\, [\![q,r]\!] \wedge \\
 & & [\![p]\!]\, T\, [\![q,r]\!] \\
[\![p]\!]\ \mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T\ [\![q,r]\!] & \equiv & [\![\mathsf{def}\, B \wedge \mathsf{val}\, B \wedge p]\!]\, S\, [\![q,r]\!] \wedge \\
 & & [\![\mathsf{def}\, B \wedge \neg\, \mathsf{val}\, B \wedge p]\!]\, T\, [\![q,r]\!] \wedge \\
 & & (\neg\, \mathsf{def}\, B \wedge p \Rightarrow r)
\end{array}$$

Total correctness and total refinement are related in the same way as for programs with one exit:

**Theorem 2** *For predicate transformers $S, T$:*

$$S \sqsubseteq T \quad \equiv \quad \forall p, q, r \cdot [\![p]\!]\, S\, [\![q,r]\!] \Rightarrow [\![p]\!]\, T\, [\![q,r]\!]$$

## 5 Partial Correctness

To make the connection between total correctness and partial refinement, we use partial correctness assertion $\langle\!|p|\!\rangle\, S\, \langle\!|q|\!\rangle$, as introduced in [23]. The notion of total correctness assumes that any possible failure has to be anticipated in the specification; the outcome in case of failure is specified by the exceptional postcondition. An implementation may not fail in any other way. Partial correctness weakens the notion of total correctness by allowing also "true" exceptions. For orderly continuation after an unanticipated exception, the restriction is that in that case, the state must not change. We introduce following notation:

$$\begin{array}{lcl} \langle\!|p|\!\rangle\, S\, \langle\!|q,r|\!\rangle & \equiv & \text{Under precondition } p, \text{ statement } S \text{ terminates and} \\ & & \text{– on normal termination } q \text{ holds finally,} \\ & & \text{– on exceptional termination } p \text{ or } r \text{ holds finally.} \end{array}$$

Both total and partial correctness guarantee termination when the precondition holds. If $\langle\!|p|\!\rangle\, S\, \langle\!|q, \mathsf{false}|\!\rangle$ holds, then statement $S$ does not modify the state when terminating exceptionally, and we write this more concisely as $\langle\!|p|\!\rangle\, S\, \langle\!|q|\!\rangle$. Let $p, q, r$ be state predicates:

$$\begin{array}{lcl} \langle\!|p|\!\rangle\, S\, \langle\!|q,r|\!\rangle & \widehat{=} & p \leq S\,(q, p \vee r) \\ \langle\!|p|\!\rangle\, S\, \langle\!|q|\!\rangle & \widehat{=} & p \leq S\,(q, p) \end{array}$$

Total correctness implies partial correctness, but not vice versa. The very definition of partial correctness breaks the duality between normal and exceptional postconditions that total correctness enjoys. This leads to some curious consequences in the correctness rules.

**Theorem 3** *Let $p, q, r$ be state predicates, $B, E$ be program expressions, $x$ be a variable, and $S, T$ be statements:*

$$\begin{array}{lcl} \langle\!|p|\!\rangle\ \mathsf{abort}\ \langle\!|q,r|\!\rangle & \equiv & p = \mathsf{false} \\ \langle\!|p|\!\rangle\ \mathsf{stop}\ \langle\!|q,r|\!\rangle & \equiv & \mathsf{true} \\ \langle\!|p|\!\rangle\ \mathsf{skip}\ \langle\!|q,r|\!\rangle & \equiv & p \Rightarrow q \\ \langle\!|p|\!\rangle\ \mathsf{raise}\ \langle\!|q,r|\!\rangle & \equiv & \mathsf{true} \\ \langle\!|p|\!\rangle\, x := E\, \langle\!|q,r|\!\rangle & \equiv & \mathsf{def}\, E \wedge p \Rightarrow q[x \backslash \mathsf{val}\, E] \\ \langle\!|p|\!\rangle\, x :\in ES\, \langle\!|q,r|\!\rangle & \equiv & \mathsf{def}\, ES \wedge p \Rightarrow \forall x' \in \mathsf{val}\, ES \cdot q[x \backslash x'] \\ \langle\!|p|\!\rangle\, S\,;T\, \langle\!|q,r|\!\rangle & \equiv & \exists h \cdot \langle\!|p|\!\rangle\, S\, \langle\!|h,r|\!\rangle \wedge \\ & & \quad [\![h]\!]\, T\, [\![q, p \vee r]\!] \\ \langle\!|p|\!\rangle\ \mathsf{try}\, S\, \mathsf{catch}\, T\, \langle\!|q,r|\!\rangle & \equiv & \exists h \cdot [\![p]\!]\, S\, [\![q,h]\!] \wedge \\ & & \quad [\![h]\!]\, T\, [\![q, p \vee r]\!] \\ \langle\!|p|\!\rangle\, S \sqcap T\, \langle\!|q,r|\!\rangle & \equiv & \langle\!|p|\!\rangle\, S\, \langle\!|q,r|\!\rangle \wedge \\ & & \langle\!|p|\!\rangle\, T\, \langle\!|q,r|\!\rangle \\ \langle\!|p|\!\rangle\ \mathsf{if}\, B\, \mathsf{then}\, S\, \mathsf{else}\, T\, \langle\!|q,r|\!\rangle & \equiv & [\![\mathsf{def}\, B \wedge \mathsf{val}\, B \wedge p]\!]\, S\, [\![q, p \vee r]\!] \wedge \\ & & [\![\mathsf{def}\, B \wedge \neg\, \mathsf{val}\, B \wedge p]\!]\, T\, [\![q, p \vee r]\!] \end{array}$$

The raise statement miraculously satisfies any partial correctness specification by failing and leaving the state unchanged. The rules for assignment and nondeterministic assignment have only conditions in case $E$ is defined; in case $E$ is undefined, the assignment fails without changing the state, thus satisfies the partial correctness specification automatically. We immediately get following consequence rule for any statement $S$:

$$\langle\!|p|\!\rangle\, S\, \langle\!|q,r|\!\rangle \wedge (q \leq q') \wedge (r \leq r') \quad \Rightarrow \quad \langle\!|p|\!\rangle\, S\, \langle\!|q', r'|\!\rangle$$

This rule allows the postconditions to be weakened, like for total correctness, but does not allow the precondition to be weakened.

For $S\,;T$ and $\mathsf{try}\, S\, \mathsf{catch}\, T$ let us consider the special case when $r = \mathsf{false}$:

$$\begin{array}{lcl} \langle\!|p|\!\rangle\, S\,;T\, \langle\!|q|\!\rangle & \equiv & \exists h \cdot \langle\!|p|\!\rangle\, S\, \langle\!|h|\!\rangle \wedge [\![h]\!]\, T\, [\![q,p]\!] \\ \langle\!|p|\!\rangle\ \mathsf{try}\, S\, \mathsf{catch}\, T\, \langle\!|q|\!\rangle & \equiv & \exists h \cdot [\![p]\!]\, S\, [\![q,h]\!] \wedge [\![h]\!]\, T\, [\![q,p]\!] \end{array}$$

The partial correctness assertion for $S\,;T$ is satisfied if $S$ fails without changing the state, but if $S$ succeeds with $h$, then $T$ must either succeed with the specified postcondition $q$, or fail with the original precondition $p$. For the partial correctness assertion of $\mathsf{try}\, S\, \mathsf{catch}\, T$ to hold, either $S$ must succeed with the specified postcondition $q$, or fail with $h$, from which $T$ either succeeds with $q$ or fails with the original precondition $p$.

# 6 Loop Theorems

Let $W \neq \emptyset$ be a well-founded set, i.e. a set in which there are no infinitely decreasing chains, and let $p_w$ for $w \in W$ be an indexed collection of state predicates called *ranked predicates* of the form $p_w = (p \wedge (\lambda\sigma \cdot v\,\sigma = w)$. Here $p$ is the *invariant* and $v$ the *variant* (*rank*). We define $p_{<w} = (\lambda\sigma \cdot (\exists w' \in W \cdot w' < w \wedge p_{w'}\,\sigma))$ to be true if a state predicate with lower rank than $p_w$ is true. The fundamental rules for total and partial correctness of loops are as follows:

**Theorem 4** *Assume that $B$ is a Boolean program expression, $r$ is a state predicate and $S$ is a statement. Assume that $p_w$ for $w \in W$ is a ranked collection of state predicates. Then*

$$\begin{array}{rl} & \forall w \in W \cdot [\![p_w \wedge \mathsf{def}\,B \wedge B]\!]\,S\,[\![p_{<w}, r]\!] \\ \Rightarrow & [\![p]\!]\ \mathsf{while}\,B\,\mathsf{do}\,S\,[\![p \wedge \mathsf{def}\,B \wedge \neg B, (p \wedge \neg\,\mathsf{def}\,B) \vee r]\!] \end{array}$$

*and*

$$\begin{array}{rl} & \forall w \in W \cdot \langle\!|p_w \wedge \mathsf{def}\,B \wedge B|\!\rangle\,S\,\langle\!|p_{<w}, r|\!\rangle \\ \Rightarrow & \langle\!|p|\!\rangle\ \mathsf{while}\,B\,\mathsf{do}\,S\,\langle\!|p \wedge \mathsf{def}\,B \wedge \neg B, r|\!\rangle \end{array}$$

These theorems separate the concerns of the two exits: on the normal exit, both rules are similar as with one exit, except that the postconditions include the definedness of the guard $B$. On the exceptional exit, the rule for total correctness is the same as its counterpart with one exit [2]; on the exceptional exit, it states that if the loop body $S$ exits exceptionally with postcondition $r$, then the loop exits exceptionally with postcondition $p \wedge \neg\,\mathsf{def}\,B$ (failure of the guard) or $r$ (failure of the loop body).

# 7 Partial Refinement

In this section, we relax total refinement to partial refinement and study its application. Partial refinement of predicate transformers $S, T$ is defined by:

$$S \sqsubseteq\!\!\!\sim T \quad \widehat{=} \quad S \sqcap \mathsf{raise} \sqsubseteq T$$

This implies that on normal exit, $T$ can only do what $S$ does. However, $T$ may fail when $S$ does not, but then has to preserve the initial state. For example, in such a case $T$ would still maintain an invariant and signal to the caller the failure. Note that the types of $S$ and $T$ require the initial state space to be the same as the final state space on exceptional exit. Partial refinement is related to partial correctness in the same way as total refinement is related to total correcntess:

**Theorem 5** *For predicate transformers $S, T$:*

$$S \sqsubseteq\!\!\!\sim T \quad \equiv \quad \forall p, q, r \cdot \langle\!|p|\!\rangle\,S\,\langle\!|q, r|\!\rangle \Rightarrow \langle\!|p|\!\rangle\,T\,\langle\!|q, r|\!\rangle$$

*Proof. Unfolding the definitions yields:*

$$(\forall q, r \cdot (S\,(q,r) \wedge r) \leq T\,(q,r)) \equiv (\forall p', q', r' \cdot p' \leq S\,(q', p' \vee r') \Rightarrow p' \leq T\,(q', p' \vee r'))$$

*This is shown by mutual implication. For any $p'$, $q'$ and $r'$, by letting $q$ and $r$ to be $q'$ and $p' \vee r'$ respectively, it is straightforward that the left side implies the right side. Implication of the other direction is more invovled: for any $p$ and $q$, letting $p'$, $q'$ and $r'$ to be $S\,(q,r) \wedge r$, $q$ and $r$ respectively gives us $(S\,(q,r) \wedge r) \leq S\,(q, S\,(q,r) \wedge r \vee r) \Rightarrow (S\,(q,r) \wedge r) \leq T\,(q, S\,(q,r) \wedge r \vee r)$. Since $S\,(q,r) \wedge r \vee r = r$, we*

*have* $(S(q,r) \wedge r) \leq S(q,r) \Rightarrow (S(q,r) \wedge r) \leq T(q,r)$*, which reduces to* $S(q,r) \wedge r \leq T(q,r)$*, thus the left side is implied.*

Total refinement implies partial refinement (in the same way as total correctness implies partial correctness); in addition to stop as top element, partial refinement has also raise as top element:

**Theorem 6** *For predicate transformers* $S, T$*:*

$$\begin{array}{l} S \sqsubseteq_{\sim} \mathsf{raise} \\ S \sqsubseteq T \quad \Rightarrow \quad S \sqsubseteq_{\sim} T \end{array}$$

The fact that $S \sqsubseteq_{\sim} \mathsf{raise}$ may be surprising, but it does allow for intentionally missing features, the second source of partiality discussed earlier on. Like total refinement, partial refinement is a preorder, as it is reflexive and transitive:

**Theorem 7** *For predicate transformers* $S, T, U$*:*

$$\begin{array}{l} S \sqsubseteq_{\sim} S \\ S \sqsubseteq_{\sim} T \wedge T \sqsubseteq_{\sim} U \quad \Rightarrow \quad S \sqsubseteq_{\sim} U \end{array}$$

Partial refinement is not antisymmetric, for example $\mathsf{stop} \sqsubseteq_{\sim} \mathsf{raise}$ and $\mathsf{raise} \sqsubseteq_{\sim} \mathsf{stop}$, but $\mathsf{stop} \neq \mathsf{raise}$. With respect to partial refinement, sequential composition is monotonic only in its first operand, while demonic choice and conditional statement are monotonic in both operands:

**Theorem 8** *For statements* $S, S', T$*:*

$$\begin{array}{lcl} S \sqsubseteq_{\sim} S' & \Rightarrow & S;T \sqsubseteq_{\sim} S';T \\ S \sqsubseteq_{\sim} S' \wedge T \sqsubseteq_{\sim} T' & \Rightarrow & S \sqcap T \sqsubseteq_{\sim} S' \sqcap T' \\ & \wedge & \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T \sqsubseteq_{\sim} \mathsf{if}\ B\ \mathsf{then}\ S'\ \mathsf{else}\ T' \end{array}$$

However, $S \sqsubseteq_{\sim} S'$ does not imply $T;S \sqsubseteq_{\sim} T;S'$ in general, since $T$ might modify the initial state on normal exit. Similarly, $S \sqsubseteq_{\sim} S'$ implies neither $S;;T \sqsubseteq_{\sim} S';;T$ nor $T;;S \sqsubseteq_{\sim} T;;S'$.

For *data refinement* we extend the refinement relationships to allow two predicate transformers on possibly different state spaces. We extend the definition of data refinement with predicate transformer, e.g. [10, 26], which uses an representation operation to link concrete and abstract spaces, to two post-conditions. Suppose $S : \mathscr{P}\Sigma \times \mathscr{P}\Sigma \to \mathscr{P}\Sigma$ and $T : \mathscr{P}\Delta \times \mathscr{P}\Delta \to \mathscr{P}\Delta$ are predicate transformers on $\Sigma$ and $\Delta$ respectively, and $R : \Sigma \leftrightarrow \Delta$ is the abstraction relation from $\Sigma$ to $\Delta$. Then we define:

$$T_R \quad \widehat{=} \quad [R];((T;\{R^{-1}\});;\{\bot, R^{-1}\})$$

The composition $(T;\{R^{-1}\});;\{\bot, R^{-1}\}$ applies the angelic update $\{R^{-1}\}$ to the normal outcome of $T$, terminating normally, and applies $\{R^{-1}\}$ to the exceptional outcome of $T$, terminating exceptionally. This can be equivalently expressed as $(T;;\{\bot, R^{-1}\});\{R^{-1}\}$. The demonic update $[R]$ maps the states in $\Sigma$ to states in $\Delta$. Then, after executing $T$, $\{R^{-1}\}$ and $\{\bot, R^{-1}\}$ map the states back to $\Sigma$ from states in $\Delta$, on each exit. In other words, $T_R$ is the projection of $T$ on $\mathscr{P}\Sigma \times \mathscr{P}\Sigma \to \mathscr{P}\Sigma$ through relation $R$. Now we can define *total data refinement* and *partial data refinement* between $S$ and $T$ through $R$ as:

$$\begin{array}{lcl} S \sqsubseteq_R T & \widehat{=} & S \sqsubseteq T_R \\ S \sqsubseteq_{\sim R} T & \widehat{=} & S \sqsubseteq_{\sim} T_R \end{array}$$

Note that here we could not define $S \sqsubseteq_{\sim R} T$ as $(S;[R]);;[R] \sqsubseteq_{\sim} [R];T$, due to the restriction of $\sqsubseteq_{\sim}$ that the on both sides the initial state space must be the same as the exceptional state space, which $[R];T$ obviously does not satisfy.

**Example: Limitation in Class Implementation**

Now let us revisit the introductory example of the class *IntCollection*. We consider an implementation using a fixed-size array. Using dynamic arrays or a heap-allocated linked list would be treated similarly, as an extension of a dynamic array and heap allocation may fail in the same way as a fixed-sized array may overflow. Since representing dynamic arrays or heaps complicates the model, we illustrate the refinement step using fixed-sized arrays. Let *SIZE* be a constant of type int:

$$
\begin{array}{l}
\mathsf{class}\ \mathit{IntCollection}1 \\
\quad \mathsf{int}\, l, \mathsf{int}[]\, a \\
\quad \mathsf{invariant}\ \mathit{inv}1 : 0 \le l \le \mathit{SIZE} \wedge \mathit{len}(a) = \mathit{SIZE} \wedge \\
\qquad (\forall x \cdot 0 \le x < l \Rightarrow \mathit{MIN} \le a[x] \le \mathit{MAX}) \\
\quad \mathsf{method}\ \mathit{init}1() \\
\qquad l := 0 \,;\, a := \mathsf{new}\ \mathsf{int}[\mathit{SIZE}]; \\
\quad \mathsf{method}\ \mathit{insert}1(n : \mathsf{int}) \\
\qquad l, a[l] := l+1, n; \\
\quad \mathsf{method}\ \mathit{sum}1() : \mathsf{int} \\
\qquad \mathsf{int}\, s, i := 0, 0; \\
\qquad \{\mathsf{loop\ invariant}\ \mathit{linv} : 0 \le i \le l \wedge \\
\qquad\quad s = \textstyle\sum x \in [0..i) \cdot a[x] \wedge \mathit{MIN} \le s \le \mathit{MAX})\} \\
\qquad \mathsf{while}\ i < l\ \mathsf{do} \\
\qquad\quad s, i := s + a[i], i+1; \\
\qquad \{s = \textstyle\sum x \in 0..l-1 \cdot a[x] \wedge \mathit{MIN} \le s \le \mathit{MAX}, \mathit{inv}1\} \\
\qquad \mathsf{result} := s
\end{array}
$$

The state spaces of classes *IntCollection* and *IntCollection*1 are $\mathsf{bag}(\mathsf{int})$ and $\mathsf{int} \times \mathsf{int}[]$ respectively. The invariant that links the state spaces is $b = \mathit{bagof}(a[0..l-1])$, where *bagof* converts an array to a bag with the same elements. The statement $a := \mathsf{new}\ \mathsf{int}[\mathit{SIZE}]$ might fail due to heap allocation failure. Writing $s^n$ for $n$ times repeating sequence $s$, allocation of an integer array is define as:

$$x := \mathsf{new}\ \mathsf{int}[n] \quad \widehat{=} \quad [\mathsf{true}, \mathsf{true}]\,; x := [0]^n$$

where $[\mathsf{true}, \mathsf{true}]$ might succeed or fail. Here the three methods refine those in class *IntCollection* respectively, formally $\mathit{init} \sqsubseteq_{\mathit{rel}} \mathit{init}1$, $\mathit{insert} \sqsubseteq_{\mathit{rel}} \mathit{insert}1$, and $\mathit{sum} \sqsubseteq_{\mathit{rel}1} \mathit{sum}1$, in which the relations are given as $\mathit{rel}\, b\, (l, a) \equiv b = \mathit{bagof}(a[0..l-1])$ and $\mathit{rel}1\, (b, \mathsf{result})\, (l, a, \mathsf{result}') \equiv b = \mathit{bagof}(a[0..l-1]) \wedge \mathsf{result} = \mathsf{result}'$, since result is part of the state space in *sum*1. We only give the proof sketch of the third one. Using *body* and *loop* as the abbreviations of $s, i := s + a[i], i+1$ and $\mathsf{while}\, i < l\, \mathsf{do}\, s, i := s + a[i], i+1$ respectively, and defining $B = (\lambda(s, i, l, a, \mathsf{result}) \cdot i < l)$, $\mathit{linv}_w = (\mathit{linv} \wedge (\lambda(s, i, l, a, \mathsf{result}) \cdot w = l - i))$, we have $\mathsf{def}\, B = \mathsf{true}$ and:

$$[\![\mathit{linv} \wedge \mathsf{def}\, B \wedge B]\!]\, \mathit{body}\, [\![\mathit{linv}_{<w}, \mathit{linv}]\!]$$

According to Theorem 4 we have

$$[\![\mathit{linv} \wedge (\lambda(s, i, l, a, \mathsf{result}) \cdot s = 0 \wedge i = 0)]\!]\, \mathit{loop}\, [\![\mathit{linv} \wedge \mathsf{def}\, B \wedge \neg B, \mathit{linv}]\!]$$

and $\mathit{linv} \wedge \mathsf{def}\, B \wedge \neg B \Rightarrow s = \sum x \in 0..l-1 \cdot a[x]$. We require that the exceptional postconditions must be independent of result since no value will be returned in failures. With the correctness rule for sequential composition we know that for arbitrary $q$, $r$:

$$[\![(\lambda\,(l,a,\mathsf{result})\cdot q\,(l,a,\sum x\in 0..l-1\cdot a[x]))\wedge r]\!]\,sum1\,[\![q,r]\!]$$

Since for arbitrary $q'$, $r'$,

$$sum\,(q',r') \quad = \quad (\lambda\,b,\mathsf{result}\cdot q'\,(\sum x\in b\cdot x,b)\wedge MIN\leq\sum x\in b\cdot x\leq MAX)\wedge r'$$

by definition we know that $sum\sqsubseteq_{\approx rel1} sum1$.

Furthermore, $inv1\,(l,a)\wedge rel\,b\,(l,a)\Rightarrow inv\,b$, which means that the original invariant *inv* is preserved by the new invariant $inv1$ through relation *rel*. In $sum1$, local variables $s$ and $i$ would be erased on both exits, thus in exceptional termination caused by arithmetic overflow inside the loop, the original state $(l,a)$ remains the same, maintaining the invariant on exceptional exit.

**Example: Incremental Development**

Another use of partial refinement is to express incremental development. When each of two programs handles some cases of the same specification, then their combination can handle no less cases, while remaining a partial refinement of the specification:

$$\begin{array}{ll} & S\sqsubseteq\!\!\!\approx \mathsf{if}\,B\,\mathsf{then}\,T\,\mathsf{else}\,\mathsf{raise}\ \wedge\ S\sqsubseteq\!\!\!\approx \mathsf{if}\,B'\,\mathsf{then}\,T'\,\mathsf{else}\,\mathsf{raise} \\ \Rightarrow & S\sqsubseteq\!\!\!\approx \mathsf{if}\,B\,\mathsf{then}\,T\,\mathsf{else}(\mathsf{if}\,B'\,\mathsf{then}\,T'\,\mathsf{else}\,\mathsf{raise}) \end{array}$$

Moreover, writing $\vee_c$ for *conditional disjunction* (conditional or), defined by $\mathsf{def}(B\vee_c B')=\mathsf{def}\,B\wedge(\mathsf{val}\,B\vee\mathsf{def}\,B')$ and $\mathsf{val}(B\vee_c B')=\mathsf{val}\,B\vee(\mathsf{def}\,B\wedge\mathsf{val}\,B')$, we have

$$\mathsf{if}\,B\,\mathsf{then}\,T\,\mathsf{else}(\mathsf{if}\,B'\,\mathsf{then}\,T'\,\mathsf{else}\,\mathsf{raise}) \quad = \quad \mathsf{if}\,B\vee_c B'\,\mathsf{then}(\mathsf{if}\,B\,\mathsf{then}\,T\,\mathsf{else}\,T')\,\mathsf{else}\,\mathsf{raise}$$

which allows the combination of more than two such programs in a switch-case style, since the right-hand side is again in the "if ... then ... else raise" form. With more and more such partial implementations combined this way, ideally more cases can be handled incrementally.

# 8 Conclusion

We introduced partial refinement for the description for programs that are unable to fully implement their specifications. Using predicate transformers with two arguments for the semantics of statements allows us to specify the behaviour on both normal and exceptional exits. The partial refinement relation suggests that programs should complete the task as expected on the normal exit, but when they fail and terminate on the exceptional exit, restore the (abstract) initial state. We have not addressed a number of issue, e.g. rules for partial refinement of loops and recursive procedures (the example of $IntCollection1$ proves refinement on predicate level). King and Morgan as well as Watson present a number of rules for total refinement, although using different programming constructs [15, 25]: $S>T$, pronounced "$S$ else $T$", can be defined here as $S\sqcap(T\,;\mathsf{raise})$, the *specification statement* $x:[p,q>r]$ with *frame* $x$ can be defined here as $\{p\}\,;[\overline{q},\overline{r}]$, where $\overline{q},\overline{r}$ lifts predicates $q,r$ to relations over $x$, the *exception block* $[\![S]\!]$ with $\mathsf{raise}\,H$ in its body, where $H$ is the exception handler, can be defined here as $S\,;;H$. This allows their total refinement rules to be used here. However, the purpose of partial refinement is different and we would expect different kinds of rules needed.

For two sources of partiality, inherent limitations of an implementation and intentionally missing implementation, the two examples in Sec. 7 illustrate how partial refinement can be used to help in

reasoning. For the third source of partiality, genuine faults, the approach needs further elaboration. In earlier work, we have applied the notions of partial correctness to three design patterns for fault tolerance, rollback, degraded service, and recovery block [23]. We believe that these ideas can be carried over to partial refinement.

Implementation restrictions are also addressed by IO-refinement, which allows the type of method parameters to be changed in refinement steps [5, 4, 7, 21, 24]. Partial refinement and IO-refinement are independent of each other and can be combined.

Data refinement as used here is a generalization of *forward data refinement* to two exits. Forward data refinement is known to be incomplete; for the predicate transformer model of statements, several alternatives have been studied, e.g. [9, 26]. It remains to be explored how these can be used for partial data refinement.

An alternative to using predicate transformers with two arguments is to encode the exit into the state and represent programs as predicate transformers with one argument, which we have not pursued. The advantage of predicate transformers with two arguments is that the state spaces on normal and exceptional exit can be different, which is useful for local variable declarations that can either fail and leave the state space unchanged or succeed and enlarge the state space. Having two separate postconditions may be methodologically stronger and syntactically shorter.

While predicate transformers allow to describe angelic nondeterminism, we have not made use of that. Finally, we note that partial refinement relies only a single exceptional exit. Common programming languages provide named exceptions and statements with several exits. Exploring a generalization of partial refinement to multiple exits is left as future work.

**Acknowledgements.** We are grateful to the reviewers, whose comments were exceptionally detailed and constructive.

## References

[1] R.-J. R. Back & M. Karttunen (1983): *A predicate transformer semantics for statements with multiple exits*. Unpublished manuscript, University of Helsinki.

[2] Ralph-Johan Back & Joakim von Wright (1998): *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, doi:10.1007/978-1-4612-1674-2.

[3] R.J.R. Back (1981): *On correct refinement of programs*. Journal of Computer and System Sciences 23(1), pp. 49 – 68, doi:10.1016/0022-0000(81)90005-2.

[4] R. Banach, M. Poppleton, C. Jeske & S. Stepney (2007): *Engineering and theoretical underpinnings of retrenchment*. Science of Computer Programming 67(2-3), pp. 301 – 329, doi:10.1016/j.scico.2007.04.002.

[5] E. Boiten & J. Derrick (1998): *IO-Refinement in Z*. In: Proc. Third BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-98.

[6] Flaviu Cristian (1984): *Correct and Robust Programs*. IEEE Transactions on Software Engineering 10(2), pp. 163–174, doi:10.1109/TSE.1984.5010218.

[7] J. Derrick & E. Boiten (2001): *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology, Springer, doi:10.1002/stvr.237.

[8] Edsger W. Dijkstra (1975): *Guarded commands, nondeterminacy and formal derivation of programs*. Commun. ACM 18(8), pp. 453–457, doi:10.1145/360933.360975.

[9] P. H. B. Gardiner & Carroll Morgan (1993): *A single complete rule for data refinement*. Formal Aspects of Computing 5(4), pp. 367–382, doi:10.1007/BF01212407.

[10] Paul Gardiner & Carroll Morgan (1991): *Data refinement of predicate transformers*. Theoretical Computer Science 87(1), pp. 143 – 162, doi:10.1016/0304-3975(91)90029-2.

[11] Intel (2013): *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. Available at `http://download.intel.com/products/processor/manual/325383.pdf`.

[12] Bart Jacobs (2001): *A Formalisation of Java's Exception Mechanism*. In D. Sands, editor: *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, Springer-Verlag, London, UK, pp. 284–301, doi:10.1007/3-540-45309-1_19.

[13] Dean Jacobs & David Gries (1985): *General correctness: a unification of partial and total correctness*. Acta Inf. 22(1), pp. 67–83, doi:10.1007/BF00290146.

[14] Ralph Jeffords, Constance Heitmeyer, Myla Archer & Elizabeth Leonard (2009): *A Formal Method for Developing Provably Correct Fault-Tolerant Systems Using Partial Refinement and Composition*. In Ana Cavalcanti & DennisR. Dams, editors: *FM 2009: Formal Methods*, Lecture Notes in Computer Science 5850, Springer Berlin Heidelberg, pp. 173–189, doi:10.1007/978-3-642-05089-3_12.

[15] Steve King & Carroll Morgan (1995): *Exits in the Refinement Calculus*. Formal Aspects of Computing 7(1), pp. 54–76, doi:10.1007/BF01214623.

[16] K. Rustan M. Leino & Jan L. A. van de Snepscheut (1994): *Semantics of Exceptions*. In Ernst-Rüdiger Olderog, editor: *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, North-Holland Publishing Co., pp. 447–466. Available at `http://authors.library.caltech.edu/26774/2/93-34.pdf`.

[17] Rajit Manohar & K. Rustan M. Leino (1995): *Conditional Composition*. Formal Aspects of Computing 7(6), pp. 683–703, doi:10.1007/BF01211001.

[18] Bertrand Meyer (1997): *Object-Oriented Software Construction*, 2nd edition. Prentice-Hall, Inc.

[19] Microsoft (2013): *NotImplementedException Class*. Available at `http://msdn.microsoft.com/en-us/library/system.notimplementedexception.aspx`.

[20] Microsoft (2013): *OverflowException Class*. Available at `http://msdn.microsoft.com/en-us/library/system.overflowexception.aspx`.

[21] Anna Mikhajlova & Emil Sekerinski (1997): *Class Refinement and Interface Refinement in Object-Oriented Programs*. In John Fitzgerald, Cliff Jones & Peter Lucas, editors: *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, Lecture Notes in Computer Science 1313, Springer-Verlag, pp. 82–101, doi:10.1007/3-540-63533-5_5.

[22] David L. Parnas (1978): *Designing software for ease of extension and contraction*. In: *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, IEEE Press, Piscataway, NJ, USA, pp. 264–277, doi:10.1109/TSE.1979.234169.

[23] Emil Sekerinski & Tian Zhang (2011): *A New Notion of Partial Correctness for Exception Handling*. In Borzoo Bonakdarpour & Tom Maibaum, editors: *Proceedings of the 2nd International Workshop on Logical Aspects of Fault-Tolerance*, pp. 116–132. Available at `https://ece.uwaterloo.ca/~bbonakda/LAFT11/papers/proc.pdf`.

[24] Susan Stepney, David Cooper & Jim Woodcock (1998): *More Powerful Z Data Refinement: Pushing the State of the Art in Industrial Refinement*. In Jonathan P. Bowen, Andreas Fett & Michael G. Hinchey, editors: *ZUM '98: The Z Formal Specification Notation*, Lecture Notes in Computer Science 1493, Springer Berlin Heidelberg, pp. 284–307, doi:10.1007/978-3-540-49676-2_20.

[25] G. Watson (2002): *Refining exceptions using King and Morgan's exit construct*. In: *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pp. 43–51, doi:10.1109/APSEC.2002.1182974.

[26] J. von Wright (1994): *The lattice of data refinement*. Acta Informatica 31(2), pp. 105–135, doi:10.1007/BF01192157.