On Excusable and Inexcusable Failures
Towards an Adequate Notion of Translation Correctness

Markus Müller-Olm and Andreas Wolf *

[1] Fachbereich Informatik, LS V, Universität Dortmund, 44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de
[2] Institut für Informatik und Praktische Mathematik,
Christian-Albrechts-Universität, 24105 Kiel, Germany
awo@informatik.uni-kiel.de

Abstract. The classical concepts of partial and total correctness identify all types of runtime errors and divergence. We argue that the associated notions of translation correctness cannot cope adequately with practical questions like optimizations and finiteness of machines. As a step towards a solution we propose more fine-grained correctness notions, which are parameterized in sets of acceptable failure outcomes, and study a corresponding family of predicate transformers that generalize the well-known wp and wlp transformers. We also discuss the utility of the resulting setup for answering compiler correctness questions.

Keywords: compiler, correctness, divergence, refinement, runtime-error, predicate transformer, verification

1 Introduction

Compilers are ubiquitous in today's computing environments. Their use ranges from the traditional translation of higher programming languages to conversions between data formats of a large variety. The rather inconspicuous use of compilers helps to get rid of architecture or system specific representations and allows thus to handle data or algorithms in a more convenient abstract form.

There is a standard theory for the syntactic aspects of compiler construction which is well-understood and documented in a number of text books (e.g. [1, 24–26]). It is applied easily in practice via automated tools like scanner and parser generators. This has made the construction of the syntactic phases of compilers, which has been a challenge back in the sixties, to a routine task nowadays.

This is different for the semantic phases concerned with the question, which output is to be generated for a given input. In this respect every translation task requires rather specific considerations and, due to the wide range of applications sketched above, no general approach is available or to be expected for this problem. Even if one restricts attention to a more narrow task, the translation of imperative programming languages considered in this paper, there is still no generally followed approach, although some well-studied frameworks like, e.g.,

* The work of the second author is supported by DFG grant La 426/15-1,2.

J. M. Wing, J. Woodcock, and J. Davies (Eds.): FM'99, LNCS 1709, pp. 1107–1126, 1999.

action semantics [19] exist. Of course much is known on efficient (and presumably correct) translation schemes and runtime environments and there is also a vast amount of literature on optimizations (a recent textbook is [20]). But these considerations do not build on a consistent, widely accepted semantic basis. As a consequence, subtle errors are present in generated code and it is difficult to fully understand which properties are guaranteed to transfer from source to target programs, in particular if aggressive optimization levels are employed in the compiler. This is exemplified by the surprising results experienced by many compiler users every now and then when running generated code.

In many applications errors and uncertainties, although annoying, can be tolerated. When compilers are used to construct software for safety-critical systems the matter changes dramatically. The mistrust in compilers is one of the reasons why such code often is certified on the level of machine- or assembler-code [15, 23]. Trusted and fully-understood compilers would permit a certification on the source language level. This would be less time-consuming, cheaper, and more reliable. From a practical point of view, the ultimate goal of compiler verification [4, 6, 12, 14, 16, 21, 22] should be to improve on this state of affairs.

Every compiler proof is in danger of burying the essential considerations under a mountain of technicalities, which could seriously affect the credibility of the established correctness claim. Thus a compiler proof should be based on a semantic definition in an abstract style. On the other hand, it is important that the semantic description is rather close to the intuition of the average programmer in order to avoid errors resulting from misunderstandings of or, seen from the perspective of the programmer, errors in the formal semantics definition. As most people have a rather concrete, operational intuition about the behavior of (imperative) programs, the ultimate reference point should thus be a rather concrete, operational semantics.

How can we resolve the obvious conflict between the requirements of using an operational as well as an abstract kind of semantics? We envision the following approach: The operational semantics is defined first and provides the ultimate reference. In particular, the correctness property to be established for the translation is interpreted in terms of the operational semantics. From the operational semantics, the more abstract semantics to be used in the compiler proof is derived. This involves defining the objects handled by the abstract semantics in terms of the operational semantics. Afterwards sufficiently strong properties of the abstract semantics are established that allow to reason in the compiler proof on the abstract level without directly recurring to its operational definition.

A particular benefit of this approach is that the abstract semantics can be suited to the specific correctness property to be established. We shall argue later in this paper (see Sect. 2) that there is no single universal notion of correct translation even in the simplified setting considered here. Instead there is a whole range of sensible notions and the abstract semantics can specifically be constructed to accommodate reasoning w.r.t. the chosen one.

So much for the context of this paper; let us now become a bit more concrete about our contribution. On the one hand, we are looking for a realistic, yet tractable, notion of translation correctness and, on the other hand, for abstract semantics suited to reasoning about it. We argue that common code-optimizing transformations and the limitations of finite machines give rise to different expectations about the relationship of the behavior of source and target code. We show that the notions of translation correctness that derive in a natural way from the classic idealized notions of partial and total correctness are not able to cope adequately with these topics. The problem results from the traditional identification of runtime errors and divergence. As a step towards a solution we propose relativized correctness notions that are parameterized in sets of acceptable failures. In order to facilitate compiler correctness proofs we also study relativized versions of the well-known wp and wlp predicate transformers and discuss the utility of the resulting setup. The aim of this line of research is to preserve as much as possible from the elegant appeal of the traditional idealized setting, while being able to cope with the more practical problems.

The remainder of this article is organized as follows. Section 2 discusses by means of small examples some pitfalls in defining semantic correctness conditions for practical compilers. The classical concepts of partial and total correctness and the associated notions of correct translation are revisited in Sect. 3 before we introduce the proposed relativized notions in Sect. 4. In particular we introduce a generalization of the classic wp and wlp predicate transformers [8, 11] called wrp (weakest relativized predicate transformer) and discuss its relationship to the classic transformers and its basic properties. In Sect. 6 we study wrp for the commands of a simple imperative programming language. These commands are applied to an example in Sect. 7 in order to indicate the utility of the proposed framework for answering translation correctness questions. We finish the paper with a number of concluding remarks.

2 On Correctness of Translations

Let us first of all set the stage for the technical discussion. We assume given a set Π of programs π. The reader should imagine imperative programs intended to compute on a certain non-empty set of states Σ. Computations of π start in a state $s \in \Sigma$; s represents the input to the program. There are three different types of computations: a computation may terminate regularly in a state $s' \in \Sigma$; it may end up in an error state; or it may diverge, i.e. run forever. Programs can be non-deterministic, i.e. there may be more than one computation from a given initial state s.

The details of program execution are not of interest for our purpose; we are only interested in the final outcomes of computations. Therefore, we assume that each program π is furnished with a relation $R(\pi) \subseteq \Sigma \times (\Sigma \cup \Omega)$. Here Ω is a non-empty set of *failure (or irregular) outcomes* disjoint from Σ. Intuitively, Ω contains the error states mentioned above and a special symbol ∞ representing divergence. Examples of error states are, e.g., 'div-by-zero', 'arithmetic overflow' etc. We call π *deterministic* if $R(\pi)$ is a function, i.e. if for any $s \in \Sigma$ there is at most one σ such that $(s, \sigma) \in R(\pi)$. As any practical program has at least one computation from a given initial state, we may safely assume that $R(\pi)$ is *total*, i.e. that there is an outcome σ with $(s, \sigma) \in R(\pi)$ for any $s \in \Sigma$. Unless otherwise stated this assumption is, however, not needed in this article.

We use the following conventions for the naming of variables: Σ is ranged over by s, Ω by ω, and $\Sigma \cup \Omega$ by σ. We also use the letter o to range over $\Omega - \{\infty\}$.

Intuitively, $(s, s') \in R(\pi)$ records that s' is a possible regular result of π from initial state s, $(s, o) \in R(\pi)$ means that error state o can be reached from s, and $(s, \infty) \in R(\pi)$ that π may diverge from s. $R(\pi)$ can be thought to be derived from an operational or denotational semantics. Relational definitions for familiar programming operators can be found in Sect. 6.

After these preparations, let us discuss what correctness properties we reasonably can expect from translations. Assume for the purpose of this discussion that π is a source program that has been translated to a target program π'. We will freely use various features and representations of imperative programs in the illustrating examples. For simplicity we assume that π and π' operate on the same state space.

If π' is to be a correct implementation of π, we clearly expect that the computations of π' are related to the computations of π in some sense. Usually, we are not interested in the intermediate states occurring in computations but just in the final outcomes produced.[1] Therefore, a relational semantics like the above introduced $R(\pi)$, which provides an abstraction of the possible computations of π to possible outcomes, is appropriate for defining correctness of translation.

At first glance, we might require that π' has the same outcomes as π for any given initial state, i.e. that $R(\pi') = R(\pi)$. But this requirement is far too strong. One of the reasons is that non-determinism in π might be resolved in a specific way in π'. Assume, e.g., that π contains an un-initialized local variable and that the result of π depends on the (arbitrary) initial value of this local variable, like in the following program.

```
BEGIN
  int y: y := 17
END;
BEGIN
  int z: x := z
END
```

The final value of x is arbitrary, i.e. we have $R(\pi) = \{(s, s[x \mapsto n]) \mid n \in \mathbb{Z}\}$ where $s[x \mapsto n]$ denotes the substitution of value n for the variable x in state s. The generated code π', on the other hand, might well provide the deterministic result 17, as it allocates for z the memory location used previously for y, which still contains y's old value. No sensible means can enforce full non-determinism in

[1] Of course, for programs with input/output instructions we are also interested in relating the communicated values. And even for strictly transformational programs, we might occasionally want to relate intermediate states; for example when we are interested in correctness of debuggers. But this is beyond the scope of this paper.

the target code and we should thus expect at most $R(\pi') \subseteq R(\pi)$: every outcome produced by the target code is a possible outcome of the source code. This is the very idea of *refinement*.

However, reality is not that simple: for various reasons, even $R(\pi') \subseteq R(\pi)$ is a too strong requirement. A realistic notion of correctness must also accommodate limitations of the execution mechanism and optimizations. Let us discuss each of these in turn.

Limited abilities of the implementation might give rise to failure outcomes of the target program that are not possible for the source program. Full implementation of recursion, e.g., requires stacks of unbounded size. Actual computers, however, provide only a finite amount of memory; we must thus be prepared to accept the outcome 'stack-overflow' or 'out-of-memory' every now and then when executing programs from languages with unrestricted use of recursion. Another example is restricted arithmetic. If the source language provides e.g. the full set of integers as a data type but the executing processor just uses, e.g., 32-bit representations, the outcome 'arithmetic overflow' will occur occasionally.

Such limitations could be handled in various ways. Firstly, we could try to model the limitations precisely in the source language semantics. This approach is often applied for restricted arithmetic (consider e.g. the ANSI/IEEE 754 standard for representation of the reals) but is generally impractical for e.g. bounded stack sizes as it would require very specific knowledge on the implementation when defining semantics of the source language. Secondly, we could simply enrich the source language semantics by the error outcomes which would allow them as possible results of the implementation. This would amount to considering

$$R(\pi) \cup \{(s, \mathit{error}) \mid \mathit{error} \text{ is an outcome reflecting a limitation}\}$$

the semantics of π. Thirdly, we could try to handle limitations as part of the relationship between $R(\pi)$ and $R(\pi')$. The latter is perhaps the most natural approach but it leads to complicated formalizations in practice. The predicate transformer semantics solution proposed below will somehow have the flavor of the second approach but avoids its somewhat unhandy nature.

Optimizations can replace error outcomes by arbitrary outcomes. As a first example consider the innocuously looking transformation pictured in Fig. 1, an instance of what is called *dead-code elimination* [20]. The justification for this transformation is that the value of e assigned to x in the initial assignment is never needed, as any path through the program overwrites x's value before using it by either the assignment $x := 12$ or $x := 42$. Hence it should not be necessary to perform the evaluation of e and the assignment $x := e$ at all. But suppose that e is the expression $1/0$. Then the left program is guaranteed to produce the error outcome 'div-by-zero' while the right program can have, depending on P, whatever outcomes you want! (Note, that it is not always as obvious as in this example that evaluation of an expression at a certain point in a program might lead to a run-time error; in general this is undecidable.)

As a second example of an optimization consider the *code motion transformation* [20] in Fig. 2 where b, e and f are assumed not to contain y, and g

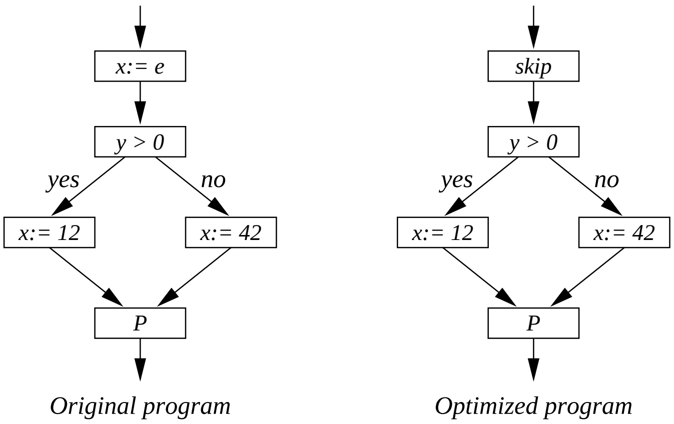

Fig. 1. Elimination of dead code.

is assumed not to contain x. In the optimized program the assignment $y := g$ appearing in both branches is moved to the start of the program in order to save code. The reasoning is that g can safely be evaluated before the branching, as it is evaluated on each path anyhow (in traditional parlance one says g is 'very busy' or 'downward safe' at the initial node). Assume now that evaluation of e, f and g can lead to different error outcomes, say g to an arithmetic overflow and e, f to a division by zero. Then the left program produces a 'division-by-zero' outcome while the optimized right program produces an arithmetic overflow. The reason is that the notion of downward safety, disregards the possibility of errors.

In summary, many common optimizing transformations can replace certain error outcomes by different regular and irregular outcomes. Some optimizations can even introduce new errors into regularly terminating programs because they compute intermediate values that are not computed by the original program. Examples are strength reduction transformations and naive code motion transformations that move loop-invariant pieces of code out of loops.

Should optimizations be banned from verified compilers for these reasons? No, this would throw out the baby with the bath water in our opinion. Optimizations play a very important role in increasing the efficiency of program execution and in many applications effects like the above can be tolerated. But the possible effects should be precisely understood and documented. A user should thus be enabled to judge which optimizations are permissible for his particular application and to select just these (e.g. by means of compiler switches).

As a curiosity, we mention that common efficiency-improving compiler options can even lead to a translation of terminating programs into non-terminating ones in rare cases. The Modula-2 loop $\mathsf{for}\ i := 0\ \mathsf{to}\ \mathsf{maxcard}\ \mathsf{do} \ldots$, for instance, obviously is terminating. A typical implementation is the following: i is initialized with the value 0; each iteration starts with a check whether i is still in the

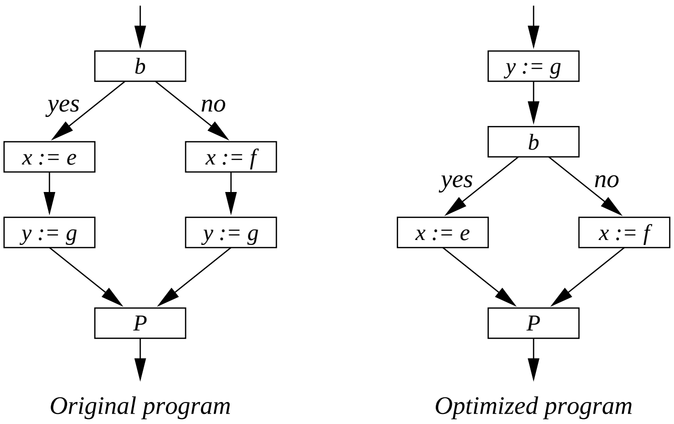

Fig. 2. A code motion transformation.

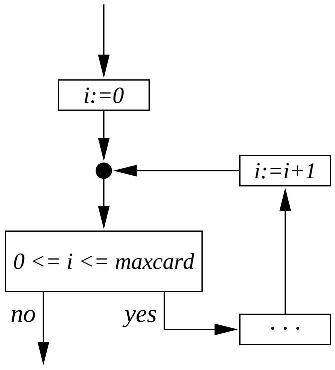

Fig. 3. Prototypical implementation of a for-loop.

range $0 \leq i \leq \mathsf{maxcard}$; at the end of each iteration i is incremented. This is illustrated in Fig 3. Now suppose an implementation disregards arithmetic overflows in order to increase the performance. Then the incrementation of i at the end of the iteration $i = \mathsf{maxcard}$ effectively sets i to 0 due to the representation of numbers. It also sets the carry-flag but sadly this is ignored. Now the test whether i is still in the range $0 \leq i \leq \mathsf{maxcard}$ succeeds! Thus, this implementation of the loop, which is actually found in practice, will not terminate in contrast to the original program.

It should have become clear that there is no single universal notion of correct translation but that different applications and translation schemes preserve a different amount from the behavior of programs. For a specific translation scheme the set $\Sigma \cup \Omega$ of (regular and irregular) outcomes can be partitioned into three sets:

- a set PO ('*preserved outcomes*') of outcomes that has to be preserved literally;
- a set AO ('*accepted outcomes*') of outcomes that may arise as result of target program executions even if not present in the source program's semantics (e.g. 'stack-overflow' or 'out-of-memory'); and
- a set CO ('*chaotic outcomes*') of outcomes of source programs that might lead to arbitrary outcomes in the target program (e.g. arithmetic errors in connection with dead code elimination).

Typically the regular outcomes belong to the set PO but also irregular outcomes may, e.g. 'division-by-zero', for debugging purposes.

Now suppose given a partition of $\Sigma \cup \Omega$ as described above. We call π' a *correct implementation* of π *w.r.t. preserved outcomes PO, accepted outcomes AO, and chaotic outcomes CO* if for all $(s, \sigma) \in R(\pi')$ (at least) one of the following is valid:

(a) σ is a preserved outcome of a computation of π from s, i.e. $\sigma \in PO \wedge (s, \sigma) \in R(\pi)$,
(b) σ is an accepted outcome, i.e. $\sigma \in AO$, or
(c) there is a chaotic outcome of a source program computation from s, i.e. $\exists \sigma' \in CO : (s, \sigma') \in R(\pi)$.

There are various ways of characterizing this property as an inclusion between relations derived from $R(\pi)$ and $R(\pi')$. One of them is the following that we are going to take as a definition.

Definition 1 (Correct implementation). *π' implements π w.r.t. preserved outcomes PO, accepted outcomes AO, and chaotic outcomes CO if and only if*

$$R(\pi') \subseteq R(\pi) \cup \{(s, \sigma) \mid \sigma \in AO \vee \exists \sigma' \in CO : (s, \sigma') \in R(\pi)\} \ .$$

Often divergence and runtime-errors are identified in simplified semantic treatments of programming languages. This has proved very helpful in establishing a rich and useful theory of program verification [2, 7, 13] and program refinement [3, 17, 18].[2] However, this idealization does not lead to a realistic notion of correct implementation: on the one hand, the single irregular outcome must be treated as chaotic, in order to accommodate the effect of optimizations like dead code elimination, because dead code elimination can change the single irregular outcome (which could represent e.g. 'div-by-zero' in this case) to an arbitrary outcome. On the other hand, it must be treated as acceptable, as it could also

[2] We should mention that Apt and Olderog [2] do consider different irregular outcomes of programs: divergence, failure, and deadlock. In their proof theories divergence and failure are identified, but in Chaps. 7 and 8 they introduce a notion of *weak total correctness* that reflects the distinction between divergence and deadlock. Weak total correctness is an instance of our relative correctness notion (Sect. 4). It is introduced in [2] in order to justify proof rules for total correctness and is said to be not of interest in itself. In contrast, we emphasize here that relative correctness is indeed often of independent interest.

report on a limitation of the execution mechanisms at hand (e.g. standing for 'out-of-memory').

We propose more fine-grained notions of program correctness and refinement intended to allow an adequate treatment of these more practical questions, while preserving as much as possible from the idealized setup. Before doing so let us have a more careful look at the classical treatment of program correctness and the notions of translation correctness to which they give rise, because our proposal is modeled on this.

3 The Classical Setup

3.1 Program Verification and Predicate Transformers

In Hoare-style program verification one is interested in proving programs partially or totally correct w.r.t. pre- and postconditions on the set of regular states. For the purpose of this paper a predicate is identified with the set of states for which it is valid. Thus, the set of predicates is $Pred = 2^{\Sigma}$; we range over $Pred$ by the letters ϕ and ψ. $Pred$, ordered by set-inclusion $\subseteq$, is a standard example of a complete Boolean lattice. The meet and join operations are $\cap$ and $\cup$; they represent conjunction and disjunction respectively, $\neg\phi$ is the complement of predicate ϕ, i.e. $\neg\phi = \Sigma - \phi$, the strongest (the smallest) and the weakest (the greatest) predicate w.r.t. this order is $\emptyset$ and Σ. We denote the latter also by `false` and `true`.

The classic literature on Hoare-style program verification and the refinement calculus identifies, for the sake of simplicity, divergence and failure outcomes or fully ignores failures. In our setting this amounts to assuming that Ω contains just one symbol, $\perp$, which represents any kind of irregular outcomes, divergence and failures, $R(\pi)$ is then a subset of $\Sigma \times (\Sigma \cup \{\perp\})$. For the purpose of the later discussion it is, however, more convenient to stay with the distinction between different irregular outcomes in the relational semantics. The definitions of total and partial correctness below treat all irregular outcomes as if they were identified and can thus equivalently be read in both models.

Partial correctness of a program π w.r.t. a precondition ϕ and postcondition ψ, denoted by $\{\phi\}\pi\{\psi\}$ can be defined as follows.

$$\{\phi\}\pi\{\psi\} \quad \text{iff} \quad \forall s, \sigma : s \in \phi \wedge (s, \sigma) \in R(\pi) \;\Rightarrow\; \sigma \in \psi \cup \Omega \ .$$

Intuitively, π is partially correct if each regularly terminating computation from a state in ϕ results in a state in ψ. Note, how the restriction to regular results is expressed by allowing all outcomes in Ω.

Total correctness of π w.r.t. precondition ϕ and postcondition ψ, denoted by $[\phi]\pi[\psi]$ additionally requires that there are no irregular computations from states in ϕ. This can be expressed nicely by *not* allowing outcomes in Ω.

$$[\phi]\pi[\psi] \quad \text{iff} \quad \forall s, \sigma : s \in \phi \wedge (s, \sigma) \in R(\pi) \;\Rightarrow\; \sigma \in \psi \ .$$

An elegant way of expressing partial and total correctness is by means of predicate transformers, i.e. mappings on the space of predicates. Dijkstra [8, 9] considers two predicate transformers. The *weakest liberal precondition* transformer wlp is suited to partial correctness and the *weakest precondition* transformer wp to total correctness.

A few words on notation: it is convenient and customary in connection with predicate transformers to denote function application by an infix dot, i.e. writing $f.x$ instead of the more familiar $f(x)$. Moreover, we adopt the usual convention that function application associates to the left, i.e. $f.x.y$ means $(f.x).y$.

For a program π, both $\mathsf{wlp}.\pi$ and $\mathsf{wp}.\pi$ are of type $2^{\Sigma} \to 2^{\Sigma}$. As their name suggests $\mathsf{wlp}.\pi.\psi$ ($\mathsf{wp}.\pi.\psi$) is the weakest predicate ϕ satisfying the Hoare-triple $\{\phi\}\pi\{\psi\}$ (resp. $[\phi]\pi[\psi]$) (see (1) and (2) below).

Based on the relational semantics $R(\pi)$ of a program π the predicate transformers $\mathsf{wlp}.\pi$ and $\mathsf{wp}.\pi$ can be defined as follows.

$$\begin{aligned} \mathsf{wlp}.\pi.\psi &= \{s \in \Sigma \mid \forall \sigma : (s,\sigma) \in R(\pi) \Rightarrow \sigma \in \psi \cup \Omega\} \\ \mathsf{wp}.\pi.\psi &= \{s \in \Sigma \mid \forall \sigma : (s,\sigma) \in R(\pi) \Rightarrow \sigma \in \psi\} \ . \end{aligned}$$

Their relationship to partial and total correctness is captured by the following equivalences, the proof of which is straightforward. These equivalences could also serve as the definition of wlp and wp.

$$\phi \subseteq \mathsf{wlp}.\pi.\psi \quad \text{iff} \quad \{\phi\}\pi\{\psi\} \ . \tag{1}$$

$$\phi \subseteq \mathsf{wp}.\pi.\psi \quad \text{iff} \quad [\phi]\pi[\psi] \ . \tag{2}$$

$\mathsf{wlp}.\pi$ and $\mathsf{wp}.\pi$ provide abstractions of $R(\pi)$ suited to partial and total correctness. Both carry less information than $R(\pi)$. This can be seen from the following examples in which we use $\mid$ to denote (demonic) nondeterministic choice.[3]

$$\begin{aligned} \pi &\stackrel{\text{def.}}{=} x := e \mid \mathsf{while}\ \mathtt{true}\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od} \\ \pi' &\stackrel{\text{def.}}{=} x := e \end{aligned}$$

Here $\mathsf{wlp}.\pi$ equals $\mathsf{wlp}.\pi'$ because the two programs yield the same result, if they terminate. On the other hand, for

$$\begin{aligned} \pi &\stackrel{\text{def.}}{=} x := 12 \mid \mathsf{while}\ \mathtt{true}\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od} \\ \pi' &\stackrel{\text{def.}}{=} x := 42 \mid \mathsf{while}\ \mathtt{true}\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od} \end{aligned}$$

$\mathsf{wp}.\pi$ equals $\mathsf{wp}.\pi'$ because both programs may diverge. Obviously, in both examples $R(\pi)$ and $R(\pi')$ differ.

It is interesting to note that in the traditional model where $|\Omega| = 1$, $R(\pi)$ can be reconstructed from $\mathsf{wp}.\pi$ *together* with $\mathsf{wlp}.\pi$. More specifically,

$$\begin{aligned} R(\pi) = \quad & \{(s,s') \mid s \notin \mathsf{wlp}.\pi.(\Sigma - \{s'\})\} \\ & \cup \{(s,\bot) \mid s \notin \mathsf{wp}.\pi.\mathtt{true}\} \ . \end{aligned}$$

This is no longer true if $|\Omega| > 1$, as, intuitively speaking, the information about the different causes of failures is not recorded in the predicate transformers.

[3] Semantics of $\mid$ is characterized by the identity $R(\pi \mid \pi') = R(\pi) \cup R(\pi')$.

3.2 Implementation Correctness

There are three natural ways to approach translation correctness. First, one can focus on *properties* that transfer from source to target programs. This point of view is particularly adequate if one is interested mainly in program proving. Second, one might focus on the *outcomes* produced by the source and target program, if one has a particular interest in actually interpreting results of program execution. Finally, one might look for a formulation in terms of *refinement*. The latter is of particular importance when *proving* correctness of translations. Fortunately, there are natural notions of implementation correctness that accommodate all three points of view as we will see in a moment.

The idea of the property-oriented point of view is to consider a program π' a correct implementation of a program π if validity of all properties from a certain class of interest transfers from π to π'. Two natural notions of this kind are preservation of partial and total correctness.

Definition 2 (Preservation of partial and total correctness).

1. *A program* π' implements π w.r.t. preservation of partial correctness (PPC) *if the following holds:* $\forall \phi, \psi : \{\phi\}\pi\{\psi\} \Rightarrow \{\phi\}\pi'\{\psi\}$.
2. π' implements π w.r.t. preservation of total correctness (PTC) *if the following holds:* $\forall \phi, \psi : [\phi]\pi[\psi] \Rightarrow [\phi]\pi'[\psi]$.

Note that, while total correctness implies partial correctness, the corresponding preservation properties are unrelated. Neither does PPC imply PTC nor vice versa.

If one concentrates on outcomes one wants to know which outcomes of the source program can result in which outcomes of the target program. This point of view was taken in Sect. 2 and we resort in the theorem below to the notion of correct implementation introduced in Def. 1. The theorem shows that we can interpret PPC and PTC also in terms of outcomes in a natural way.

Theorem 3 (Outcome interpretation of PPC and PTC).

1. *π' implements π w.r.t. PPC iff π' implements π w.r.t. preserved outcomes Σ, accepted outcomes Ω, and chaotic outcomes $\emptyset$.*
2. *π' implements π w.r.t. PTC iff π' implements π w.r.t. preserved outcomes Σ, accepted outcomes $\emptyset$, and chaotic outcomes Ω.*

Hence for PPC we have to choose $AO = \Omega$ and $CO = \emptyset$ and for PTC, just to the opposite, $AO = \emptyset$ and $CO = \Omega$; in both cases we take $PO = \Sigma$.

The goal of the refinement-oriented view is to devise a semantic model of programs that accommodates reasoning about implementation relationships. More specifically, one is looking for an interpretation of programs in a semantic space that is equipped with an ordering; π' should implement π iff its interpretation in the model is related to π's by the order.

For PPC and PTC adequate interpretations are well-known: they are given by wlp and wp. The semantic space is the set of monotonic predicate transformers $2^\Sigma \to 2^\Sigma$. It is ordered by the pointwise extensions $\leq$ of the inclusion relation on 2^{Σ}, which is defined by $f \leq g$ iff $\forall \psi : f.\psi \subseteq g.\psi$: a predicate transformer g is considered a refinement of another predicate transformer f if it establishes all postconditions from weaker preconditions. Restricting attention to *monotonic* predicate transformers (i.e. those transformers for which $f.\psi \subseteq f.\phi$ if $\psi \subseteq \phi$) makes functional composition monotonic.

As indicated, refinement in the space of predicate transformers corresponds to PPC and PTC.

Theorem 4 (Refinement characterization of PPC and PTC).
1. *π' implements π w.r.t. PPC iff* $\mathsf{wlp}.\pi \leq \mathsf{wlp}.\pi'$.
2. *π' implements π w.r.t. PTC iff* $\mathsf{wp}.\pi \leq \mathsf{wp}.\pi'$.

In the traditional setup, where $|\Omega| = 1$, the idealized notion of implementation correctness $R(\pi') \subseteq R(\pi)$ can be regained from wlp and wp. In this case,

$$R(\pi') \subseteq R(\pi) \quad \text{iff} \quad \mathsf{wlp}.\pi \leq \mathsf{wlp}.\pi' \wedge \mathsf{wp}.\pi \leq \mathsf{wp}.\pi' \ . \tag{3}$$

Again, this is no longer true if $|\Omega| > 1$.

It follows from (3) that for the translations discussed in Sect. 2 refinement w.r.t. either PPC or PTC does not hold, as they did not satisfy $R(\pi') \subseteq R(\pi)$. Thus, many practical compilers are either incorrect in the sense of PPC or PTC. A little further reflection unveils that the situation is as worse as it could be: reported limitations of the execution mechanism prohibit PTC, optimizations prohibit PPC. Consequently, most practical compilers preserve neither partial nor total correctness!

However, not the compilers are to be blamed for this sad state of affairs but the restricted selectivity of the notions of partial and total correctness, particularly their indiscriminate identification of any kind of run-time errors and divergence. We, therefore, establish a finer framework in the next section.

4 The Relativized Setup

4.1 Relative Correctness and Relativized Predicate Transformers

For evaluating partial correctness assertions all irregular outcomes of programs are disregarded; in contrast in total correctness assertions all irregular outcomes are taken as disproof. The correctness concept we are going to elaborate now is built around the idea of *parameterizing* assertions w.r.t. the set of accepted outcomes. The irregular outcomes that are not accepted are taken as disproof.

Suppose given a set $A \subseteq \Omega$ of outcomes to be accepted. We introduce the notion of a program π being *relatively correct* w.r.t. a precondition ϕ, a postcondition ψ, and the set A of accepted outcomes, denoted by $\langle\phi\rangle\pi\langle\psi\rangle_A$ for short. It is defined as follows:

$$\langle\phi\rangle\pi\langle\psi\rangle_A \quad \text{iff} \quad \forall s, \sigma : s \in \phi \wedge (s, \sigma) \in R(\pi) \ \Rightarrow \ \sigma \in \psi \cup A \ .$$

Intuitively, a program π is relatively correct if the following holds.

> Whenever π is started in a state contained in ϕ we can be sure that either π terminates regularly in a state contained in ψ, irregularly with a failure in A, or, if $\infty \in A$, diverges.

We can also define a corresponding predicate transformer along the lines of wlp and wp. It is called the *weakest relativized precondition* transformer $\mathsf{wrp}_A.\pi : \mathit{Pred} \to \mathit{Pred}$. [4]

$$\mathsf{wrp}_A.\pi.\psi \ = \ \{s \in \Sigma \mid \forall \sigma : (s, \sigma) \in R(\pi) \Rightarrow \sigma \in \psi \cup A\} \ .$$

Again, we have the following equivalence, that shows that $\mathsf{wrp}_A.\pi$ indeed deserves the name weakest relativized precondition transformer.

$$\phi \subseteq \mathsf{wrp}_A.\pi.\psi \quad \text{iff} \quad \langle\phi\rangle\pi\langle\psi\rangle_A \ .$$

These relativized notions generalize the classical ones. It is easy to see that partial and total correctness are just the border cases of relative correctness for the sets $A = \Omega$ and $A = \emptyset$. Similarly we have for wlp and wp:

$$\mathsf{wlp}.\pi \ = \ \mathsf{wrp}_\Omega.\pi \qquad \text{and} \qquad \mathsf{wp}.\pi \ = \ \mathsf{wrp}_\emptyset.\pi \ ,$$

so wp and wlp are just the extreme relativized predicate transformers.

4.2 Implementation Correctness

Each set $A \subseteq \Omega$ now gives rise to a notion of translation correctness relatively to A. As in the classic case it can be characterized in terms of preservation, refinement, and outcomes. More precisely, we have the following theorem, where we again refer to the notion introduced in Def. 1.

Theorem 5 (Preservation of relative correctness). *For all programs π, π' and accepted sets of outcomes $A \subseteq \Omega$, the following three conditions are equivalent.*

1. *(Preservation)* $\forall \phi, \psi : \langle\phi\rangle\pi\langle\psi\rangle_A \Rightarrow \langle\phi\rangle\pi'\langle\psi\rangle_A$.
2. *(Refinement)* $\mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi'$.
3. *(Outcomes) π' is a correct implementation of π w.r.t. preserved outcomes Σ, accepted outcomes A, and chaotic outcomes $\Omega - A$.*

The intuitive interpretation of these conditions is as follows. There is no restriction for the behavior of the target program from initial states for which the source program has a failure outcome in $\Omega - A$; otherwise, we don't care about the accepted outcomes in A, and every other outcome of the target program must also be possible for the source program.

[4] If we would allow error outcomes in postconditions, we could have defined $\mathsf{wrp}_A.\pi.\psi \ = \ \mathsf{wp}.\pi.(\psi \cup A)$. But this would destroy the homogeneity of pre- and postconditions, and lead to a more complicated definition of sequential composition of predicate transformers.

This looks fine, but it is not as general as the aspired notion of correct implementation from Def. 1, where we assumed that the set of outcomes $\Sigma \cup \Omega$ is partitioned into preserved, accepted and chaotic outcomes PO, AO and CO. From the definition of wrp it is clear that each element of the set A that we carry in wrp's index is just accepted, not preserved; and the outcomes in $\Omega - A$ are treated chaotically. What about those failure outcomes we really want to preserve? A compiler user, for instance, might require that an observed outcome 'div-by-zero' indeed is caused by a division by zero on the source level. Roughly speaking we have to treat those outcomes twice, firstly as accepted, and secondly as chaotic. If we can prove refinement for each of these choices, we have proved that it is preserved. More formally, we have the following result.

Theorem 6. *π' implements π w.r.t. preserved outcomes PO, accepted outcomes AO, and chaotic outcomes CO iff, for all A with $AO \subseteq A \subseteq AO \cup (PO \cap \Omega)$, $\mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi'$.*

Thus, although the notion of correct implementation from Def. 1 is not accommodated by refinement reasoning w.r.t. a single fixed set A, it can still be established by refinement arguments that are appropriately parameterized in A.

As a corollary to Theorem 6, the relational inclusion $R(\pi') \subseteq R(\pi)$ can also be established with wrp-based reasoning. To see this, just choose $PO = \Sigma \cup \Omega$ and $AO = CO = \emptyset$ and observe that the notion of correctness of implementations degenerates to the relational inclusion $R(\pi') \subseteq R(\pi)$ with this choice.

Corollary 7. *$R(\pi') \subseteq R(\pi)$ iff $\mathsf{wrp}_A.\pi \leq \mathsf{wrp}_A.\pi'$ for all $A \subseteq \Omega$.*

Relativized refinement enables us hence to be as fine-grained w.r.t. outcomes as on the relational level, if desired.

5 Properties of wrp

In the next lemma we collect some basic properties enjoyed by the family of wrp-transformers. Validity of 4, 7, and 8 depends on the program relation $R(\pi)$ being *total*.

Lemma 8. *Suppose π is a program, ψ a predicate, and $A, B \subseteq \Omega$ are sets of irregular outcomes.*

1. $\mathsf{wrp}_{A \cap B}.\pi \;=\; \mathsf{wrp}_A.\pi \wedge \mathsf{wrp}_B.\pi$.
2. $\mathsf{wrp}_A.\pi \;\leq\; \mathsf{wrp}_B.\pi$, *if* $A \subseteq B$.
3. $\mathsf{wrp}_A.\pi.\psi \;=\; \mathsf{wrp}_B.\pi.\psi \cap \mathsf{wrp}_A.\pi.\mathtt{true}$, *if* $A \subseteq B$.
4. $\mathsf{wrp}_\emptyset.\pi.\mathtt{false} \;=\; \mathtt{false}$.
5. $\mathsf{wrp}_A.\pi$ *is positively conjunctive, i.e. distributes over every non-empty conjunction of predicates.*
6. $\mathsf{wrp}_\Omega.\pi$ *is universally conjunctive, i.e. distributes over every, even the empty conjunction of predicates.*
7. $\mathsf{wrp}_A.\pi.\psi \;\subseteq\; \neg(\mathsf{wrp}_{\Omega - A}.\pi.\neg\psi)$.
8. $\mathsf{wrp}_A.\pi.\psi \;=\; \neg(\mathsf{wrp}_{\Omega - A}.\pi.\neg\psi)$ *iff π is deterministic.*

Dijkstra and Scholten [9] discuss so-called *healthiness conditions* of wp and wlp. In our notation they look as follows.

- $\mathsf{wp}.\pi.\psi \ = \ \mathsf{wlp}.\pi.\psi \cap \mathsf{wp}.\pi.\mathtt{true}$ (Pairing condition).
- $\mathsf{wp}.\pi.\mathtt{false} \ = \ \mathtt{false}$ (Excluded miracle).
- wp is positively conjunctive.
- wlp is universally conjunctive.

In the sense of [9] these properties have to be satisfied by a pair of predicate transformers to model an adequate semantics of implementable programs. The items 3–6 of Lemma 8 show how the healthiness conditions generalize to the family of wrp-transformers. Note that in our framework they are derived properties and not postulates as in [9], due to our point of view that predicate transformer semantics is derived from an underlying, more concrete operationally-based semantics. Property 8 generalizes the equivalence

- $\mathsf{wp}.\pi.\psi \ = \ \neg\mathsf{wlp}.\pi.\neg\psi$ iff π is deterministic

that is used as the definition of deterministic programs in [9].

6 Programming Operators

In this section we discuss briefly the wrp characterizations of typical commands of an imperative programming language. More specifically, we consider assignments $x := e$, conditionals $\mathsf{if}\ b\ \mathsf{then}\ \pi_1\ \mathsf{else}\ \pi_2$, while-loops $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$, and sequential composition. We would like to show that wrp enjoys similar, and only slightly more complicated characterizations as the classic predicate transformers. Reasoning in terms of wrp seems obviously to be more tractable than reasoning in terms of an operational or relational semantics.

We suppose given three additional sets of syntactic objects: variables x, expressions e and Boolean expressions b. The set of variables is denoted by Var. We assume interpretation functions for expressions and Boolean expressions $\mathcal{E}(e) : \Sigma \to (\mathit{Val} \cup \Omega)$ and $\mathcal{B}(b) : \Sigma \to (\mathbb{B} \cup \Omega)$. Here Val is the value set of variables; we range over Val by the letter v. The set $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ represents the truth values. For the purpose of this section, states are valuations of variables, i.e. $\Sigma = (\mathit{Var} \to \mathit{Val})$. As usual $s[x \mapsto v]$ denotes the substitution of value v for the variable x in state s. Intuitively, results $\mathcal{E}(e)(s), \mathcal{B}(b)(s) \in \Omega$ represent failures during evaluation of (Boolean) expressions. Such failures are assumed to propagate to the statement level.

For simplicity we identify syntax and semantics when writing concrete predicates. In order to deal with partially defined expressions we assume special types of basic predicates: $\mathsf{def}(e)$ and $\mathsf{in}_A(e)$ for expressions e and $A \subseteq \Omega$, and $\mathsf{def}(b)$, $\mathsf{in}_A(b)$, $b = \mathsf{tt}$, and $b = \mathsf{ff}$ for Boolean expressions b. They are interpreted as follows: $\mathsf{def}(e) \overset{\text{def.}}{=} \{s \mid \mathcal{E}(e)(s) \in \mathit{Val}\}$, $\mathsf{in}_A(e) \overset{\text{def.}}{=} \{s \mid \mathcal{E}(e)(s) \in A\}$, $b = \mathsf{tt} \overset{\text{def.}}{=} \{s \mid \mathcal{B}(b)(s) = \mathsf{tt}\}$. The interpretation of the remaining predicates is obvious. Note that Boolean expressions can have 'undefined' failure results while predicates cannot.

Let us first consider assignments, conditionals, and the sequential composition operator. Their relational semantics reads as follows, where we rely on the convention (from Sect. 2) that s ranges over Σ and ω over Ω.

$$\begin{aligned}
R(x := e) = {} & \{(s, s[x \mapsto v]) \mid \mathcal{E}(e)(s) = v\} \\
& \cup \{(s, \omega) \mid \mathcal{E}(e)(s) = \omega\} \\
R(\mathsf{if}\ b\ \mathsf{then}\ \pi_1\ \mathsf{else}\ \pi_2) = {} & \{(s, \sigma) \mid \mathcal{B}(b)(s) = \mathsf{tt} \wedge (s, \sigma) \in R(\pi_1)\} \\
& \cup \{(s, \sigma) \mid \mathcal{B}(b)(s) = \mathsf{ff} \wedge (s, \sigma) \in R(\pi_2)\} \\
& \cup \{(s, \omega) \mid \mathcal{B}(b)(s) = \omega\} \\
R(\pi_1; \pi_2) = {} & \{(s, \sigma) \mid \exists s' \in \Sigma : (s, s') \in R(\pi_1) \wedge (s', \sigma) \in R(\pi_2)\} \\
& \cup \{(s, \omega) \mid (s, \omega) \in R(\pi_1)\}
\end{aligned}$$

Note how the last set in the clauses for assignments and conditionals expresses that failures propagate from the expression level to the statement level.

From these relational definitions the following characterizations for the weakest relativized predicate transformer can be derived. The proofs are easy but a bit tedious and hence omitted.

$$\begin{aligned}
\mathsf{wrp}_A.x := e.\psi &= \mathsf{in}_A(e) \vee (\mathsf{def}(e) \wedge \psi[e/x]) \\
\mathsf{wrp}_A.\mathsf{if}\ b\ \mathsf{then}\ \pi_1\ \mathsf{else}\ \pi_2.\psi &= \mathsf{in}_A(b) \vee (b = \mathsf{tt} \wedge \mathsf{wrp}_A.\pi_1.\psi) \vee (b = \mathsf{ff} \wedge \mathsf{wrp}_A.\pi_2.\psi) \\
\mathsf{wrp}_A.\pi_1; \pi_2.\psi &= \mathsf{wrp}_A.\pi_1.(\mathsf{wrp}_A.\pi_2.\psi)
\end{aligned}$$

Note how the disjuncts $\mathsf{in}_A(e)$ and $\mathsf{in}_A(b)$ handle the case of an acceptable failure. As for wp and wlp, sequential composition corresponds to functional composition of predicate transformers.

Loop. The situation gets more interesting for loops. The semantics of a while loop $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$ can be captured in an intuitive way in terms of the following notion of a (b, π)-path [21]: A (b, π)-path is a finite or infinite sequence $p = s_1, s_2, \ldots$ of states in Σ, such that the following conditions are valid.

- Progression: each state in p, except for the last one in the finite case, satisfies b, i.e. $B(b)(s_i) = \mathsf{tt}$ for all $1 \leq i < |p|$, and
- Succession: successive state are related by $R(\pi)$, i.e. $(s_i, s_{i+1}) \in R(\pi)$ for all $1 \leq i < |p|$.

Here the *length* $|p|$ of (b, π)-path is the number of states in p in the finite case and ∞ in the infinite case. A finite (b, π)-path is said to go from s to s' if s and s' are its first and last state respectively. Intuitively, the states in a (b, π)-path represent the intermediate states at the beginning of the loop in a prefix of a computation with $|p| - 1$ iterations of the body.

The relational semantics of a while loop $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$ can now be defined as follows.

$$
\begin{aligned}
&R(\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}) \ = \\
&\qquad \{(s,s') \mid \text{there is a finite } (b,\pi)\text{-path from } s \text{ to } s' \text{ with } \mathcal{B}(b)(s') = \mathsf{ff}\} \\
&\quad \cup \{(s,\omega) \mid \text{there is a finite } (b,\pi)\text{-path from } s \text{ to } s' \text{ with } \mathcal{B}(b)(s') = \omega\} \\
&\quad \cup \{(s,\omega) \mid \text{there is a finite } (b,\pi)\text{-path from } s \text{ to } s' \text{ with } \mathcal{B}(b)(s') = \mathsf{tt} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{and } (s',\omega) \in R(\pi)\} \\
&\quad \cup \{(s,\infty) \mid \text{there is an infinite } (b,\pi)\text{-path starting in } s\}
\end{aligned}
$$

The first set describes the case of regular termination; the other three sets are concerned with the different causes for failures of loops. First, evaluation of the guard could fail; second, the evaluation of the body could fail; and, finally, the loop may diverge.

$\mathsf{wrp}_A.\mathsf{while}\ b\ \mathsf{do}\ \pi$ can be characterized as a (semantic) fixpoint of the equation

$$X \ = \ \mathsf{if}\ b\ \mathsf{then}\ \pi; X\ \mathsf{else\ skip\ od}\ .$$

Not surprisingly, the cases whether divergence is an accepted outcome or not, differ substantially. We have to take the greatest fixpoint w.r.t. $\leq$, if $\infty \in A$, and the smallest fixpoint if $\infty \notin A$.

Alternatively, the relativized predicate transformer of a loop can be characterized by a recurrence on the predicate level. This generalizes and justifies the well-known postulates from [9].

Theorem 9. *Suppose $A \subseteq \Omega$ and $\psi \in Pred$. Then $\mathsf{wrp}_A.\mathsf{while}\ b\ \mathsf{do}\ \pi.\psi$ is the greatest (weakest) solution of the predicate equation*

$$\phi \ = \ \mathsf{in}_A(b) \vee (b = \mathsf{tt} \wedge \mathsf{wrp}_A.\pi.\phi) \vee (b = \mathsf{ff} \wedge \psi)$$

if $\infty \in A$, and the smallest (strongest) solution otherwise.

Due to lack of space, we cannot give the full proof. Let us for explanation just mention that, if we accept diverging loops, i.e. $\infty \in A$, then there are more initial states from which all outgoing computations either satisfy postcondition ψ or have an outcome contained in A. Thus, the solution must have a greater cardinality in this case. This makes it plausible that indeed the weakest solution is the right one.

7 An Application

In order to show the utility of the relativized setup, let us recall one of our examples from Sect. 2. We are going to study a question of the kind 'Is a given transformation (translation) permitted w.r.t. some set of accepted outcomes?'. We consider a simplified version of the dead-code elimination example (Fig. 1). Suppose π and π' are the following programs:

$$\pi \ \stackrel{\text{def.}}{=} \ x := e \,;\, x := f \,;\, P \qquad\qquad \pi' \ \stackrel{\text{def.}}{=} \ x := f \,;\, P$$

The expression f is assumed not to contain x; intuitively, it should thus safely be possible to remove $x := e$ from π as the value of x is over-written immediately. Let us see whether we can justify the transformation from π to π' with the relativized predicate transformers. Note that π can be written in the form $x := e \,;\, \pi'$.

Using the identities from the previous section we obtain the following.

$$\begin{aligned} \mathsf{wrp}_A.\pi.\psi &= \mathsf{in}_A(e) \vee (\mathsf{def}(e) \wedge (\mathsf{wrp}.\pi'.\psi)[e/x]) \\ \mathsf{wrp}_A.\pi'.\psi &= \mathsf{in}_A(f) \vee (\mathsf{def}(f) \wedge (\mathsf{wrp}.P.\psi)[f/x]) \end{aligned}$$

From the assumption that f does not contain x it follows by standard logical arguments that the substitution $[e/x]$ has no effect when applied to $\mathsf{wrp}_A.\pi'.\psi$. Thus, the identity for $\mathsf{wrp}_A.\pi.\psi$ can be simplified.

$$\mathsf{wrp}_A.\pi.\psi = \mathsf{in}_A(e) \vee (\mathsf{def}(e) \wedge (\mathsf{wrp}.\pi'.\psi))$$

Now, deciding whether π' implements π amounts to checking whether $\mathsf{wrp}_A.\pi.\psi$ implies $\mathsf{wrp}_A.\pi'.\psi$ for all predicates ψ. This is certainly the case if $\mathsf{in}_A(e)$ is equivalent to `false`, i.e. does not hold for any state. Indeed in the absence of any further knowledge about e, f and P this is the only safe statement we can make.

What does this mean intuitively? The transformation from π to π' is permissible, if we can be sure that none of the failures potentially produced by e belong to the accepted failures in A. This is in particular the case if A does not contain any arithmetic error, i.e. none of the errors produced by arithmetic expressions.[5] For a more far-reaching conclusion we would need more specific knowledge about e. For example, we might conclude from the fact that e does not contain a division that A might contain the 'div-by-zero' failure.

It is interesting to discuss also the border cases for this example. In the PTC-case we have $A = \emptyset$; then $\mathsf{in}_A(e)$ is equivalent to `false` for trivial reasons. Thus, π' indeed implements π w.r.t. PTC. In the PPC-case, on the other hand we have $A = \Omega$. Then $\mathsf{in}_A(e)$ might be valid for some state if evaluation of e might fail. Thus, the transformation might be invalid in the sense of PPC, depending on the shape of e. So, the formal framework confirms the informal reasoning from Sect. 2.

A similar analysis might be performed for the other examples from that section.

8 Conclusion

In this paper we suggested a semantic framework for performing compiler correctness or refinement proofs in scenarios where optimizations and finiteness of machines are allowed for. The proposed notions of weakest relativized preconditions and the corresponding predicate transformers permit to abandon the

[5] Formally, we call an error ω an *arithmetic error* if there is an expression e and a state s such that $\mathcal{E}(e)(s) = \omega$.

irregular outcomes from the scene in which we actually are working. We have to take them into account only when interpreting the programs in question. Afterwards the actual reasoning can take place in the familiar complete Boolean lattices of predicates and predicate transformers. Nevertheless the obtained correctness results can immediately be interpreted in terms of the more concrete objects of our operational intuition. We see our work as a step towards bridging a gap between elegant theory and practical needs.

This paper draws its motivation partly from work performed in the Verifix project [10] funded by the German DFG (Deutsche Forschungsgemeinschaft), which aims at a fully verified and correctly implemented compiler. Its roots also lie in the ProCoS project [5] in which we pursued a rather comprehensive compiler proof [21] for a prototypic real-time programming language to Transputer code. In that proof monotonic predicate transformers proved to provide a very convenient space that facilitates achieving modularity in the correct construction of the compiling mapping. Modularity is a very important requirement for such an undertaking as otherwise things might easily become unmanageable and untrustworthy. wrp is intended to permit an elegant treatment of runtime errors and finiteness of machines while staying in the familiar and well-studied realm of predicates and predicate transformers. No new theory about predicate transformers is necessary; wrp just provides a different interpretation of programs than wlp and wp, but by objects of the same kind.

For simplicity we have assumed that source and target programs act on the same state space. Of course this is an unrealistic assumption, from a practical point of view. It is, however, a useful idealization if one is mainly interested in considerations concerning control flow implementation. The more realistic situation of different state spaces can be handled with data refinement techniques and Galois connections. For more information on this topic and corresponding references see [21].

Future work includes a more thorough study of wrp and its utility for compiler correctness proofs. More specifically, we are currently investigating the use of wrp for proving the correctness of the translation of nested parameterless procedures to machines with bounded stacks.

Acknowledgments. We are grateful to our colleagues from the ProCoS and Verifix project for many discussions that shaped our view of compiler correctness and verification; a special thank goes to Hans Langmaack for encouraging us to write this paper. The funny example of the MODULA 2 loop was communicated by Gerhard Goos. We also thank Jens Knoop, Hans Langmaack, and an anonymous referee of FM'99 for comments that helped to improve on a draft version.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. K.-R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 2nd edition, 1997.

3. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
4. E. Börger and I. Durdanović. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39(1), 1996.
5. J. P. Bowen et al. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the EATCS*, 50:128–137, June 1993.
6. L. M. Chirica and D. F. Martin. Towards compiler implementation correctness proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
7. J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
9. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
10. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The Verifix approach. In P. Fritzson, editor, *Proc. Poster Session CC'96*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.
11. D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
12. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
14. C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
15. H. Langmaack. Software engineering for certification of systems: Specification, implementation, and compiler correctness (in German). *Informationstechnik und Technische Informatik*, 39(3):41–47, 1997.
16. J. S. Moore. *Piton, A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.
17. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994.
18. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
19. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
20. S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.
21. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, LNCS 1283. Springer-Verlag, 1997.
22. T. S. Norvell. Machine code programs are predicates too. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing. Springer-Verlag and British Computer Society, 1994.
23. E. Pofahl. Methods used for inspecting safety relevant software. In W. J. Cullyer, W. A. Halang, and B. J. Krämer, editors, *High Integrity Programmable Electronics*, pages 13–14. Dagstuhl-Sem.-Rep. 107, 1995.
24. S. Sippu and E. Soisalon-Soininen. *Parsing Theory Vol. I*. Springer-Verlag, 1988.
25. W. M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, 1984.
26. R. Wilhelm and D. Maurer. *Übersetzerbau*. Springer, 1992.